UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Dissertation

Fault Attacks and Countermeasures

Martin Otto
martin@martin-otto.de

Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Universität Paderborn

Betreuer: Prof. Dr. Johannes Blömer
Dezember 2004

for my parents

Acknowledgments

During the course of writing my Ph.D. thesis, many people have been supporting me and my work, and this thesis is a good opportunity to express my gratitude for them.

First of all, I would like to thank my supervisor, Prof. Dr. Johannes Blömer, for his great support. He introduced me to the very interesting field of fault attacks and showed me in many helpful discussions how to improve my work and suggested in which direction my research should go. The atmosphere in his research group was very creative. In particular, Johannes offered me the opportunity to do the work in my own style and time.

I would also like to thank Prof. Dr. Joachim von zur Gathen, who has been my second supervisor in the PaSCo graduate school, and who introduced me to the field of cryptography. Additionally, I would like to thank Dr. Jean-Pierre Seifert from Intel, with whom I cooperated closely. He provided great insight in various problems and was a great help in understanding the practical issues of cryptography on smartcards from the viewpoint of commercial applications.

Then I would like to thank Valentina Damerow, Thomas Leyer, Mirko Hessel-von Molo and Dr. Alexander May for carefully reading parts of my thesis and helping me improve its quality with many valuable comments.

I enjoyed the friendly and relaxed atmosphere at my work place, both in the research group of Johannes and in the research group of Prof. Dr. Friedhelm Meyer auf der Heide, and especially working together with the office crew, Sabrina Geissler, Birgitta Grimm, Marcel R. Ackermann, Tobias Selms, and Dirk Pommerenke. I have to thank Oliver Vogel for writing X-Blast, which served perfectly as an opportunity for short distractions.

My research was made possible by two grants, for which I am very thankful. First, I was a member of the DFG graduate school No. 693 “Scientific Computation: Application-oriented Modelling and Development of Algorithms”, hosted by the Paderborn Institute for Scientific Computation (PaSCo). Here, my work was supported beyond financial coverage by the interdisciplinary environment of a graduate school with people from several different countries. This offered some great opportunities to present and discuss my work with people from different scientific areas. In the last phase of my research, I received a grant from the University of Paderborn, allowing me to finish my thesis.

I would also like to thank a bunch of people for support during various stages of my time at the University of Paderborn, especially Barbara Fritsch, the people from ”mafia”, and Marcus Nachtkamp, who created the birds.

Last, but not least, I would also like to thank my family, who has been a great support during all times.

I had a lot of fun doing research at the University of Paderborn and especially in the field of fault attacks. I hope the reader will sense some of the enthusiasm while reading this thesis.

Martin Otto
Paderborn, December 2004

Contents

List of Figures

List of Tables

Introduction

Exchanging messages securely between two parties, without allowing a third party to wiretap the conversation, is a problem, which is thousands of years old. It has been addressed by cryptography, i.e., by encrypting messages, as early as in ancient Egypt around 1900 B.C., where a scribe used non-standard hieroglyphs in an inscription [Sch00b, p. 86]. The ancient idea of encrypting messages is to use symmetric encryption, i.e., a cryptosystem where both communicating parties share the same secret key. Obviously, this common key has to be exchanged safely at some time, which is a major problem when communicating with people over large distances. Using cryptography has long been too cumbersome to be deployed beyond intelligence agencies and the military.

However, in 1976, Whitfield Diffie and Martin Hellman [DH76] came up with a new direction in cryptography: asymmetric cryptosystems. Asymmetric cryptosystems, or *public key cryptosystems*, allow to use two different keys, a public key and a private key. The public key is solely used for encrypting a message. Decrypting a ciphertext cannot be done with the public key. To decrypt a ciphertext, the secret key must be known. If it is infeasible to compute the secret key from the public key, the public key may be published freely. This solves the problem of exchanging the keys, which prevented widespread use of symmetric cryptosystems. Consequently, this revolution in cryptography triggered the use of cryptography in numerous applications. Nowadays, cryptography is at the core of everyday life, securing almost all aspects of, e.g., communication via the internet, e-commerce, and online banking. It is a crucial cornerstone, which makes the global economy, and especially the internet work. There is subtle irony in the fact that making information public solves major problems of exchanging secret messages.

The basic principle of asymmetric cryptosystems are *one-way functions*. A one-way function is a function, which is easy to compute, but computationally infeasible to invert. Computing a one-way function corresponds to encrypting a message, and inverting the function corresponds to decrypting a ciphertext. However, to make decryption possible, the secret key serves as additional knowledge, which allows to invert a one-way function easily. Such additional knowledge is referred to as a trapdoor, and such a one-way function as a *trapdoor one-way function*. These functions are the underlying structure of every public key cryptosystem.

In 1978, Ronald Rivest, Adi Shamir, and Leonard Adleman [RSA78] proposed the first public key cryptosystem: RSA. The RSA encryption function is based on modular exponentiation modulo a composite number $N = p \cdot q$, where p and q are primes. A message m is exponentiated with a public key e to yield a ciphertext $c = m^e \bmod N$. Given the secret key d, the ciphertext can be decrypted by computing $m \equiv c^d \bmod N$. It should be mentioned that asymmetric cryptography as well as the RSA cryptosystem had already been developed a few years earlier by James Ellis, Clifford Cocks, and Malcolm Williamson, however, they worked for a secret British agency named GCHQ and were not allowed to publish their results [Sch00b]. Other cryptosystems have been proposed thereafter, e.g., the ElGamal cryptosystem [ElG85], which is based on the discrete exponentiation function in groups, e.g., the group of points on an elliptic curve. The RSA and the ElGamal cryptosystems are described in detail in Protocol 2.1 and Protocol 4.3, respectively.

It has not been proved by anyone, that the RSA encryption function really is a trapdoor one-way function. And even worse: it is not known whether one-way functions exist at all. Still, public key cryptosystems are being used extensively with the help of functions, which are *candidate one-way functions*, i.e., functions, which are hoped to be and which are generally assumed to be one-way functions.

The RSA cryptosystem has survived more than 25 years of attacks. While details had to be adjusted, the basic idea of the RSA cryptosystem is still unbroken, and the RSA system is used extensively today. Hence, it is reasonable to put some confidence in the believe, that the RSA encryption function as used today is a very good candidate for a one-way function.

Asymmetric Cryptography and Digital Signatures

The security of modern cryptography relies on various unproved assumptions. In this thesis, we will show that some of these assumptions do not hold in practice. This observation allows for new and partly devastating attacks, and this thesis aims to add knowledge to this field.

In this thesis, we focus on digital signature schemes. In principle, many public key cryptosystem can also be used as a digital signature scheme. Here, one party, named Alice, decrypts a message using her secret key to obtain a digital signature. Then, another party, named Bob, can verify the signature by encrypting the digital signature with the public key. If the result is equal to the message, the message is believed to be authentic.

Secure digital signatures are a second major area of modern cryptography. They have been motivated by the needs of commerce and communication via the internet. If two parties want to agree on a contract via the internet, both have to sign the contract. A digital signature scheme can be used to provide authentication of data, data integrity, and non-repudiation. Moreover, applicability in the real world requires that digital signatures are also recognized by courts. Today, digital signature schemes exist, which can be used as a legal alternative for handwritten signatures, i.e., as specified by the German Signaturgesetz [Sig01a], [Sig01b]. However, their secure use must be guaranteed. And guaranteeing security is all but an easy problem.

Security of Modern Cryptosystems: The Standard Approach

Let us first review the standard terms, in which security of a cryptosystem used for digital signatures is measured. Here, three aspects are considered. First, one has to define the objective of an adversary, e.g., what the goal of his attack is. Second, one has to agree on some model, in which security is defined, i.e., a model to measure security. And third, one has to define the

power of the adversary, i.e., what kind of attacks he is able to mount on a system. Given these three aspects, researchers try to prove that a certain adversary cannot achieve a specific goal with respect to some notion of security. Upon existence of such a proof, a system is considered to be secure. Since we are mainly concerned with signature schemes in this thesis, we will describe all these aspects in terms of digital signatures.

Objectives. For a signature scheme, an adversary may have one of three different objectives. First, he might want to achieve a *total break* of the system. Here, he wants to recover the secret key, which enables him to sign any arbitrary message and to act as the legitimate signer without any chance of distinguishing between them. Second, he could be satisfied with a *selective forgery*. A selective forgery means that if the adversary is given a message m, he is able to compute a valid signature for this message. This does not need to imply that the adversary recovers the secret key. And third, as the weakest goal an adversary might have, there is *existential forgery*. Here, an adversary's objective is to compute a pair (m, s), such that m is a message and s is the valid signature of that message m. The message m might not be selectable by the adversary and might not represent any meaningful text. The total break is the strongest objective, while the existential forgery is the weakest objective.

Models of Security. A naive approach to security would demand that a signature scheme is secure if the weakest objective, i.e., existential forgeries, is impossible to achieve by the strongest adversary possible. However, this approach is not practical. It is clear that for any valid message, a valid signature exists, hence, an adversary can always construct valid message/signature pairs by simply guessing the signature with some non-zero probability. Hence, the best thing to prove is that an adversary's success probability is not significantly higher than the probability that a randomly chosen signature is a valid signature of a given message. In practice, the following two models of security are considered for public key cryptosystems, although others exist as well (cf. [MvOV96]).

The most practical model of security is *computational security*. To have confidence in the security of a proposed cryptosystem, it should be computationally secure. For this notion of security, an adversary is assumed to have limited resources, i.e., time and memory polynomial in the input parameters. Security is based on the fact that the problem of breaking a system is related to solving a problem, which is assumed to be computationally hard. Here, "hard" does usually not represent NP-hard, but the assumed hardness of a well-studied problem. For example, some popular systems are reduced to the integer factorization problem (e.g., RSA) or the discrete logarithm problem (e.g., ElGamal). However, simply reducing the problem of breaking a system to solving some hard problem is not enough. It is desired, that the two problems are computationally equivalent. If this can be shown, a cryptosystem is called *provably secure*. Both, the RSA cryptosystem and the ElGamal cryptosystem exist in provably secure variants, which depend on different assumptions about the hardness of some problems.

Another possible model of security is *ad-hoc security*. Here, security only relies on extensive research and heuristics. No security is guaranteed, yet, if a large number of researchers tried to break a cryptosystem for a significant amount of time, if it is secure against standard attacks, and if there are good arguments why breaking the system should be computationally infeasible, one might be tempted to trust the cryptosystem. However, here is no proof and the system could be broken the very next day by some clever and new idea.

Attack Models. In a signature scheme, it is general consensus that an adversary always has both, access to all data being transmitted by two communicating parties and exact knowledge of every aspect of the used signature scheme, with the only exception of the secret key. Hence, the security of a system must rely on the secret key only. An adversary might also be able to eavesdrop on a conversation with many signed messages being exchanged. This situation is captured in an attack model named an *adaptive chosen message attack*. Here, the knowledge of the adversary about various pairs of messages and valid signatures is modeled by the assumption that an adversary has access to a signature oracle. An adversary can request signatures of polynomially many (in the size of the input parameters) chosen messages to achieve his objective. He may choose the messages to be signed adaptively, i.e., based on all message/signature pairs collected before. Obviously, in order to achieve a forgery, he must be able to construct a new, previously unseen message/signature pair. Any practical system should be secure against such an adversary. It might be argued that for some special setups, an adaptive chosen message attack is impossible. In this case, weaker attack models exist (see [MvOV96, § 11.14] for details).

The basic assumption about an adversary is a harsh restriction on his knowledge. In all attack models, the adversary is limited to knowledge of a certain number of input/output pairs, i.e., message/signature pairs, and knowledge about the specification and implementation of the cryptographic protocol (the latter is usually referred to as Kerckhoffs' Assumption). This scenario is a black-box assumption, depicted in Figure 1. It allows purely theoretical proofs on paper, without having an algorithm implemented, without having to actually use a system.

Figure 1.: Black Box Assumption

Given the black-box approach, numerous cryptosystems can be proved to be secure. However, what is the practical relevance of such a proof? If you consider a cryptosystem alone in a vacuum, you would probably be fine with such a security proof. However, imagine that this system is not used in a well defined mathematical environment, but in the real world. Here, cryptographic protocols are implemented on computers, which adhere to the laws of physics. Would you still be confident?

Security of Modern Cryptosystems: Getting Real

Cryptographic protocols are used in the real world, and this has severe implications for the term "security". In the real world, cryptosystems have to be implemented using some programming language or hardware circuits, which have to be run on some sort of computer, and human beings use it. Therefore, cryptography does not exist in a vacuum and security does not only depend on mathematical properties. In real life, an adversary has a lot of possibilities to break a "provable secure" cryptosystem.

Real World Attacks. In the real world, an adversary can go beyond the mathematical concept and attack the implementation rather than the specification. Obviously, the real world offers many possibilities for attacks, which cannot be modeled or prevented by mathematics. For example, an adversary could force the legitimate owner to disclose his secret key. Or, he could

exploit the fact that today's secret key lengths are far beyond the abilities of humans to memorize them. Hence, a 2048 bit RSA key would be stored somewhere and would be protected with, say, an 8-letter password. Here, an adversary could hope that the password is chosen as the name of the husband or wife of the legitimate owner. Or, an adversary can hope that a programmer, implementing a mathematical concept in some programming language, uses unsecured shared memory to store the secret key. Or, he could try to transmit a virus or a trojan horse on a users computer, which dumps the memory content while a cryptographic application reads the secret key.

The given examples show that security in the real world depends on some assumptions, cryptographers silently agree on. First, it is assumed that all parties play by the rules and follow the specification. This is a reasonable assumption. However, it is also assumed that a cryptosystem running on some device cannot be attacked by other means than described by the black box model. This assumption is not realistic. One major problem for security in the real world is the environment, in which a cryptographic algorithm is executed. The device running an algorithm, as well as the operating system, must represent a safe harbor. A cryptographic protocol cannot be secured against a hostile operating system, which analyzes the code while it is executed.

Now, it is common perception that personal computers are all but secure platforms. The extensive occurrence of viruses, worms, and trojan horses shows that a desktop computer can easily be compromised. This implies that cryptographic applications cannot be guaranteed to be secure if run on an ordinary computer. However, signature schemes, which are intended to be used as a legal alternative to handwritten signatures, must be guaranteed to achieve a certain level of security (cf. [Sig01a], [Sig01b]). This triggered the use of dedicated and specially secured hardware for cryptographic purposes, especially for digital signatures.

Using Smartcards. Smartcards have long been considered as an ideal environment for digital signature schemes. They do not offer any possibility to have new code or updates being loaded onto them, hence viruses and other malicious code are no longer a problem. Moreover, smartcards have a significant computational power if compared to their size. Modern smartcards are faster and have more memory than a C64 standard computer used in the 1980s. Hence, even cryptographic schemes, which involve a significant amount of arithmetic operations, e.g., elliptic curve cryptosystems, can be run on them. There is no need to hope for a friendly operating system, since the complete system is provided by the manufacturer, including the hardware. A smartcard is designed for the only purpose of a specific task. This makes it easier to test a system thoroughly for bugs and other weaknesses. Moreover, smartcards are small and easy to carry around, which makes them highly attractive, since they ease the use of digital signatures in everyday life. Hence, smartcards seem like an ideal environment for digital signature schemes.

Side-Channels. However, smartcards are electronic devices. They must obey the laws of physics. Hence, if a smartcard computes a result, it requires a certain amount of time and a certain amount of energy, the electronic circuits emit a certain amount of radiation, energy, and even sound, and they may be affected by their environment. Since smartcards are not equipped with an own power source or an own clock signal generator, they have to be connected to a *smart card reader*. This reader can easily measure, e.g., time and power consumption of the smartcard. If any of this data is somehow correlated to secret data — and this is of course the case for digital signature computations — an adversary gets additional information. This situation is not captured by the black box model depicted in Figure 1. These additional sources of information are referred to as *side-channels*. Reality cannot be modeled as a black box, it only

allows for a "gray box", where an adversary has access to several side-channels. This situation is captured in Figure 2.

Figure 2.: Real World Assumption

It has been shown by various authors that a large number of these side-channels provide information, which reveals important and compromising details about secret data. Some of these details can be used as new trapdoors to invert a trapdoor one-way function without the secret key. This allows an adversary to break a cryptographic protocol, even if it proved to be secure in the classical, mathematical sense. The various side-channels include timing measurements ([Koc96b]), power consumption and the power profile ([KJJ99]), electromagnetic emissions ([QS01], [RR01], [GMO01]), sound ([ST04]), presence and abuse of testing circuitry ([Koc96a], [YWK04]), data gathered by probing circuitry or bus lines ([HPS99]), cache memory behaviour ([Pag02]), and faults ([AK96], [BDL97]). Research in side-channel attacks is still a vivid area of research, hence, additional side-channels may be discovered on a daily basis.

Fault Attacks. This thesis concentrates on one specific side-channel: faults. Here, an adversary induces faults into a device, while it executes a known program, and while the adversary observes the reaction. These attacks are named *fault attacks*, and they are fundamentally different from other side-channel attacks. Other side-channel attacks are passive attacks, which just listen to some side-channel without interfering with the computation. Fault attacks are active attacks, where an adversary has to tamper with an attacked device in order to create faults, thereby opening the desired side-channel. Smartcards have long been considered to be *tamper-proof* devices, until an article by Ross Anderson and Markus Kuhn [AK96] suggested that smartcards may at most be *tamper-resistant*, but definitely not tamper-proof. If an adversary can inflict some sort of physical stress on the smartcard, he can induce faults into the circuitry or memory. These faults become manifest in the computation as errors. If an error occurs, a faulty final result is computed. If the computation depends on some secret key, a comparison between correct data and faulty data may allow to conclude facts about the secret key.

The first successful fault attacks have been reported by Dan Boneh, Richard DeMillo, and Richard Lipton from Bellcore Labs in 1997 [BDL97]. They presented two important attacks on variants of RSA, used for computing digital signatures. Their first attack targeted CRT-RSA, a fast variant of repeated squaring, which will be discussed in detail in Chapter 3 in this thesis. Here, a single faulty result may allow an adversary to completely break the given instance of RSA. A second attack on RSA targeted repeated squaring. It is described in detail in Chapter 2. Here, a secret key d of length $n := l(d)$, where $l(d)$ denotes the binary length of d, can be recovered with probability at least $1/2$ given $O(n \log(n))$ faulty signatures.

These results triggered extensive research in the field of fault attacks. In the following, several authors extended the ideas from Boneh, DeMillo, and Lipton to other cryptosystems, using other fault models and different means of physical attacks. Chapter 1 will discuss known physical attacks and fault models in greater detail. Some attacks, referred to as *oracle attacks*, do not even need a faulty result at all, they deduct information about the secret key using only the

information whether a fault changed the normal behaviour of a device or not. Fault attacks have been mounted successfully on symmetric and asymmetric cryptosystems alike, and they have even been used to break completely unknown ciphers [BS97]. Most fault attacks reported so far are attacks, where an adversary needs to compare the final result to some known correct data to verify that a fault occurred at all.

The Threat is Real. Fault attacks are a practical scenario. Fault attacks have been used to break security mechanisms even before the cryptographic community became aware of them. Pay TV card hackers used rapid transient changes in the clock signal, called *clock glitches*, to access pay TV channels before 1996 [AK97]. Other possible scenarios are easily sketched, which show that fault attacks are a real threat to smartcards. For example, imagine a customer using his signature smartcard in some Mafia shop, where the smartcard reader is provided by the hostile seller. In this case, the seller can attack the smartcard and induce faults even under the eye of the customer, who only sees the reader. Since smartcards are fast, a seller can easily attack the card a few times, before initiating the legitimate transaction. Obviously, by taking a stolen smartcard to a laboratory, an even wider variety of attacks is possible. Hence, fault attacks can be mounted on smartcards in reality.

Security Model. Since the threat of fault attacks is real, both customers and hardware manufacturers are looking for secure smartcards. Here, two approaches are possible. First, hardware countermeasures can be applied to detect and prevent known methods of inducing faults, e.g., detectors for clock glitches. Second, new algorithms can be developed, which are immune against fault attacks.

Research in fault attacks is relatively new, with the first attacks having been reported in 1997 [BDL97]. New attacks are still discovered frequently, and new physical attacks allow to induce faults with an increasing variety of physical setups. Therefore, the scientific community has not yet been able to develop a theoretical framework to allow general security proofs for algorithms supposedly secure against fault attacks. Some approaches have been made, but they are not satisfying yet.

For passive side-channels, it is an obvious approach to ensure that the available information, e.g., the power profile, is independent of the secret key. A lot of countermeasures against various attacks relying on power consumption traces, time measurements, or electromagnetic emission measurements have been proposed, but it is still challenging to develop algorithms, which are secure against *all* passive side-channel attacks. It happened often that a countermeasure intended against one side-channel attack did not protect against another side-channel attack (e.g., [OS00], [Wal03], [OH03]), or even benefited another attack (e.g., [JQYY02], [YKLM01a], and a similar situation will be described in Chapter 4 in this thesis). There have been some approaches trying to unify the different passive side-channel attacks (e.g., [KW03]), and to develop a theoretical framework (e.g., [JQYY02], [CJRR99]), [CKN00], [CCJ03], [MR04]). However, the issue of provable security for passive side-channel attacks is far from being settled.

For active attacks, i.e., fault attacks, the situation seems easier at first sight. If we assume that an adversary can only induce a single fault, it is sufficient to run an algorithm three times on the same input and return the majority vote. This provides security against such an adversary. However, this approach is not satisfying for practical applications, since it slows down a smartcard significantly. This is not acceptable in practice. Hence, new notions of security against fault attacks are desperately needed. There have been some approaches, e.g., [Itk02], [Itk03], [MR04], and [Wag04], however, none of these models allows for an efficient

scheme, no practical scheme could be proved secure in these models. Nonetheless, algorithmic countermeasures against fault attacks must be developed, which can be used in the real world. Until a suitable model for security against fault attacks has been developed, the best way to go is to show that a given scheme is secure against all known attacks. This resembles the ad-hoc security model defined for classic security analysis, and it is undoubtedly the weakest notion of security possible. However, this is the best notion available today.

Our Research. Our research, which culminated in this doctoral thesis, investigated fault attacks on asymmetric cryptosystems only. It was motivated by two goals: first, we were interested in finding new fault attacks and new fault types, extending the attacks of Boneh, DeMillo, and Lipton. However, fault attacks also establish the need for algorithms, which are secure against them. Therefore, as our second goal, we were searching for new algorithms, which are not susceptible to fault attacks. We have been successful in both areas and present our results in this dissertation. We present results for new attacks and for countermeasures for both, variants of the RSA cryptosystem and variants of elliptic curve cryptosystems. For both of our countermeasures, we will prove the new algorithms to be secure against previously reported fault attacks.

Organization and Main Results

We briefly describe the organization of this thesis and present the main results.

Chapter 1. We start with a full characterization of fault models in Chapter 1. Here, we first review all physical attacks, which have been used successfully to induce faults into smartcards. Afterwards, we propose a new characterization for fault models as it is needed today. We describe in detail the known fault types, the most important aspect of any fault model. Afterwards, we define five different fault models, which capture all reasonable fault models used in previous attacks. These definitions will be the basis for the remaining chapters.

Chapter 2. In Chapter 2, we present new results for fault attacks on plain RSA, i.e., on modular exponentiation via repeated squaring. First, we start from the attack on plain RSA as introduced by Boneh, DeMillo and Lipton in [BDL97]. We exhibit and correct two minor flaws in the original attack. Afterwards, we extend the original fault attack, which has been described for the right-to-left version of repeated squaring, to its twin version, the left-to-right repeated squaring algorithm. Here, we show that the bits of the secret key can be recovered both starting from the least significant bits, as well as starting from the most significant bits.

Chapter 3. A fast alternative to repeated squaring is the CRT-RSA algorithm, due to Christophe Couvreur and Jean-Jacques Quisquater [CQ82]. Due to its advantage in speed, this algorithm is widely used on smartcards. However, it has been shown to be particularly susceptible to fault attacks like the "Bellcore attack" described in [BDL97]. In Chapter 3, we present a new version of the CRT-RSA algorithm, which is secure against the Bellcore attack in the most realistic fault model. We will also show how the algorithm can be modified to be used in stronger, less realistic fault models. The main results of this chapter have been published in [BOS03] as a joint work with Johannes Blömer and Jean-Pierre Seifert.

Chapter 4. After presenting attacks and countermeasures for the RSA cryptosystem in the previous two chapters, we turn to elliptic curve cryptosystems in Chapter 4. We first introduce elliptic curves and review previously published attacks. We will show that contrary to previous beliefs, it is possible to induce faults during the elliptic curve repeated doubling algorithm, which are not detected by the standard countermeasure proposed by different authors. This will be shown by a brief but careful analysis of the error propagation during elliptic curve point addition. Since this analysis is rather technical, detailed results beyond instructive examples are presented in Appendix A.

The results of the analysis give rise to a new fault type, *Sign Change Faults*. We end the chapter be defining this fault type and by describing how an adversary can induce faults of this type. We show that this yields a new realistic fault model. Sign Change Faults are the basis for the next three chapters.

Chapter 5. In Chapter 5, we use the new Sign Change Fault Model motivated by Chapter 4 for fault attacks on various versions of elliptic curve repeated doubling, i.e., left-to-right and right-to-left NAF-based repeated doubling, classic binary expansion-based repeated doubling, and Montgomery's Binary Method. We will show that the basic scheme of fault attacks as described in Chapter 2 can also be applied successfully to elliptic curve repeated doubling. Therefore, the new fault model is a new threat for elliptic curve cryptosystems. Parts of the results presented in this chapter have been published together with results from Chapter 6 in [BOS04] as a joint work with Johannes Blömer and Jean-Pierre Seifert.

Chapter 6. Given the new sign change fault attacks presented in Chapter 5, a new secure version of repeated doubling on elliptic curves is presented in Chapter 6. This countermeasure against Sign Change Faults seizes ideas already used successfully for the countermeasure against fault attacks on CRT-RSA presented in Chapter 3. It can be used with any existing algorithm for elliptic curve repeated doubling, which does not require field inversions. We show that our new algorithm is secure against Sign Change Attacks and previously reported attacks. The countermeasure presented in this chapter has been published together with results from Chapter 5 in [BOS04] as a joint work with Johannes Blömer and Jean-Pierre Seifert.

Chapter 7. In Chapter 7, we return to modular exponentiation and the RSA cryptosystem. We show that Sign Change Attacks are also a threat against the RSA cryptosystem. However, Sign Change Attacks are much harder to realize for RSA type cryptosystems than for Elliptic Curve cryptosystems. Moreover, they are successful only against the right-to-left repeated doubling algorithm.

1. Fault Attacks and Fault Models

Since the first report of a successful fault attack in [BDL97], many researchers have published results about fault attacks. However, there have been few attempts for a unifying characterization of fault models. Most researchers have implicitly assumed certain fault models, others have left important details undefined. Therefore, in this section, we will present a full characterization of fault models, as they need to be defined today.

Theoretically, an adversary may imagine any kind of error, e.g., that a device outputs the secret key instead of the result of the computation. This may lead to the definition of any fault model imaginable. Consequently, research in fault attacks established the habit that if a certain fault model is used, it is also described how such faults can be realized in the real world. Otherwise, the result may still be of mathematical interest, however, it does say little about the security of today's computers. In this view, research on fault attacks takes an engineering point of view (cf. [YKLM01a]), since it is oriented by real devices and the real physical world. Therefore, it is crucial to motivate all fault models from the real world.

Consequently, we will start this section by reviewing the most important methods known to the public to induce faults into devices, i.e., smartcards, in Section 1.1. This will show that such methods are numerous. However, a mathematical view on inducing faults is needed for profound theoretical analyses, to prove that attacks can recover secret information and break a system, and to prove that algorithmic countermeasures really work. Hence, we need a theoretical framework. This framework relies on carefully specified fault models. We will start with a full characterization of fault models in Section 1.2. The most important aspect of any fault model is the assumed fault type. Therefore, we devote a whole section, Section 1.3, to the discussion of all fault types used today in the open literature. Finally, in Section 1.4, we will define the most prominent fault models, which are used throughout the literature. These definitions will be the basis for all results in the remaining chapters of this thesis.

Fault attacks can exploit faults in two different ways. One way is to cause the attacked device to malfunction and to output a faulty result. This result is then used to derive secret information. However, there are also fault attacks, which do not use the actual faulty result for computations, but only the information whether the final result was faulty or not. Such attacks are called *oracle attacks*. In the remainder of this thesis, almost all fault attacks use faulty results to compute secret data. For completeness, we will discuss oracle attacks in Section 1.6.2.

First, the term "fault attack" needs to be clarified to avoid misinterpretation. This is due to the different notions of attacks, faults and errors. As a *fault attack*, we understand a complete method, approach, or algorithm, which is given a device and which returns secret data. During the course of a fault attack, an adversary may run the device several times while *inducing faults* into memory cells or other structural elements of an attacked device. These faults are induced by some *physical attack*, i.e., some physical setup, which exposes the device to some sort of physical stress. As a reaction, the device malfunctions, i.e., memory cells change their current, bus lines transmit different signals or structural elements are damaged. All these effect will be referred to as *faults*.

In this thesis, we are not concerned with the actual physical realization of inducing faults.

Therefore, we will translate the physical events into a mathematical formulation. This allows us to identify memory cells with their values and to speak of faults being induced into variables or bits and bytes. Hence, we are only concerned with the effect of a fault as it manifests itself in modified data or a modified program execution. For simplicity, we say that a fault attack induces faults into a variable, such that the attacked variable is *faulty*, i.e., it has a different value. This difference is described by the *error term*. For an unprotected device, a fault affecting a single memory cell will usually have the effect that a single bit of a variable is modified. However, in the presence of countermeasures, the effect of such a fault may yield a completely different error term.

Given a variable x, which is targeted on purpose or hit by accident during a fault attack, the actual induction of a fault will be expressed as a function acting on the variable x, denoted by $x \overset{\lightning}{\mapsto} \tilde{x}$, where $\tilde{x}$ denotes the modified variable x. We will always characterize the value of $\tilde{x}$ by the *error term* or *error value* $e(x)$, capturing the difference between the original value of x and its faulty value $\tilde{x}$, i.e., $\tilde{x} = x + e(x)$. Here, $e(x)$ is a variable, which takes values according to some fault model.

1.1. Inducing Faults in the Physical World

Fault attacks exploit the physical properties of devices. Theoretical fault attacks are based on fault models, which in turn model physical behaviour of attacked devices. In this section, we will give an overview over actual physical methods to induce faults. This will show that there are numerous ways to induce faults into physical devices. Since this dissertation focuses on the theoretical and algorithmic aspects of fault attacks, the description of the physical and technical details will be brief and rough.

Cosmic Rays. Research in electrical equipment used in aviation or space travel found that cosmic rays can flip single bits in the memory of an electronic device. Such faults have been called "single event upsets" in space travel, and they are an issue since the first days of space flight. Cosmic rays are very high-energy subatomic particles originating in outer space. They are comparable to high-energy protons and neutrons produced by large particle accelerators. Relying on cosmic rays means that an attacker has to wait until a desired bit is flipped by a cosmic ray by chance. According to [GA03], in 1996 the DRAM cells of a typical PC were expected to suffer such a bit flip about once per month [ORT+96]. Since then, miniaturization decreased the susceptibility of DRAM cells, such that now the authors in [GA03] expect that an adversary has to wait "for several months" for a bit flip caused by cosmic rays.

Particle accelerators can be used to simulate cosmic rays, however, there is no public access to such equipment. An alternative for particle accelerators are Americium-Beryllium sources as a source for high-energy ionizing radiation, since they produce neutrons [Nor96]. Such sources are used in oil exploration and although access is regulated, at least corporate or government attackers can use such sources for attacks easily. However, the main problem with attacks based on high-energy rays is to achieve a sufficient targeted precision.

A summary about the susceptibility of memory cells to cosmic rays and similar high-energy particle sources can be found in [GA03] (which we used as the source for our description above) and in [BCN+04].

α-, β-, and X-rays. α particles are helium nuclei, which are typically emitted by a radioactive source. Such sources can be easily acquired by anyone, e.g., a weak source is used in smoke detectors. Since α particles cannot penetrate thick layers of packaging, they only have a very limited chance of penetrating the packaging of the chip of a modern smartcard. Evidently, in the past, chip packaging material used to be contaminated by radioactive sources frequently, such that the packaging material itself would serve as a source for α particles. Additionally, the impurities in the silicon often served as a source for α particles. Since quality assurance became aware of that problem, such contamination does not occur anymore these days. The same holds for β particles, i.e., high-energy electrons, which interact strongly with plastic and metal packaging material such that they have only a negligible probability of inducing usable faults unless the coating is removed.

For X-rays, the situation is different. Although standard commercial X-ray sources, e.g., an airport baggage scanner, produce X-rays which have not enough energy per particle to interact with DRAM circuitry, there are high-energy "hard" X-ray sources, which "might possibly do the job" [GA03]. All three approaches can flip single bits in memory if applied successfully, however, targeting a specific bit is very difficult.

Again, we refer the interested reader to [GA03] or to [ZCM+96] for details. We have used [GA03] as the basis for our description above.

Heat / Infrared Radiation. Typically, electronic equipment only works reliable in a certain range of temperature. If the outside temperature is too low or too high, faults occur. Every PC has a fan to ensure that the heat produced by the internal circuitry does not overheat the computer.

An experiment where infrared radiation coming from a simple 50-watt spotlight clip-on lamp together with a variable AC power supply has been used successfully to induce faults into a desktop PC (cf. [GA03]). This attack succeeded to induce single bit flips for temperatures between 80° and 100° Celsius. However, the experiment also showed that unless finely tuned, the attacks often cause the operating system to crash, thus requiring a complete reinstallation. For portable devices such as smartcards, a heating source can be easily focused at the device, while attacking a PC requires to open it or to disable the fan. However, changes in memory due to heat usually affect a large area, i.e., many bits.

The authors of [GA03] also suggest to heat up specific memory chips by exercising them repeatedly, e.g., by a large number of load/store operations. Inducing faults using temperature deviations has been considered as an example for inducing faults already in the earliest publications on fault attacks, e.g., [BDL97], [Pet97]. In [Koc96a], the author explains how heat helps to induce biased faults, which only change ones to zeros or vice versa.

Power Spikes. A smartcard is a portable device without any own power supply. Hence, it always requires a *smartcard reader* providing it with power in order to work. This reader can be easily replaced by an adversary with laboratory equipment, capable of both tampering with the power supply as well as measuring power consumption (thus enabling power attacks). It has been defined by standards, e.g., [ISO02b], that a smartcard must tolerate a certain variation in the power supply V_{CC} (see Figure 1.1) of $\pm 10\%$ of the standard voltage of 5V. However, if the variation is significantly higher than 10%, the card is no longer required to work properly. In fact, short massive variations of the power supply, which are called *spikes*, can be used to induce errors into the computation of the smartcard. Spikes allow to induce both memory faults as well as faults in the execution of a program (code change attacks), cf. [KK99]. Both can be used

to affect an arbitrary number of bits, starting with single affected bits. Code change attacks, which aim at confusing the program counter, can cause conditionals to work improperly, loop counters to be decreased (e.g., for DES), and arbitrary instructions to be executed (see Section 1.6.1). Spikes can have different effects depending on nine different parameters, including time, voltage value, and the shape of the transition.

In [ABF+02], the authors describe in detail experimental results for inducing faults into smartcards using spikes. Spikes are cited as a standard example for methods to induce faults by various authors, e.g., [NM96], [BDL97], [BS97], [Pet97], [KR97], [Wei00], [BDL01], [BS03], [PQ03], [YMH03], [BCN+04], [KKT04].

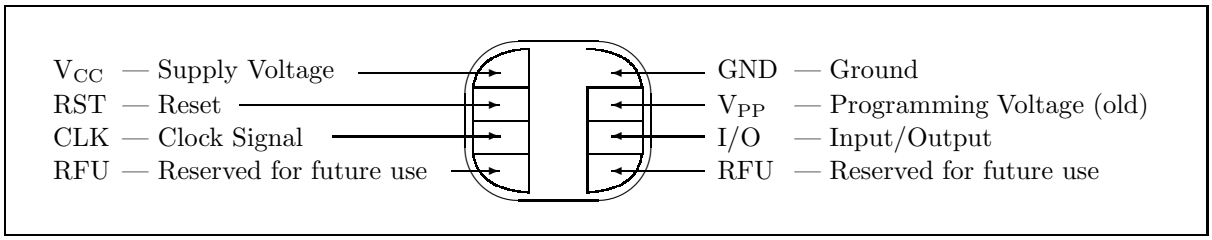


Figure 1.1.: Assignment of the contacts of a smartcard as defined in ISO 7816-2 [ISO02a].

Clock Glitches. Similar to the power supply, smartcards do not create their own clock signal either. Although modern high-end smartcards use a randomized clock, they only randomize the clock signal provided by the external card reader. Similar to the supply voltage, smartcards are required to tolerate a voltage variation in the clock signal CLK (see Figure 1.1), where the high signal V_{IH} may range from $0.7 \cdot V_{CC}$ to V_{CC} and the low signal V_{IL} from 0 to $0.5 \cdot V_{CC}$, where V_{CC} is the power supply voltage. The smartcard must also work properly with deviations of clock rise and clock fall times of 9% from the standard period clock cycle (cf. [BS03], [ISO02b]). Smartcards are usually provided with a 3.5 MHz signal. Since the adversary may replace the card reader by laboratory equipment, he may provide the card with a clock signal, which incorporates short massive deviations from the standard signal, which are beyond the required tolerance bounds. Such signals are called *glitches*. Glitches can be defined by a "huge range of different parameters" [BS03], and they can be used to both induce memory faults as well to cause a faulty execution behaviour (code change attacks). Hence, the possible effects are the same as for spike attacks. However, clock-signal glitches are the the simplest and most practical attacks according to [KK99] and [BCN+04]. Interestingly, clock glitch attacks did not emerge in the scientific community but in the pay-TV hacker community (cf. [AK97]).

Details about glitches can be found in [AK96], [AK97], [KK99], and [BS03]. Glitches are cited together with spikes and radiation as the standard methods for inducing faults by various authors, e.g., [NM96], [BS97], [KR97], [BMM00], [Wei00], [BS03], [PQ03], [YMH03], [BCN+04], [KKT04], [PV04].

Light Attacks / Optical Attacks. If a smartcard is unpacked, such that the silicon layer is visible, it is possible to use a laser cutter (red or green laser) or focused UV light in order to destroy individual structures of the chip [KK99]. This allows to induce a great variety of destructive faults.

Moreover, non-destructive attacks are possible as well. Memory cells used for EEPROM memory and semiconductor transistors have been found to be sensitive to coherent light, i.e.,

lasers, in the same way as to ionizing radiation such as cosmic rays. This is due to photoelectric effects and already works for white light. By using lasers or focused ultraviolet light, EEPROM bits can be erased, i.e., set to 0. This happens if the photon energy of the applied kind of light exceeds the semiconductor band gap. Modern green or red lasers can be focused on relatively small regions of a chip, such that faults can be targeted fairly well. However, according to [BS03], such "light attacks" are usually not usable for a systematic attack, since they cannot be focused with a sufficiently high precision to change selected bits. Additionally, according to [SA02], researchers have only studied the effects of light in transistors, but never tried to create specific faults on purpose.

In 2002, however, a new attack using focused flash light was presented in [SA02]. Here, the authors show that it is possible to focus the flash of an ordinary camera flash light using a microscope and aluminum foil. This attack has been named "optical attack" by the authors, presumably because they use cheap optical equipment for their attack. This attack allows them to set or unset individual chosen bits of an SRAM memory cell. The attack requires that the attacked chip has been unpacked to allow visual contact with the memory cells. This yields a very strong attack since it allows to target single chosen bits in memory.

Ultraviolet light has been suggested already in the earliest papers concerning fault induction, e.g., [AK96] and [BS97]. The optical attack from [SA02] has been referred to by several authors since its presentation, e.g., [BS03], [PQ03], [BCN+04], [KKT04].

External Electrical Field Transients / Eddy Currents. Changes in the external electrical field have been considered as a possible method for inducing faults into smartcards for a long time, e.g., in [Koc96a], [AK97], [Pet97], and [KK99]. Here, faults are sought to be induced by placing the device in an electromagnetic field, which may influence the transistors and memory cells. Some of these approaches require that a probe has contact to the metallic surface of a chip. However, the main problem using such an approach is to target specific bits or variables stored on the card.

In 2002, a different approach was presented in [QS02]: given a coil, which is placed near a conducting surface, a magnetic field can be created if the coil is subject to an alternating current. This magnetic field induces eddy currents on the surface of the near conducting material. Eddy current are also known as "Courant de Foucault" according to [QS02]. The use of eddy currents to induce faults has been motivated by electromagnetic analysis of smartcards, which has been proposed as a passive side-channel for these devices in, e.g., [QS01], [RR01], [GMO01].

Eddy currents are present in everyday life, where they are used for a variety of applications, ranging from inductive cookers to braking fast trains. In science, they are used for two additional applications. First, eddy currents can be used very well for measurements, since they are modified by several factors of the investigated surface, including conductivity, permeability and the geometry and distance from an object under investigation. Hence, eddy currents can be used to measure cracks in a surface, as well as electromagnetic emissions. If the passive probing needle is subject to an alternating current, the resulting eddy currents can be strong enough to interfere with the operation of a transistor or memory block. The property exploited for fault attacks is the fact that eddy currents can modify the number of electrons inside a transistor's oxide grid (cf. [Koc96a]). This changes the threshold voltage of the transistor, such that it cannot be switched anymore. Depending on the actual transistor, this can be used to ensure that a memory cell contains the value 0 or 1. This effect can be used to induce transient, permanent, and even destructive faults.

Second, eddy currents can also be used to heat a material in a uniform way, possibly until it

is melted (e.g., silicon bars). Hence, it can also be used to induce heat in a transient, permanent or destructive way to a smartcard. This may induce faults as explained above.

It has been shown in [QS02] that eddy currents can be used to induce faults very precisely, such that individual chosen bits can be set or reset. Heating the attacked device increases its susceptibility for such an attack. Eddy currents represent a very strong attack, similar to the optical attack presented in [SA02]. Inducing eddy currents does not require to unpack a chip, hence, attacks can easily be conducted. However, this requires that an adversary knows the layout of an attacked device in order to control the targeted precision.

Details on the attack can be found in [QS02]. It has been considered as a practical scenario in [KKT04].

Focused Ion Beams. Ion beams are used frequently in the reverse engineering of smartcards. They consist of a particle gun, shooting, e.g., Gallium ions from a liquid metal cathode, and a microscope, which focuses the beam. Ion Beams can be used to drill holes in the passivation layer of a smartcard, which can then be filled by a conducting material in order to access individual elements or bus lines with measuring equipment. This requires that the card is unpacked first. However, an ion beam can also be tuned finely enough to ionize silicon locally, which may be interpreted as a signal by the circuit. Additionally, destructive faults are possible by destroying circuit elements or bus lines.

Details on both laser cutters and ion beams as a source for faults can be found in [KK99]. Both have been considered as a method to induce faults by several authors, e.g., [AK96], [AK97], [Wei00], [BCN+04].

Other Sources for Attacks. Several authors have speculated about the possible use of additional sources for inducing faults, such as microwave radiation, static electricity, or ultrasonic vibrations of a microprobe, e.g., [AK96], [BDH+98], [BDL01]. We are not aware of actual experiments confirming these approaches. In [KR97], the authors speculate about software errors, which might be used to induce faults into parameters, e.g., in the case of overflowed arrays or exceeded boundary conditions. However, the applicability of these ideas have not yet been demonstrated in practice, either.

Remark. Evidently, error rates for SRAM cells used for cache memory are "orders of magnitude higher" than error rates for DRAM, which is used for long term EEPROM memory, according to [GA03]. Hence, transient attacks on intermediate variables as used throughout this dissertation are even more likely than attacks on DRAM cells.

1.1.1. Countermeasures

Smartcard manufacturers have been aware of the danger of faults for some time now, hence, they have developed a large variety of hardware countermeasures. These countermeasures are usually specifically constructed for different means of physical attacks. One major group are sensors and filters, which aim to detect attacks, e.g., using anomalous frequency detectors, anomalous voltage detectors, or light detectors. Other countermeasures are to use redundancy, i.e., dual-rail logic, where memory is doubled, doubled hardware, capable of computing a result twice in parallel, or doubled computations, where a computation is performed twice on the same hardware. If two results are computed, they are considered to be error-free if both values match. This is a very expensive countermeasure, and hence, it is not standard. Other standard

countermeasures are the use of a randomized clock to achieve an unstable internal frequency, bus line and memory encryption, dummy random cycles, and active and passive shields protecting the internal circuits. Hardware countermeasures are beyond the scope of this thesis, and will not be discussed in detail. For a more comprehensive overview, we refer the reader to [NM96], [Wei00], [BCN+04], and [RE00].

Using only hardware countermeasures has a great disadvantage. Highly reliable countermeasures are very expensive and most moderately priced countermeasures only detect specific attacks. Since new fault attacks are being developed frequently these days, detecting currently known forms of physical tampering will most probably not be sufficient against future developments. Although preventing an error is always the best countermeasure, this cannot be guaranteed by most hardware countermeasures. Therefore, software countermeasures are needed. Moreover, a software countermeasure "is superior, because it is more cost efficient and easier to deploy" [JBF02].

Current approaches for software countermeasures include checksums, randomization, masking, variable redundancy, and counters and baits (cf. [BCN+04]). Additionally, personalization and watermarking is used to bound the effect of a break or to prosecute attackers. However, the development of software countermeasures has not been very successful yet. Therefore, one goal of this thesis is to develop new software countermeasures. We will present such countermeasures both for repeated squaring (in Chapter 3), used in the context of RSA, and for repeated doubling (in Chapter 6), used in the context of elliptic curve cryptography.

1.2. Characterizing Fault Models

Fault attacks are based on tampering with a device in a way such that the device performs abnormally. From the reaction of the device, which may be a faulty result, an error message, or some form of security reset (including a destruction of the device), an adversary wishes to learn something about the secrets hidden in a device. As cryptographic algorithms need to be public in order to allow users to put trust in it — there is no such thing as security by obscurity — an adversary can determine what variables are used and what values they have depending on the secret key. This allows to determine what kind of error will provoke a certain reaction, which may be observable by the adversary. For example, if a single bit in the secret key is flipped during an attack, and the device does not detect this fault, a faulty result with a specific pattern is returned. By comparing this faulty result with the correct one, an adversary might be able to deduce one bit of the secret key. An adversary may also target the flow of operations, such that certain operations are repeated or skipped. To achieve and exploit a desired effect, he needs to have knowledge about how a certain physical attack will affect the logical flow of the attacked algorithm. Only then will an adversary be able to bound his success probability and to compute secret data from a faulty output.

There has been a large number of different fault attacks in the literature. They differ in the power to locate and time the induced faults, in the number of bits affected, in the effect of an attack (referred to as the fault type in this thesis), in the probability of the implied effect of an induced fault, in the duration of an effect, and in prior work that has to be applied to the card in order to use a specific physical setup. However, the characterization of the fault models has mostly been simple and insufficient to derive usable frameworks for a satisfactory analysis. Therefore, we present a characterization of the different parameters needed to fully describe all known fault models. This approach captures all previously described fault attacks.

Moreover, the approach leads to a proper mathematical formulation of the errors resulting from such attacks. Note that our characterization extends ideas from [YJ00], [YKLM01a], and [BS03], who already feature main aspects of our characterization.

- For **control on the fault location** we define the three classes "no control", "loose control" (a selected variable can be targeted), and "complete control" (selected bits can be targeted).
- For **control on the timing** we also define the three classes "no control", "loose control" (a fault is induced in a block of a few operations), and "precise control" (the exact time can be met).
- For **the number of bits** affected, we differentiate between a "single faulty bit", "few faulty bits" (e.g., a byte), and a "random number of faulty bits" (bounded by the length of the affected variable).
- The **fault type** describes the character of the fault as it manifests itself in the chip. This parameter has appeared in the literature as the "fault model". We break with this tradition as a reasonable description of a fault model must contain more than just the type of the fault. Fault types describe the effect of a fault on each individual bit. Section 1.3 will discuss fault types in detail.
- Attacks also have a **certain probability** associated with them. Usually an attack is not guaranteed to be successful, it is only so most of the time. Therefore, any effect as well as the control on location and timing might require a probability or even a distribution to be completely described. For example, some physical attacks might have a greater probability of resetting a bit than of setting that bit (see [Koc96a], [BDL01], [BS03]). No control on the location usually implies that a specific location is expected to be hit with a certain probability 1/(number of locations). Here, a uniform distribution is assumed.
- Depending on the type of physical stress applied, faults may show effects of different **duration**. We differentiate between *transient* faults, *permanent* faults and *destructive* faults.

  A *destructive fault* occurs if an adversary destroys a physical structure on the chip, which causes a certain bit or variable to be fixed at a specific value for all successive runs of the device. Destructive faults cannot be reversed.

  Permanent and transient faults are both faults which do not modify the hardware of an attacked device, thus allowing the device to recover from the induced faults after a certain period of time.

  Permanent faults change an affected variable until that variable is explicitly overwritten, e.g., by a reset at the start of the next run.

  Transient faults are faults where the induced fault is only short-lived, such that after a given amount of time, the effect ceases to exist and the correct value is present again. It is generally assumed that during the decay of a transient fault, there are no intermediate states, i.e., there is only a unique faulty value and a correct value. Concerning the duration of a transient fault, it is general consensus in the literature, that a transient fault only affects the next request for the affected variable. All further requests yield the correct value again. Although longer transient faults also yield usable results in analyses, such

faults are not considered by any author. Therefore, we will also assume that a transient error only affects a single use of a variable.

Permanent and transient faults have the same effect if a faulty variable is referenced only once. If the variable is used more often, they are fundamentally different. In the case of permanent faults all successive requests yield the faulty value. In the case of transient errors only the immediate next request yields the faulty value, subsequent requests yield the initial correct value.

As a summary, we present the characterization of the parameters demanded to fully describe a fault model in Table 1.1.

Parameter	Possible Values
location	no control, loose control, or complete control
timing	no control, loose control, or precise control
number of bits	single faulty bit, few faulty bits, or a random number of faulty bits
fault type	stuck-at fault, bit flip fault, random fault, or bit set or reset fault (to be defined in Section 1.3)
probability	(various possible values)
duration	destructive, permanent, or transient faults

Table 1.1.: Parameters and their possible values characterizing fault models

Since this thesis concentrates on algorithmic aspects of fault attacks, we do not elaborate deeply on the actual physical work an attacker has to do prior to an attack. This depends on the actual physical attack the adversary is using. Section 1.1 has summarized several methods to induce faults into devices. Some of these methods require that the chip is unpacked, some require contact to the metal surface, others need at least visual contact to be able to focus light on certain parts of the chip.

1.3. Definition of Fault Types

The characterization of fault models used in most publications about fault attacks merely concentrates on the fault types. However, this characterization falls short of giving enough information to judge about the feasibility of an attack. Therefore, we have presented a new approach in the last section, which captures all information needed from today's point of view. In the next section, we will give examples of the most common fault models. Prior to that, we want to describe all existing fault types in greater detail. We do this in order to stress the difference between fault types and fault models.

Fault types describe the effect of a fault on an arbitrary set of bits, while fault models also capture what set of bits belonging to which variable will be affected. Up to now, no publication implied that different fault types can be used in one single attack. Therefore, we will always assume that any fault model relies on only one specific fault type. In addition to that, no researcher has reported dependent bit faults before, i.e., any fault type can be described by the effect on individual bits. This does not prevent bits to be correlated to each other due to an explicit setting of the adversary, e.g., using the bit set or reset fault type described below.

However, no report suggests that it is possible that a fault modifying one bit automatically changes other bits depending on the new value of the first bit.

Before defining fault types, we introduce some notation.

Definition 1.1 (Error Notation).
Let x be a variable modified by an induced fault. We denote by $\overset{\not\,}{\longmapsto}$ the mapping of x to a faulty value $\tilde{x}$ according to some given fault type or fault model. Since faults change an affected variable x to some other value, we can always write a faulty value $\tilde{x}$ as the sum of the original value plus the error term, i.e.,

$$x \overset{\not\,}{\longmapsto} \tilde{x} = x + e(x),$$

where $e(x)$ is the absolute error. Frequently, we will write e instead of $e(x)$ for simplicity. If a fault may yield several values for $e(x)$, the error term is usually regarded as a random variable, which may take any specific value with a certain probability defined by the fault model.

Definition 1.2 (Fault Type: Stuck-At Fault (saf type)).
Let $B = \{b_0, b_1, \ldots, b_n\}$ be an arbitrary set of bits stored in memory. Assume that the bits of B are modified by a stuck-at fault*. Then the bits of B are fixed to their values*

$$b_i \overset{\not\,}{\longmapsto} b'_i = b_i \qquad \forall 0 \leq i \leq n.$$

The values of the affected bits are not changed any more, even if a variable x, which uses these bits, is overwritten. The effect is permanent, but not necessarily destructive, i.e., a complete reset of a device might be able to release all faulty bits again. The value of any b_i is not known to the adversary.

For stuck-at faults, note that the fault will have a noticeable effect only if the variable is overwritten at some point. In practice, stuck-at faults are usually considered to be destructive faults, where it is assumed that a destroyed wire, gate or memory cell will cause the faulty bit to be stuck at the value 0 (cf. [BS97]). Obviously, a transient stuck-at fault does not make any sense, since such a fault does not yield a faulty behaviour.

Definition 1.3 (Fault Type: Bit Flip Fault (bf type)).
Let $B = \{b_0, b_1, \ldots, b_n\}$ be an arbitrary set of bits stored in memory. Assume that the bits of B are modified by a bit flip fault*. Then all bits of B are set to their complementary values*

$$b_i \overset{\not\,}{\longmapsto} b'_i = 1 - b_i \qquad \forall 0 \leq i \leq n.$$

The effect may be transient, permanent, or destructive. The value of any b'_i is not known to the adversary unless he knows b_i already.

A bit flip fault is a very popular fault type, due to two reasons. First, the flipping of a single bit value, stored in a single memory cell, is easy to imagine, and it can be motivated by physical properties of memory cells. Second, this fault type guarantees that a variable using the bits of the set B has a faulty value, which is different from its original value.

Definition 1.4 (Fault Type: Random Fault (rf type)).
Let $B = \{b_0, b_1, \ldots, b_n\}$ be an arbitrary set of bits stored in memory. Assume that B is modified by a random fault*. Then the bits of B are set to random values*

$$b_i \overset{\not\,}{\longmapsto} b'_i \in_R \{0, 1\} \qquad \forall 0 \leq i \leq n.$$

The random choices of all b'_i are assumed to be independent. Unless otherwise noted, the uniform distribution is assumed. The effect may be transient, permanent, or destructive. Again, the value of any b'_i is not known to the adversary.

A random fault is usually considered to be the most realistic fault type. Since several physical methods of fault induction are hard to control precisely, non-destructive physical stress is most likely to change a variable according to the random fault type. Beyond that, some authors have even called stuck-at faults and bit flip faults controversial (e.g., [YKLM01a]).

Usually, a random fault is assumed to create a bit value $b'_i = 0$ with probability 1/2 and $b'_i = 1$ with probability 1/2. However, some authors have also considered an asymmetric distribution, where it is much more likely that a bit is reset to 0 rather than set to 1. This has been used for attacks on unknown symmetric cryptosystems in [BS97], and it has been mentioned, but not used, in [BDL97]. Such asymmetric distributions are usually motivated by specific physical attacks. It could be reasoned that faults, which always (or almost always) cause bits to be set to 0 should be considered as an additional fault type. However, this has not been done before in the literature and such faults have been considered only once, in [BS97]. Hence, we stick to tradition and do not define this as a new fault type.

Similarities and Differences Between Fault Types. The above three fault types, (saf), (bf) and (rf) fault types, have already been described in the earliest publications concerning faults, [AK96] and [AK97], and they have been used in many publications since then.

Although the three fault types are fundamentally different, these differences are usually not addressed in publications. The reason for this is the fact that in practical attacks, it is sufficient that a certain effect occurs at some time. For example, an attack assuming the bit flip fault type will usually also be successful if the random fault type is used instead.

The reason that most of the times, one fault type can be replaced by another fault type, is that attacks usually only require that a specific effect occurs at some time. If an adversary can somehow detect if a desired effect occurs or not, he does not have to use a specific fault type. It is usually acceptable if an adversary has to induce faults multiple times during an attack until a desired effect occurs. For example, both stuck-at faults and bit flip faults can be interpreted as random faults with extreme distributions, which depend on the actual value of any individual bit. Hence, the effects of stuck-at faults or bit flip faults can also be achieved given the random fault type and enough attempts. On the other hand, a uniformly distributed random fault can be seen as a bit flip fault with a success probability of only 1/2 for every affected bit.

Given the random fault type, all bits from the targeted set B might be changed to the values they already had. If the duration of the fault is transient, the attack will not be noticeable. This also holds for individual bits, e.g., given a bit b_i, no error will be detected in bit b_i, if the fault results in $b'_i = b_i$. However, if $b'_i \neq b_i$, the effect is the same as for the bit flip fault type. Hence, it is possible to interpret a fault affecting a set B of n bits using the random fault type as a fault affecting only a subset $B' \subseteq B$ using the bit flip fault type. Here, B' contains all those bits from B, whose values have been changed.

For simplicity, most authors assume only bit flips if the attack assumes single faulty bits. For multiple faulty bits, the random fault type is usually preferred. To our knowledge, stuck-at faults have only been used in [BS97] as destructive faults. We are also not aware of any attack which explicitly uses the fact that a bit flip fault is guaranteed to change a fixed set of multiple bits to their complementary value. Most attacks described in the open literature, which rely on classical fault types, work for any of these three types.

The next fault type is based on experiments described in [SA02] and [QS02]. They have been described in detail in Section 1.1. The name of this fault type is due to [BS03].

Definition 1.5 (Fault Type: Bit Set Or Reset Fault (bsr type)).
Let $B = \{b_0, b_1, \ldots, b_n\}$ be an arbitrary set of bits stored in memory. Assume that B is modified by a bit set or reset fault. *Then the bits of B are set to chosen values*

$$b_i \overset{\not\leftarrow}{\mapsto} b_i' = c_i, \quad c_i \in \{0,1\} \qquad \forall 0 \leq i \leq n.$$

The important fact about this fault type is that the values c_i are known and are usually chosen by the adversary. The effect may be transient, permanent, or destructive. The original values of the bits of B are not known to the adversary.

Definition 1.5 shows that the effect of the random fault type can be realized using the bit set or reset fault type, but the effects of both the stuck-at fault type and bit flip fault type can only be realized if the original values of the bits are known. However, since the adversary can choose the values c_i freely, he can choose them uniformly i.i.d. (independent identically distributed) at random and can thus achieve the effect of a stuck-at fault or a bit flip fault with a probability of $1/2^{n+1}$.

The bit set or reset fault type, however, is fundamentally different from the other three fault types, since an adversary can set any specific bit to any chosen value. This represents an extremely powerful adversary. It is the strongest fault type known and modern smartcards take several measures to reduce the power of such an adversary. Taking the properties of modern smartcards into account, this fault type may be assumed as almost impossible to achieve in practice.

1.4. Definition of Fault Models

Since characterizing fault attacks requires more than just the specification of the fault type, we will now describe a variety of fault models, which yield complete descriptions of fault attack settings. We will present the most popular fault models, which have been used extensively in the literature, and we will derive new justifications for known fault models by carefully examining the properties of modern high-end smartcards and the power of an up-to-date adversary. As we have explained before, fault models need to be justified by possible real world attacks in order to be considered as valid. Therefore, we will always present both a motivation from the real world for our fault models, as well as a brief overview over the literature, showing where these fault models have been used.

Modern Smartcards. To derive reasonable fault models, which can be justified by physical attacks, we combine parameter settings known from actual attacks with hardware countermeasures in effect on the card. Since this thesis approaches fault attacks from an algorithmic point of view, we do not go into depth when describing the physical properties and the details on chip card and integrated circuit design, neither do we describe hardware countermeasures in detail. We refer the interested reader to [RE00] for details. As a reasonable sketch for the architecture of modern high-end smartcards, we present Figure 1.2.
Since we are guided by practical considerations, we always assume the most powerful adversary, i.e., the adversary presented and justified by experiments in [SA02] and [QS02]. This adversary is able to use the bit set and reset fault type defined in Definition 1.5, which yields an extremely

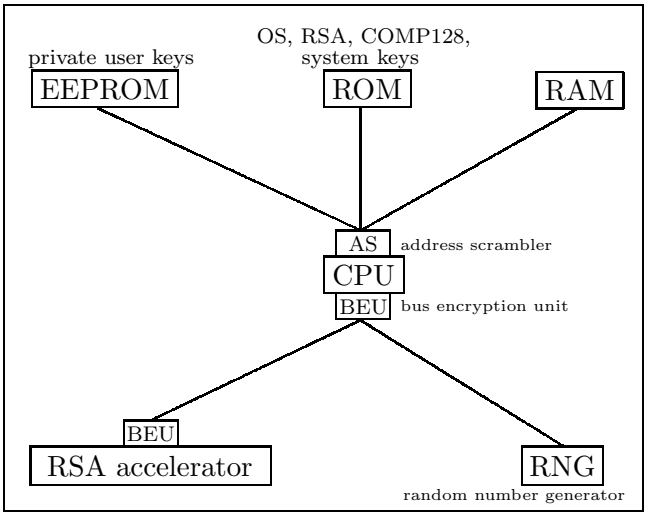


Figure 1.2.: Architectural sketch of a modern high-end smartcard.

strong adversary. Such an adversary can target any specific area on the card at a precise time (on *his* clock) and set the stored bit to any specific value. We also consider smartcards with several hardware countermeasures in effect (see [RE00], [KK99]). In these realistic scenarios, the effective power of the adversary is reduced significantly, e.g., knowledge about the location of the induced fault does not need to imply knowledge about the position of the bit within an algorithm. Similarly, precise timing not necessarily implies knowledge of the actual step performed at that time. There are three most important hardware countermeasures used on modern smartcards. First, a randomized clock is used to blur the timing behaviour of a device. Second, memory and data encryption is incorporated to ensure that a large number of bits is affected if a single bit is flipped by a fault. Third, address and bus line scrambling is employed such that the physical layout of the memory cells does not reflect the logical layout. These countermeasures degrade the power of the strongest adversary known today. In the discussion of the fault models defined below, we will give examples of how the adversary is weakened. In general, the stronger the countermeasures of a smartcard, the weaker the attack and the more bits are affected by an attack. The following fault models can be motivated by gradually improving the strength of the card's countermeasures.

The resulting fault models describe faults induced into a single bit, into a bounded number of bits, e.g., a byte, and into all bits of a given variable. Some fault attacks just require the induction of an unspecified fault, such as the attack on CRT-RSA described in Chapter 3. And last, some attacks only require the knowledge whether a fault occurred or not.

Throughout this thesis, we will always assume that an adversary can only induce a single fault during each run of an attacked algorithm. This implies that only one variable may be targeted. This is an important restriction, used throughout the literature. It ensures that an adversary cannot induce faults into both, a certain variable and a checking procedure. Otherwise, he could try to make sure that internal checks are circumvented. Although some authors suggest that no checking procedure should represent a single point of failure (cf. [AK96], [YKLM01b]), there have been no reports anywhere suggesting that such correlated attacks are possible with a reasonable success probability. Only once, multiple, yet uncorrelated faults have been considered (in [BS97]). However, this has not led to a new type of attacks. Besides that, it should be clear

that during a fault attack, an adversary may run the attacked device multiple times, and may induce faults during each execution.

We also assume that an adversary cannot change the fault model during an attack. This restriction is motivated by the assumption that an adversary uses only a single means of physical attack to induce faults. This is a reasonable assumption, as otherwise, we could just define a new fault model, capable of catching all aspects of a given attack with multiple different effects. However, such an attack has not yet been described anywhere in the literature.

1.4.1. Fault Models Affecting a Single Bit

In this section, we describe fault models, where faults affect only a single bit of a targeted variable. Such scenarios assume a very strong adversary and it has been claimed by some authors that such fault models are unrealistic in practice (cf. [YKLM01a], [Dot02]). Since all trustworthy smartcards are fully armed with a variety of countermeasures, precise control of the location of an affected bit or byte seems to be unrealistic these days. However, fault models assuming a single faulty bit, usually based on the bit flip fault type, are very popular fault models, which are widely used for fault attacks.

Chosen Bit Fault Model

We first describe the strongest adversary, which can be justified by experiments today. Here, the bit set or reset fault type from Definition 1.5 is assumed for an adversary, who attacks an unprotected card.

Definition 1.6 (Fault Model: Chosen Bit Fault Model).
We define the Chosen Bit Fault Model *with the following parameters.*

location:	*complete control (a specific bit of a specific variable used in a specific line of code of an attacked algorithm can be targeted)*
timing:	*precise control (can target a specific point of time, if the affected variable is used in a loop, a specific iteration can be targeted)*
number of bits:	*one bit of the affected variable is changed*
fault type:	*bit set or reset faults (as defined in Definition 1.5)*
probability:	*certain (every physical attack results in a fault of the desired kind, but if the adversary sets the targeted bit to its original value, the correct result will be returned)*
duration:	*permanent faults and transient faults (both variants are investigated if they differ, however, it is assumed that an adversary cannot use both kinds during one fault attack)*

Mathematical model: *The effect of an attack using this fault model on a targeted variable can be modeled as an addition or subtraction of a single bit, i.e., a variable X is changed to*

$$X \overset{\not\downarrow}{\mapsto} \tilde{X} = X \pm 2^k \qquad \text{for } 0 \leq k \leq l(X) - 1.$$

Both k and the sign of $\pm 2^k$ are assumed to be chosen by the adversary.

Motivation from the Real World. The Chosen Bit Fault Model resembles the (bsr) fault type that is achieved by physical attacks described in [SA02] or [QS02]. It can be achieved by inducing

faults into RAM or EEPROM of an unprotected smartcard. Although high-end smartcards implement sophisticated hardware countermeasures, many smartcards currently used are either too old or too cheap to do so. Hence, this fault model is a realistic one for such cards. It assumes the strongest adversary and the weakest card.

Although this model is unrealistic for modern high-end smartcards, it is a particularly interesting model, since algorithms secure in this fault model are secure in the weaker models as well.

Motivation from the Literature. The Chosen Bit Fault Model has been considered in [Dot02] for attacks on FLASH, SFLASH and QUARTZ, but has not been worked out since the model has been considered "not to be very realistic". It has also been considered briefly in [BS97]. We have proposed this fault model as "Fault Model #1" in [BOS03].

Single Bit Fault Model

We now describe the most important and mostly used fault model, the fault model introduced by Boneh, DeMillo, and Lipton in the first paper about fault attacks, [BDL97].

Definition 1.7 (Fault Model: Single Bit Fault Model).
We define the Single Bit Fault Model *(or BDL Fault Model) with the following parameters.*

location:	*loose control (a specific variable used in a specific line of code of an attacked algorithm can be targeted, but no specific bit of the variable can be targeted)*
timing:	*no control (cannot target a specific point of time, if the affected variable is used in a loop, every iteration of that loop is hit with the same (uniform) probability)*
number of bits:	*one bit of the affected variable is flipped (every bit of the variable is hit with the same (uniform) probability)*
fault type:	*bit flip faults (as defined in Definition 1.3)*
probability:	*certain (every physical attack results in a fault of the desired kind)*
duration:	*permanent faults and transient faults (both variants are investigated if they differ, however, it is assumed that an adversary cannot use both kinds during one fault attack)*

Mathematical model: *The effect of an attack using this fault model on a targeted variable can be modeled as an addition or subtraction of a single bit, i.e., a variable X is changed to*

$$X \not\mapsto \tilde{X} = X \pm 2^k \qquad \text{for } 0 \leq k \leq l(X) - 1.$$

The bit position k is assumed to be chosen according to the uniform distribution and subsequent choices are assumed to be i.i.d..

Fault Model 1.7 could be further relaxed, to use random faults instead of bit flip faults as the fault type. Since only a single bit is affected, a uniformly distributed random fault has the same effect as a bit flip fault with success probability 1/2. Although being more realistic, this relaxation does not yield new attacks. Using the bit flip fault type simply means to neglect all faults, which change a bit to the original value. This is a reasonable approach since such faults do not yield a faulty result. Therefore, it is not necessary to assume random faults. Originally,

it is noted in [BDL97] that the authors assume a physical setup, where the induced faults are more likely to flip bits from 1 to 0, rather than from 0 to 1. However, they do not use such an asymmetric behavior and do not consider it in their fault model. Therefore, we neglect this property as well and assume a uniform distribution of bit flips.

The mathematical model of Fault Model 1.7 also neglects the fact, that bit flips can only flip the current value of a bit in reality, i.e., only one of the two possible error terms $+2^k$ or -2^k is actually possible. However, the chosen model simplifies notation and analysis. Moreover, it is the standard model used in the literature. Therefore, we use this mathematical model as well.

Motivation from the Real World. The Single Bit Fault Model is unrealistic for modern high-end smartcards, which are armed with a variety of countermeasures. Such a protection prevents an adversary from realizing the bit set or reset fault type. However, there are cards, which are too old or too cheap to incorporate sufficient countermeasures. If a card at least uses address scrambling for every single variable or bus line scrambling, the adversary's power is degraded to Fault Model 1.7. The main motivation for this model, however, comes from literature. Here, it is very popular, since it is an easy model, which allows to describe fault attacks in a simplified setting, which can then be generalized to a larger number of affected bits.

Motivation from the Literature. Fault Model 1.7 is one of the oldest and most prominent fault models. It has been used explicitly in [BDL97] and [BDL01] for attacks on plain RSA. It has also been used for fault attacks among others in [BS97] (where it is also considered in a relaxed version, where the targeted variable is unknown), and in [JQ97a], [JQ97b], [Pet97], [BDH+98], [BMM00], [YKLM01b], [Dot02], [JQYY02], [JBF02], [YKLM03], [GA03], [CJ03], and [BCN+04].

1.4.2. Fault Models Affecting a Bounded Number of Bits

Another common model assumes that only a bounded number of bits ("multiple bits", "few faulty bits") are affected by an induced fault. This model catches the fact that mobile devices store and load variables in blocks, where the length of a block is the word size of a device. The word size is usually one byte, hence, attacks which assume that a byte is affected by induced faults are considered to be the most realistic fault attacks in several papers (e.g., [PQ03], [GK04]). Hence, if the standard block size on smartcards increases, the fault model defined below should be adapted to that change. A bounded number of bits may also be affected by faults, which target a specific bit, but have such a strong effect on the memory that neighbored bits are changed as well. In this case, the fault is smeared over a bounded number of surrounding bits, where boundary bits are possibly changed with a lower probability than the target bit. However, it is more practical to assume a random change of the whole byte. Moreover, we are not aware of any attack, which exploits a probability distribution other than the uniform distribution.

The byte fault model represents all attacks, where an unspecified "small" number or "multiple" bits are affected by the induced faults. In all such cases, the number of attacked bits is assumed to be small enough such that all possible error patterns are efficiently sampleable. This allows an adversary, who does not know how the faulty bits have actually been changed, to try all possible patterns as a basis for a recovery of secret data. The goal of such an exhaustive enumeration is to identify the correct error pattern. An exhaustive search requires that the number of faulty bits can be bounded by a small number. Therefore, many authors, who assume "multiple" faulty

bits, assume some variant of the byte fault model, which may in some cases cover few more than just 8 bits.

Definition 1.8 (Fault Model: Byte Fault Model).
We define the Byte Fault Model *with the following parameters.*

location:	*loose control (a specific variable used in a specific line of code of an attacked algorithm can be targeted, but no specific block of 8 bits of the variable can be targeted)*
timing:	*no control (cannot target a specific point of time, if the affected variable is used in a loop, every iteration of that loop is hit with the same (uniform) probability)*
number of bits:	*few faulty bits, i.e., one byte (8 bits) of the affected variable is changed (every block of 8 bits of the variable is hit with the same (uniform) probability)*
fault type:	*random faults (as defined in Definition 1.4)*
probability:	*certain (every physical attack results in a fault of the desired kind, but with probability 1/256 all affected bits are set to their original value and the correct result is returned)*
duration:	*permanent faults and transient faults (both variants are investigated if they differ, however, it is assumed that an adversary cannot use both kinds in one attack setting)*

Mathematical Model: *The effect of an attack using this fault model on a targeted variable can be modeled as an addition or subtraction of an unknown error byte at an unknown position, i.e., an affected variable X is changed to*

$$X \overset{\not}{\mapsto} \tilde{X} = X \pm b \cdot 2^k \qquad \text{for an unknown } 0 \leq k \leq l(X) - 8 \text{ and an unknown } b \in_R \mathbb{Z}_{2^8}.$$

In this model, we assume that both b and k are chosen i.i.d. at random from the corresponding sets of possible values according to the uniform distribution.

Usually, a device will load and store parameters in fixed blocks of 8 bits, starting with the least or most significant byte. Therefore, in practice, not all sets of 8 contiguous bits may be equally likely to be affected by a byte fault. However, this aspect is usually neglected, as long as it is not important for the analysis which byte is faulty. It simplifies analysis at a negligible cost. Hence, we assume that any block of 8 contiguous bits is hit with the same uniform probability.

In the case of transient faults, one can simply consider all those bits to be faulty, whose values have been flipped by the random fault type. This allows to exchange the fault type in Fault Model 1.8 to the bit flip fault type. This interpretation is used by many authors, who simply assume "multiple" faulty bits.

A popular way to relax the Byte Fault Model is to allow that the affected bits are not contiguous, which is done by several authors, who assume "multiple" faulty bits. For a variable of length n, there are less than n^8 possible sets of 8 different bits, hence, from a complexity theoretic point of view, testing all these possible sets adds only a polynomial factor to the overall complexity. This remains true as long as the number of affected bits is a constant, which guarantees that the number of tests is small enough.

Motivation from the Real World. In the real world of modern high-end smartcards, this model is motivated in three variants.

The Byte Fault Model as defined above is motivated by an adversary, who is able to induce bit set or reset faults, and a smartcard, which uses block-wise memory encryption or bus line encryption and address scrambling on RAM and EEPROM (cf. [RE00]). Usually, all data stored in EEPROM and RAM is encrypted ([RE00]). Hence, if a fault is induced into memory, the value of the affected block seen by the CPU will be unpredictable. Therefore, encryption degrades the power of the adversary, since any single modified bit in memory affects a whole byte of data. Address scrambling prevents the attacker to specify when the faulty block is requested by the CPU. However, we assume that the memory layout allows an adversary to target a specific variable. The same model is derived if the bus lines are targeted. We have proposed this fault model before as "Fault Model #3" in [BOS03].

As a second variant of the Byte Fault Model, we may assume that the adversary knows which byte of the targeted variable is faulty. In the mathematical model, this can be expressed by a *known* parameter k. This model is motivated in the real world by the fact that the strong adversary's power is reduced on smartcards if only encryption of the data is used and no address scrambling. We have proposed this fault model as "Fault Model #2" in [BOS03].

As a third variant of Fault Model 1.8, a generalized version may be assumed, where the adversary knows that only one byte is changed, but he has no control on both timing and location. In this case, he might not know which variable was affected by his attack. In the mathematical model, this can be expressed by assuming that the variable X is randomly chosen from a set of possible variables. In the real world, this variant can be motivated by two scenarios. First, a smartcard could use block-wise memory encryption and address scrambling as above. However, we now assume that the memory layout does not allow an adversary to target a specific variable. Second, a smartcard may use memory encryption in RAM and EEPROM together with a randomized clock (cf. [CCD00]). In this case, the strong adversary's power is effectively reduced to this model. Memory encryption causes any single bit fault to affect a whole block of data. In this case, the attacker knows that a successfully induced fault will change a block of data. However, the randomized clock implies that he does not know the exact time of the change within the algorithm. If he targets intermediate variables in RAM, in an accelerator, in cache memory, or the bus lines, the attacker does not know the position of the block as it is used in the CPU. We have proposed this fault model as "Fault Model #3' " in [BOS03].

Motivation from the Literature. Variants of the byte fault model as defined above have been used in [BDL01] (where "a small number of bits flips" are assumed), in [JQ97b] (where "multiple" faulty bits and bit flips are assumed), in [BDH$^+$98] (where multiple faulty bits are assumed, whose positions are known), in [JQYY02], in [PQ03] (who also consider a relaxed model where the affected variable is not exactly known), in [GA03] (where "a few" bits are assumed to be faulty, e.g., 6), in [BCN$^+$04] (where multiple bits do not need to be contiguous), in [GK04] (who quote our "Fault Model #3" from [BOS03]), and in [Wag04] (where only $l(x)/8$ possible bytes in a variable x are considered). Additionally, it is mentioned in [YJ00], and in [YKLM01a] (where a "limited number of faulty bits" is referred to as more difficult to achieve than an arbitrary number of faulty bits). In [Wag04], the author also proposed an extreme version of a bounded fault model, where all but the 160 most significant bits of a 1024 bit RSA modulus are affected. However, this model has not been motivated by real world considerations.

1.4.3. Fault Models Affecting an Arbitrary Number of Bits

Random Fault Model

Sometimes, an adversary knows not enough about his induced faults to bound their effects sufficiently. While he knows the affected variable, he does not know how the change of a single bit affects the value of the variable. In this case, it is sometimes reasonable to assume that an affected variable is changed to some random value. As long as all values might occur with non-negligible probability, using the assumption that the resulting values are distributed uniformly is a reasonable simplification for analyses.

Definition 1.9 (Fault Model: Random Fault Model).
We define the Random Fault Model *with the following parameters.*

location:	*loose control (a specific intermediate variable used in an attacked Algorithm can be targeted)*
timing:	*no control (cannot target a specific point of time, every line of code / every iteration of a loop, is hit with the same (uniform) probability)*
number of bits:	*random number of bits, i.e., all bits of the targeted variable are affected, the value is replaced with a random value in situ, i.e., using the same number of bits, leading zeros allowed (all possible random values occur with the same (uniform) probability)*
fault type:	*random faults (as defined in Definition 1.4)*
probability:	*certain (every physical attack results in a fault of the desired kind, but the original value is allowed to appear as well)*
duration:	*transient and permanent faults (both variants are investigated if they differ, however, it is assumed that an adversary cannot use both kinds during one fault attack)*

Mathematical Model: *The effect of an attack using this fault model on a targeted variable can be modeled as an addition or subtraction of an unknown error from a bounded set. We assume that a targeted variable X is changed to some random value, i.e., $\tilde{X} \in [0, 2^l - 1]$. In this model, a fault may result in any faulty value. We have*

$$X \overset{\lightning}{\mapsto} \tilde{X} = X + e(X) \qquad \text{with } e(X) \in_R \{-X, -X+1, \ldots, 2^l - X - 1\}.$$

Here, l is the maximal length the variable X may take. If $X \in \mathbb{Z}_p$, we have $l = l(p-1) = \lfloor \log_2(p-1) \rfloor + 1$. Otherwise, l is the maximal length of the memory space reserved for X. We assume that the random variable $e(X)$ takes any value from the set $\{-X, -X+1, \ldots, 2^l - X - 1\}$ with the same (uniform) probability and that subsequent choices are i.i.d. according to the uniform distribution.

The Random Fault Model can be used if an adversary either knows that an induced fault will set the affected variable to a random value according to the uniform distribution, or if the success of his fault attack does not rely on special values to appear at some time or with a specific probability. The naming of this fault model should not be confused with the random fault type from Definition 1.4, however, the intuition behind both concepts is similar, both describe that a given set of bits is replaced by random bit values.

In practice, the adversary may be able to target a desired variable in a reasonable small interval of operations or even loop iterations. This interval may have been derived from other

sources of information, for example from the power profile of the card (see [ABF+02]). In this case, an adversary may gain some control on timing and bound the lines of codes or the loop iterations, in which the targeted variable has been hit.

As a more general variant of this fault model, one might also assume that even the affected variable is not known with certainty, e.g., an adversary might only be able to hit a variable from a set of a few different variables.

Motivation from the Real World. This scenario is motivated by strong high-end smartcards completely armed with countermeasures. Memory encryption, address scrambling and a randomized clock imply that any fault induced into memory or the CPU at a vague point will leave the attacker at most with the information that a certain variable is faulty. It therefore enforces a very weak adversary.

Motivation from the Literature. The Random Fault Model has been used for successful fault attacks in [YKLM01b], [Dot02] (for an attack on ESIGN-D), in [YKLM03], in [KKT04] (where also a different model was used which assumed that bits could only be set to 0), and in [PV04]. We have proposed this fault model as "Fault Model #4" in [BOS03].

Arbitrary Fault Model

We now describe the weakest fault model. Here, an adversary can only target a specific line of code or a set of several intermediate variables. He does not have any additional information, e.g., about the type of the fault or the error distribution. Attacks based on this weak adversary are extremely dangerous, because the weaker an adversary, the more practical the attack.

Definition 1.10 (Fault Model: Arbitrary Fault Model).
We define the Arbitrary Fault Model *with the following parameters.*

location:	*loose control (a specific line of code can be targeted, but no specific variable used in that line)*
timing:	*no control (cannot target a specific point of time, if the affected line of code is from a loop, every iteration of that loop is hit with the same (uniform) probability)*
number of bits:	*random number of faulty bits (any number of bits may be affected using an unknown probability distribution, however, at least one bit is changed)*
fault type:	*unknown (any fault type is possible, but the adversary does not know the type)*
probability:	*certain (every physical attack results in a fault of the desired kind, i.e., the final result computed by the affected line of code is different from the correct result)*
duration:	*transient and permanent faults (both variants are investigated if they differ, however, it is assumed that an adversary cannot use both kinds during one fault attack)*

Mathematical Model: *The effect of an attack using this fault model on a targeted line of code can be modeled as an addition or subtraction of an unknown error from a bounded set. Let X be the variable, which is assigned the result of the affected line of code. Then in the Arbitrary*

Fault Model, X is changed to the faulty value $\tilde{X}$, such that

$$X \not\mapsto \tilde{X} = X + e(X) \qquad \text{with } e(X) \in_R \{-X, -X+1, \ldots, 2^l - X - 1\} \setminus \{0\}.$$

Here, l is the maximal length the variable X may take. If $X \in \mathbb{Z}_p$, we have $l = l(p-1) = \lfloor \log_2(p-1) \rfloor + 1$. Otherwise, l is the maximal length of the memory space reserved for X. We emphasize the fact that the probability distribution of the random variable $e(X)$ is arbitrary and unknown.

In some situations, an adversary has extremely limited control over the induced faults. He may target a specific line of code, but it is unknown, which variable is affected and how that variable is changed. Since the Arbitrary Fault Model assumes very little information about the fault, all other fault models defined above can be used for attacks based on the Arbitrary Fault Model.

The Arbitrary Fault Model is not equivalent to the Random Fault Model 1.9. Basically, we have two differences. First, the Random Fault Model allows an adversary to assume that any specific error value appears with some known probability. This is not the case for the Arbitrary Fault Model, where the probability distribution is completely unknown. Second, an adversary using the Random Fault Model can target any specific variable, while the Arbitrary Fault Model only allows to target a specific line of code. This implies that in the Arbitrary Fault Model, transient faults on any variable or operation used in an affected line of code yield the same situation as if the result of the targeted line of code had been changed by the fault. For permanent faults however, the affected variable is not known with certainty. In this case, it is usually assumed that any variable used in the targeted line of code is hit with the same uniform probability. Hence, if an attack depends on a specific variable being affected, this situation will occur among a known (low) number of induced faults. This information is usually sufficient for fault attacks.

A fault attack, whose setting is described by the Arbitrary Fault Model, can be successful, if the adversary does not need to be able to guess the error term in order to derive information, and if he does not need assumptions about the distribution of the error value. He must be able to recover the secret data with any non-zero error value. Usually, an adversary cannot check all possible error values, since the set of possible bit patterns is not assumed to be efficiently sampleable in this model.

A generalized version may assume that an adversary cannot target a specific line, but will hit any line with a known probability, e.g., according to the uniform distribution. In this case, one out of several attacks can be expected to hit a desired line. This may be sufficient for success.

Motivation from the Real World. The Arbitrary Fault Model assumes the weakest adversary. In practice, the strong adversary's power of inducing bit set or reset faults may be reduced to this model in the presence of a large variety of countermeasures. Modern high-end smartcards feature a great variety of sensors and filters, hence, inducing faults is not an easy task. However, it is not impossible to do so. Still, the adversary may not be able to bound the effect of an induced fault sufficiently to assume other, stronger fault models.

Motivation from the Literature. This fault model has been used for fault attacks on CRT-RSA in [BDL97] and [BDL01]. This attack is discussed in detail in Chapter 3. Here, any fault, which changes the final result of a certain line of code, is sufficient to recover the secret key. This fault model has also been used in [JKQ97], [ABF+02], and [YMH03].

1.5. Validity and Applicability of the Proposed Fault Models

The five fault models presented in the last sections have been motivated by a real world scenario, comparing the power of the adversary with the power of the countermeasures present on a smartcard. This has been described in detail in the paragraphs titled "motivations from the real world". We always assume the strongest adversary known from open literature, i.e., an adversary capable of using the bit set and reset fault type from Definition 1.5. Depending on the strength of the smartcard's countermeasures, this power is weakened.

The Chosen Bit Fault Model is the strongest fault model possible. An adversary who is able to apply a Chosen Bit Fault Model can recover any bit of any variable used in an algorithm. This can be done by an oracle attack (which will be discussed in Section 1.6.2). However, this fault model is extremely unrealistic for modern smartcards, since smartcard manufacturers are aware of the danger of fault attacks. However, it might still be valid for old or cheap smartcards. Moreover, it is a particularly interesting model, since algorithms secure in this fault model are secure in the weaker models as well.

The Single Bit Fault Model assumes some protection of the smartcard, however, sensitive cryptographic applications will not be implemented on such cards anymore. However, due to its simple modeling, it is a very popular starting point to analyze the security of algorithms against fault attacks.

The Byte Fault Model represents a moderately strong adversary. It assumes a fairly well protected smartcard. Hence, it is much more realistic than the fault models assuming a single faulty bit, but it is not clear, to what extent smartcards are being manufactured, which allow this model. It is definitely unrealistic for modern high-end smartcards. Just like the previous two fault models, it may still be valid for old or cheap smartcards. Moreover, if the best protected cards should exhibit vulnerabilities in their line of defense, an adversary will most probable be able to base his attacks on this fault model. Therefore, it does have practical relevance.

The best protected smartcards degrade any powerful adversary known today to the Random Fault Model or even the Arbitrary Fault Model.

The Random Fault Model is one of the most realistic fault models to assume. Modern high-end smartcards will ensure that an adversary cannot hope to base successful attacks on any of the stronger fault models. Therefore, algorithmic countermeasures which aim at defending devices against attacks based on this fault model are needed.

The Arbitrary Fault Model is the most realistic fault model assuming the weakest adversary. It is a very practical model, since it can be realized by any physical attack, which can be mounted with a sufficiently high success probability. Attacks, which are successful in the Arbitrary Fault Model are also successful in any other fault model. However, the Arbitrary Fault Model is only realistic for a modern, high-end, and expensive smartcard.
We present a comprehensive summary of the defined fault models in Table 1.7.

A smartcard may react in various ways to an induced fault: Unprotected cards will fail to notice the fault and output a faulty result, leaving them vulnerable to a successful fault attack. Well protected cards may detect some or most induced faults by internal checking procedures and react with some error behaviour, e.g., with an error message or with a security reset. An error message may either depend on the actual error detected or may be unspecific. The optimal protected cards are equipped with a variety of additional hardware countermeasures (see [BS03]), which might detect some physical attacks even before the attack successfully induces an error. In this case, a complete security reset is triggered. Such countermeasures are ideal, since they prevent faults before they occur. Unfortunately, countermeasures which are capable

	Chosen Bit Fault Model	**Single Bit Fault Model**	**Byte Fault Model**	**Random Fault Model**	**Arbitrary Fault Model**
Definition	1.6	1.7	1.8	1.9	1.10
control on location	complete	loose	loose	loose	loose/no
control on timing	precise	no	no	no	no
number of affected bits	1	1	8	random	random
fault type	(bsr)	(bf)	(rf)	(rf)	unknown
success probability	certain in all models				
duration	all consider transient and permanent faults				

Table 1.7.: Overview of the fault models defined in Section 1.4

of preventing any possible physical fault induction are unrealistic, especially given the fact that new physical attacks are reported frequently. This emphasizes the need for algorithmic countermeasures and checking procedures, which do not depend on the physical attack, but only on the induced fault.

Therefore, an adversary can always hope to apply some physical attack, which cannot be prevented by the attacked card. All the fault models presented in Section 1.4 represent changes of the data used by a card. These fault models are used for actual attacks, where it is assumed that the faulty final result is returned by the smartcard. This faulty final result allows the adversary to compute secret data in whole or in parts, possibly requiring a correct result as well. Such attacks will be described for plain RSA (Chapter 2), CRT-RSA (Chapter 3), and elliptic curve cryptosystems (Chapter 5).

A natural approach to defend against these attacks is to incorporate checking procedures, which aim to prevent that faulty outputs are returned. Modern sophisticated smartcards may have both hardware and software countermeasures which alter the final result to some random value or use detection mechanisms that report an error to the user without revealing a faulty output. However, any such countermeasure will always leak the information that an induced fault successfully changed the internal data. Surprisingly, there are attacks, which are capable of recovering secret data only based on this information. We will refer to these attacks as *oracle attacks*, and describe this kind of fault attacks in Section 1.6.2.

If an adversary can recover secret data using an oracle attack, his attack is extremely powerful. An attacked device can only hide the information whether a fault occurred or not with a significant loss in efficiency. Assume that during a computation of a signature, a fault occurred, such that the final result is incorrect. If the device does not detect this, an adversary can simply compare the output with the correct result and gets the information that a fault occurred. If the device detects this error, it may either react with an error behaviour such as an error message or a security reset, or it may invoke the computation once again. Both reactions can be detected, either by looking at the output, or at the increased duration of the operation (timing attack). Therefore, even if a device implements a large variety of (efficient) countermeasures, such as

checking the final result before output, it may not be able to withhold the desired information from the attacker.

Hiding this information requires that the behaviour of a device does not change even if a fault occurred. This implies that both the duration of a computation as well as the final result must be independent from any error. Since we always assume that only a single fault can be induced, the device needs to compute the final result at least twice. This doubles the computation time, which is a significant loss in efficiency. Even doubled computations may not be enough to be secure against permanent faults.

However, as we will show in Section 1.6.2, most oracle attacks rely on extremely powerful fault models, hence, their applicability in practice is controversial. Although we will not consider this kind of fault attacks in depth in this thesis, we present in Chapter 7 an oracle attack on RSA based on a new fault type, which will be developed in Section 4.5.

1.6. Other Attack Scenarios

1.6.1. Code Change Attacks

All fault models presented in the previous sections assume that data is manipulated, such that variable values used in an algorithm can be pertubated. However, we have described the possibility that certain physical scenarios might induce faults in the program counter of a CPU or ALU, e.g., power spikes and clock glitches as presented in Section 1.1. This might cause certain operations to be skipped, yielding a *code skip attack*, it might cause certain operations to be executed multiple times, or it might cause a loop counter to decrease. The latter has been described theoretically in [BS97] and [PV04], whose authors suggest to induce faults into the loop counter to reduce the rounds in DES or pairing based cryptosystems to values, which allow to break the systems.

However, code attacks are not considered by many authors. In this thesis, we will only consider fault attacks on memory values, therefore, we will not describe code attacks in detail.

1.6.2. Oracle Attacks

As explained above, an oracle attack is a fault attack capable of recovering secret data based only on the fact whether an error occurred or not. Although such fault attacks are hard to realize in practice, we will demonstrate their power in this section. First, oracle attacks are formalized in the following definition.

Definition 1.11 (Oracle Attacks).
Any fault attack, which unveils secret information from a cryptographic device only based on (at most) a correct result and the information whether an induced fault changed the behaviour of a device, is called an oracle attack*. The induced faults may be defined according to any fault model.*

Typically, oracle attacks require a much stronger fault model than other types of fault attacks. In this section, we will briefly present the main flavors of such attacks. Oracle attacks usually exploit conditional statements. Examples for such attacks are attacks using an extremely powerful fault model (cf. [AK97]), and the more realistic *memory-safe errors* (cf. [YJ00], [JQYY02], [YKLM03]), and *computational-safe errors* (cf. [YKLM01a]). We will briefly introduce these attacks.

As a practical scenario to demonstrate the feasibility of oracle attacks, consider the following variant of the left-to-right repeated squaring algorithm in the context of RSA signatures, which uses dummy operations. We present this algorithm from [YKLM01a] as Algorithm 1.12. In Chapter 3, we also present a slightly different version of this algorithm as Algorithm 3.10.

Algorithm 1.12: Left-to-Right Square-And-Multiply-Always, Variant 1

Input: A message $m \in \mathbb{Z}_N$, a secret RSA key $0 \leq d$, where $l(d)$ denotes the binary length of d, i.e., the number of bits of d, and an RSA modulus N
Output: $m^d \bmod N$
\# **init**
1 Set $y := 1$
\# **main**
2 For k from $l(d) - 1$ downto 0 do
3 Set $y_0 := y^2 \bmod N$
4 Set $y_1 := y_0 \cdot m \bmod N$
5 If $d_k = 1$ then set $y := y_1$ else set $y := y_0$
6 **Output** y

This *square-and-multiply-always* algorithm always computes the value $y_1 = y_0 \cdot m \bmod N$, and sets $y = y_1$ only if $d_k = 1$. Introducing dummy operations helps to even out the power profile and to achieve a uniform timing behaviour.

Bit Set Or Reset Oracle Attack. If an adversary is able to induce faults according to the Chosen Bit Fault Model 1.6, an oracle attack is obviously possible. Assume that Algorithm 1.12 is attacked. If an adversary is able to set the bit d_k in Line 5 to an arbitrary but known value, say $d_k \overset{\lightning}{\mapsto} 1$, he can read out the correct value of d_k by comparing the actual result with the correct result. If both match, he changed the bit d_k to its correct value, i.e., we have $d_k = 1$. If the two results differ, the wrong branch was executed and we have $d_k = 0$. This scheme works for any variable: the adversary can recover a variable bit by bit, by targeting all bits of the variable one after the other. If the smartcard does not randomize the variables, an adversary may induce several faults in subsequent runs using the same input values and recover the exact value. However, this attack represents an extremely powerful adversary and is therefore not practical. Modern high-end smartcards do not admit of this fault model.

In the following, we describe two more realistic fault scenarios, memory-safe errors and computational-safe errors. Memory-safe errors aim at data, which is used or overwritten depending on secret data, computational-safe errors aim at instructions, whose result is used depending on secret data. Both require a powerful adversary, yet, not as powerful as represented by the Chosen Bit Fault Model.

Safe Error Oracle Attacks. In Algorithm 1.12, it can be seen that two values y_0 and y_1 are computed, but only one of them is used for the computation of y in Line 5. Hence, if a fault is induced into exactly one of the two values, it has no effect if the other one is chosen. The choice is determined by the secret key bit d_k. Such faults, which may have no effect on the computation depending on the secret key, have been named *safe errors* by Yen and Joye in [YJ00]. Such faults can be used for oracle attacks, if they can be targeted precisely enough.

Memory-Safe Errors. The authors of [YJ00] originally investigate the right-to-left repeated squaring algorithm in the context of RSA signatures (see Algorithm 2.3). However, the attack carries over to Algorithm 1.12 seamlessly. The basis for an oracle attack is to induce a fault during the computation of $y_1 = y_0 \cdot m \bmod N$ in Line 4 of Algorithm 1.12. The authors assume that the multiplication of $y_0 \cdot m$ is interleaved with the modular reduction. To compute $y_0 \cdot m \bmod N$, the variable y_0 is scanned from left to right. If a fault is induced into memory cells holding upper bits of y_0 after these bits have been used by the computation, this fault has no effect on the result y_1. Hence, the value y_1 is correct, while the value y_0 is faulty. If the secret key bit d_k is 1, we have $y = y_1$ and the faulty intermediate result y_0 is overwritten, when $y_0 = y^2 \bmod N$ is computed in the next round. This implies that the faulty memory cells are overwritten as well, and no error occurs. However, if $d_k = 0$, the faulty value y_0 is used in Line 5 and y is faulty. It is assumed that if any intermediate variable y is faulty, the final result of the algorithm will be faulty as well. This situation allows an oracle attack, since we have $d_k = 0$ if and only if the final result is faulty and $d_k = 1$ otherwise. The attack requires a strong adversary, who must be able to attack specific bits of y_0 during a specific line of computation in a specific loop iteration. The attack becomes easier if the desired bits of y_0 are stored in register cells, which can be targeted by an adversary without changing the lower bits. Imprecise control on location can be traded off for an increased number of attacks using statistics. Faults as described above have been named memory-safe errors, because they occur in memory cells, which are no longer used in a computation, e.g., as in Line 4 of Algorithm 1.12.
Memory-safe errors have also been described for oracle attacks in [JQYY02] and [YKLM03].

Computational-Safe Errors. The idea of memory-safe errors can be easily extended to interfere with the computation itself. While for memory-safe errors, some bits in memory are targeted, which are no longer used, computational-safe (or *c-safe*) faults aim at instructions, whose result is no longer needed, so-called *dummy instructions*. In [YKLM01a], Yen et. al. describe computational-safe errors.

The authors assume an attack on Algorithm 1.12. Here, the variable y_1 is targeted. If a fault is induced during the computation of y_1, the final result of Algorithm 1.12 is faulty if and only if we have $d_k = 1$ and it is correct if $d_k = 0$, since the value y_1 is not used in this case. While still representing a very strong adversary, the fault model assumed for computational-safe errors is not as strong as for memory-safe errors. C-safe errors can be induced if an adversary is able to target a specific instruction (during a specific iteration of a loop), however, since the effect of the induced fault in y_1 is not important, control on location and timing does not need to be as precise as for memory-safe errors, where the upper bits have to be targeted after being used. Basically, an adversary has more time to attack y_1 than to attack y_0 without changing y_1.

Contrary to the believe in [YKLM01a], the original (plain) repeated squaring algorithm can be attacked by computational-safe errors as well (see Algorithms 2.2 and 2.3). Assume that a fault is induced into the CPU of a device such that in the conditional statement, the execution of the then branch or the complete conditional statement is skipped. In this case, the final result will be wrong if $d_k = 1$ and correct if $d_k = 0$. Once again, the secret key bit can be recovered using the correct result, which is compared to the actual result to conclude whether the induced fault yielded a faulty output or not. This attack requires that an adversary is able to have instructions skipped by the device, which is a very strong assumption. However, several authors describe physical attacks, which potentially tamper with branch instructions, e.g., using power spikes or clock glitches as presented in Section 1.1.

Additionally, any attack, which induces transient faults in intermediate variables used in a

dummy statement can potentially be a source for an attack using computational-safe faults. In Algorithm 1.12, the value y_0 is used in Line 4 and in Line 5. If a transient fault is induced into y_0 in Line 4, only the result y_1 is faulty, the value y_0 in Line 5 is set back to its original correct value. This also represents a computational-safe error. This case shows that memory-safe errors and computational-safe errors are hard to differentiate. They represent two sides of one medal.

2. New Attacks on Plain RSA

In this chapter, we present new attacks on the classic version of the RSA cryptosystem, which we refer to as *plain RSA*. As the classic version, we understand implementations where the modular exponentiation $M^d \bmod N$ is performed using the classical modular exponentiation algorithm, i.e., repeated squaring. We will state results about a faster exponentiation algorithm, i.e., CRT-RSA, in Chapter 3.

We start with a brief overview over existing fault attacks on plain RSA in Section 2.1. Here, we review the first known fault attack on RSA, which was presented by Boneh, DeMillo, and Lipton in [BDL97]. In [BDL97], the authors propose an attack on the right-to-left version of the repeated squaring algorithm only. In Section 2.1.1, we will show that this attack contains minor flaws, which we correct in Section 2.1.2. Afterwards, Section 2.2 is devoted to show that their attack can be extended to the left-to-right version of repeated squaring as well. For an attack on the left-to-right version, an adversary may choose to recover the secret key d starting from the MSBs (Section 2.2.1), or starting from the LSBs (Section 2.2.2). Both versions will be presented and the success probability will be proven for both cases under the assumption that -1 is a quadratic non-residue modulo N. To the best of our knowledge, the correction of the flaw in [BDL97] and the attack on the left-to-right version of repeated squaring have not been published before.

To agree on notation, let us briefly recall the RSA cryptosystem, proposed by Rivest, Shamir, and Adleman in 1978 [RSA78].

Protocol 2.1 (The RSA Cryptosystem).

System parameters:
- *Choose two distinct large primes p and q of the same bit length (in practice, other variants are possible, cf. [Sha95], and several additional restrictions on the choice of p and q apply, cf. [MvOV96, § 8])*
- *Compute $N = p \cdot q$ as the RSA modulus.*
- *The primes p and q are secret values, N is public.*

Key generation:
- *Let $\varphi(N) = (p-1) \cdot (q-1)$ denote Euler's totient function of N.*
- *Choose a public key $3 \leq e \leq \varphi(N) - 2$ coprime to $\varphi(N)$*
- *Compute as the secret key the unique integer $1 \leq d < \varphi(N)$ such that $e \cdot d \equiv 1 \bmod \varphi(N)$ (e.g., using the Extended Euclidean Algorithm).*

Encryption
- *If used for encryption, compute a ciphertext $C := M^e \bmod N$, given a message $M \in \mathbb{Z}_N$.*

Decryption
- *If used for decrypting a ciphertext $C \in \mathbb{Z}_N$, a message is recovered as $M := C^d \bmod N$.*

Signature Generation:
- *If used to create a signature of a message $M \in \mathbb{Z}_N$, the signature $S \in \mathbb{Z}_N$ is created as $S := M^d \bmod N$.*

Verification:
- *If used for verification, a signature $S \in \mathbb{Z}_N$ of a message (or hash value) $M \in \mathbb{Z}_N$ is verified as $(M, S) = true \Leftrightarrow S^e \equiv M \bmod N$.*

The values M, C, and S are often denoted by small letters in the literature. However, our main goal in this chapter is to extend an attack presented in [BDL97] and [BDL01], where RSA messages are denoted by capital letters. Hence, we decided to describe our results in the same notation.
In practice, RSA is used with additional features, e.g., signing a hash value instead of a message, or using a secure padding scheme such as OAEP [BR95b] instead of encrypting a plain message. Details can be found in [MvOV96] or [PKC02]. However, when dealing with smartcards, the padding scheme and the padded message is usually known to the adversary, since the padding is performed outside of the card. Hence, for fault attacks on smartcards, it is reasonable to consider plain RSA as presented in Protocol 2.1.

2.1. The Differential Fault Attack Proposed by Boneh, DeMillo, and Lipton on Right-To-Left Repeated Squaring

In their seminal paper [BDL97] and in its journal version [BDL01], Boneh, DeMillo, and Lipton describe a fault attack on plain RSA signatures. They state that this "was the starting point of our research on fault based cryptanalysis" ([BDL01, p. 6]). Boneh, DeMillo, and Lipton describe the attack on the plain right-to-left version of repeated squaring, presented in this thesis as Algorithm 2.3. In this section, we will recall the attack of Boneh, DeMillo, and Lipton and outline and correct two minor flaws.

Algorithm 2.2: Left-To-Right Rep. Squaring

Input: a message $M \in \mathbb{Z}_N$, a secret RSA key $1 \leq d$, where $l(d)$ denotes the binary length of d, i.e., the number of bits of d, and an RSA modulus N
Output: $M^d \bmod N$

```
# init
 1 Set y := 1

# main
 2 For k from l(d) − 1 downto 0 do
 3      Set y := y^2 mod N
 4      If d_k = 1 then set y := y · M mod N
 5 Output y
```

Algorithm 2.3: Right-To-Left Rep. Squaring

Input: a message $M \in \mathbb{Z}_N$, a secret RSA key $1 \leq d$, where $l(d)$ denotes the binary length of d, i.e., the number of bits of d, and an RSA modulus N
Output: $M^d \bmod N$

```
# init
 1 Set y := 1
 2 Set z := M
# main
 3 For k from 0 to l(d) − 1 do
 4      If d_k = 1 then set y := y · z mod N
 5      Set z := z^2 mod N
 6 Output y
```

For attacks on Algorithms 2.2 and 2.3, we assume the BDL Fault Model, i.e, the Single Bit Fault Model defined in Definition 1.7. This is the fault model used in [BDL97] and [BDL01]. It assumes that a specific variable can be targeted, and that the induced fault flips an unknown, yet uniformly distributed bit of the affected variable. As the targeted variable is used inside a loop, note that Fault Model 1.7 assumes that every iteration of the loop is hit with the same probability and that the actual iteration is not known to the adversary. Both transient and permanent errors are considered if their effects differ.

An adversary is allowed to run an attacked algorithm a polynomial number of times while inducing faults. We assume that the adversary knows both the input messages and the faulty

signatures. The messages do not have to be chosen by the adversary and may be arbitrary. However, the analysis of the attack assumes that the messages are chosen i.i.d. from $\mathbb{Z}_N$ according to the uniform distribution. It is assumed that the adversary is provided with a list $(M_i, \tilde{S}_i)$ of pairs of messages and corresponding faulty signatures. First, we will describe the original attack presented in [BDL01], where Algorithm 2.3 is attacked. Here, the adversary targets the intermediate variable y used on the right hand side of Line 4 of Algorithm 2.3, i.e., he targets the input value y used in $y \cdot z$ and not the result y. However, if the result y on the left hand side is affected, we have the same result as if y was affected on the right hand side one iteration later.

We can now describe a faulty result in detail. Assume that faults are induced into y according to Fault Model 1.7 during some unknown iteration i. Let M_v be an input to Algorithm 2.3, let $S_v = M_v^d$ be the correct result, and let $e(y) = \pm 2^b$, with $0 \leq b < l(N)$, be the error. In this case, the faulty result $\tilde{S}_v$ can be described as

$$\begin{aligned} \tilde{S}_v &\equiv \left(\prod_{j=0}^{i-1} M_v^{2^j d_j} \pm 2^b \right) \cdot \prod_{j=i}^{n-1} M_v^{2^j d_j} \equiv S_v \pm 2^b \cdot \prod_{j=i}^{n-1} M_v^{2^j d_j} \bmod N \\ \Rightarrow S_v &\equiv \tilde{S}_v \pm 2^b \cdot M_v^w \bmod N, \qquad \text{where} \quad w := \sum_{j=i}^{n-1} d_j 2^j \\ \Rightarrow M_v &\equiv \left(\tilde{S}_v \pm 2^b \cdot M_v^w \right)^e \bmod N. \end{aligned} \tag{2.1}$$

Equation (2.1) is evident from Algorithm 2.3. It shows that an adversary does not need to know the correct signatures S_v in order to describe the faulty values.

The main idea in [BDL01] is to recover all bits of d in blocks of m bits starting from the most significant bit d_{n-1}. Here, m is a parameter, such that 2^m is an acceptable amount of offline work. The letter m will be used in this sense throughout this thesis frequently. The attack described in [BDL01] tries to find values w and $\pm 2^b$, such that Equation (2.1) is satisfied. If a value $\tilde{S}_v$ exists, such that w can be generated using known bits of d and at most m unknown bits of d, all of the possible 2^m choices for these m bits can be tested to satisfy Equation (2.1). If the *verification step*, i.e., Equation (2.1), is satisfied for a given choice of w and $\pm 2^b$, it is concluded that the bits of w correctly describe the corresponding bits of d, and $\pm 2^b$ correctly represents the error. Although an adversary does not know which faulty signature $\tilde{S}_v$ resulted from a fault induced at a specific interval of iterations, all faulty signatures can be tested. Boneh, DeMillo, and Lipton show that this approach yields d with probability about $1/2$. The proof relies on the assumption that for each contiguous interval of m iterations, at least one iteration is hit by a fault affecting the variable y, accounting for a faulty signature $\tilde{S}_v$. This assumption holds with probability $1/2$, as shown by the following fact. It has been proved in [BDL01, Theorem 2.1].

Fact 2.4 (Number of Necessary Attacks).
Let $x = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$ and let $\mathcal{C}$ be the set of all contiguous intervals of length $m < n$ in x. If $c = (n/m) \cdot \log(2n)$ bits of x are chosen uniformly independently at random, then the probability that each interval in $\mathcal{C}$ contains at least one chosen bit is at least $1/2$.

Fact 2.4 shows that the adversary needs to collect at least $c = (n/m) \cdot \log(2n)$ many faulty signatures to guarantee a success probability of $1/2$. This yields a trade-off between m and c, i.e., between the acceptable amount of offline work and the required number of faulty signatures.

The idea described above allows for an inductive approach. We present this approach from [BDL01] as Algorithm 2.5.

Algorithm 2.5: Attack on Right-To-Left Repeated Squaring, Original Version

Input: Access to Algorithm 2.3, n the length of the secret key d, e the RSA public key, m a parameter for acceptable amount of offline work.

Output: The secret key d with probability at least 1/2.

\# **Phase 1:** Collect Faulty Outputs

1 Set $c := (n/m) \cdot \log(2n)$

2 Create c faulty outputs of Algorithm 2.3 on inputs M_i by inducing faults according to Fault Model 1.7 in the intermediate variable y in random iterations.

3 Collect the set $\mathcal{S} = \{(M_i, \tilde{S}_i) |\ M_i$ an input to Algorithm 2.3 and $\tilde{S}_i$ a faulty output$\}$.

\# **Phase 2:** Inductive Retrieval of Secret Key Bits

4 Set $s := n$ representing the index of the highest known bit of d.

5 While $(s > 0)$ do

\# Compute the known MSB part of the exponent d.

6 Set $w_s := \sum_{j=s}^{n-1} 2^j d_j$

\# Try all possible bit pattern with length $r \leq m$.

7 For all lengths $r = 1, 2, \ldots m$ do

8 For all candidate r-bit vectors $x = (x_{s-1}, x_{s-2}, \ldots x_{s-r})$ of length r do

\# Compute the test candidate w

9 Set $w := w_s + \sum_{j=s-r}^{s-1} 2^j x_j$

\# Verification Step: Verify the test candidate using Equation (2.1)

10 for all $(M_j, \tilde{S}_j) \in \mathcal{S}$ do

11 For all b from 0 to $n-1$ do

12 if $\left(\tilde{S}_j \pm 2^b M_j^w\right)^e \equiv M_j \bmod N$ then

13 conclude that $d_j = x_j$ for all $s - r \leq j \leq s - 1$,

14 set $s := s - r$ and continue at Line 5

15 **Output** d

Using Algorithm 2.5, Boneh, DeMillo, and Lipton show that the following theorem holds. The proof relies on Fact 2.4.

Theorem 2.6 (Fault Attack on Right-To-Left Repeated Squaring).
Let $N = p \cdot q$ be an n-bit RSA modulus. For any $1 \leq m \leq n$, given $(n/m)\log(2n)$ pairs $(M_i, \tilde{S}_i)$, the secret exponent d can be extracted from Algorithm 2.3 with probability at least 1/2. The probability is over the location of the induced faults and random messages $M_i \in \mathbb{Z}_N$. The algorithm's running time is dominated by the time it takes to perform $O((2^m n^3 \log^2(n))/m^2)$ full modular exponentiations modulo N.

Algorithm 2.5 assumes that n, the exact length of the secret key d, is given. In practice, this is usually not the case, however, the assumption is reasonable. This is due to the fact that there are at most $\log(N)$ many possible values for n, hence, an adversary can always guess n and run Algorithm 2.5 for all possible guesses. This adds only a polynomial factor to the overall runtime, which is acceptable. In practice, n will always be large, hence, even in the case where n needs to be guessed, one may expect few trials if n is tested in decreasing order.

The running time of Algorithm 2.5, stated in Theorem 2.6, has been proved in [BDL01].

Since it can be derived easily by counting the operations, we refer the reader to [BDL01] for details. However, in order to prove the success probability claimed in Theorem 2.6, it must be shown that a wrong candidate bit pattern (as chosen in Line 8 of Algorithm 2.5) will satisfy the verification step in Line 12 with negligible probability only. The analysis of this case yields the following lemma in [BDL01].

Lemma 2.7 (Accepting Wrong Candidates).
Let $\delta > 0$ be a fixed constant. For all n-bit RSA moduli $N = p \cdot q$ at least one of the following claims holds:

1. *The probability that a wrong candidate x' passes the test of Line 12 in Algorithm 2.5 is less than $1/n^{\delta}$. The probability is over the random choices of messages $M_i \in \mathbb{Z}_N$ given to the attack algorithm and the random choice of the decryption exponent d.*

2. *A non-trivial factor of N can be computed in expected polynomial time (in n and 2^m).*

2.1.1. Flaws in the Plain RSA Attack

The original proof of Lemma 2.7 from [BDL01] analyzes the situation, where a faulty signature $\tilde{S}$ can be described by two different bit patterns and (possibly) two error values. In this case, the wrong bit pattern and the correct bit pattern cannot be distinguished by the verification step, and a wrong candidate bit pattern could be accepted. Hence, Algorithm 2.5 would fail.

The analysis of this situation starts by assuming that some faulty signature $\tilde{S}$ is given with $(M, \tilde{S}) \in \mathcal{S}$. $\tilde{S}$ resulted from a fault being induced during some iteration, say during iteration i. In this case, we have

$$\tilde{S} \equiv S \pm 2^{b_1} M^{w_1} \bmod N \qquad \text{where} \quad w_1 = \sum_{j=i}^{n-1} 2^j d_j. \tag{2.2}$$

Here, $S \equiv M^d \bmod N$ represents the correct signature. Now assume that a wrong bit pattern x' is accepted in Line 12. This implies that $\tilde{S}$ can also be identified as $\tilde{S} \equiv S \pm 2^b M^w \bmod N$, where w is defined as in Step 9 of Algorithm 2.5 using the bits of x'. The pattern x' is a wrong bit pattern, if it holds that $x'_j \neq d_j$ for some $0 \leq j < n$. The authors claim that this implies that $w \neq w_1$ must hold. However, as we will show below, this is not true in general.

If $w \neq w_1$ was true, then we would have

$$S \pm 2^{b_1} M^{w_1} \equiv S \pm 2^b M^w \bmod N.$$

Rearranging terms we get $M^{w-w_1} \equiv \pm 2^{b_1 - b} \bmod N$. In other words, M must be a root of a polynomial of the form

$$t^{w-w_1} \equiv a \bmod N$$

for some known constant $a = \pm 2^{b_1 - b}$. This situation allows to compute a non-trivial factor of N in expected polynomial time if it occurs with a non-negligible probability, as we will show in detail later.

A Flaw in the Original Proof of Lemma 2.7. We will now show that the reasoning above contains a subtle flaw, caused by implicit assumptions, which are not ensured by Algorithm 2.5.

We start by assuming that Algorithm 2.5 already recovered several bits of d, say the $n-s$ most significant bits $d_{n-1}, d_{n-2}, \ldots, d_s$. At this time, the algorithm enumerates all r-bit patterns x with $1 \leq r \leq m$ and tries to verify these guesses in Line 12. We assume that one of these guesses, namely $x' = (x'_{s-1}, x'_{s-2}, \ldots, x'_{s-r'})$ satisfies the verification step for some error value $\pm 2^b$ and some faulty final signature $\tilde{S}$, with $(M, \tilde{S}) \in \mathcal{S}$, although x' is a wrong bit pattern. We assume that the values w_1 (defined as in Equation (2.2)) and $\pm 2^{b_1}$ correspond to the actual location of the fault during the computation of $\tilde{S}$. In the original proof of Lemma 2.7, it is implicitly assumed that this implies that the corresponding exponents

$$w_1 := \sum_{j=i}^{n-1} 2^j d_j \tag{2.5}$$

$$\text{and} \qquad w := \sum_{j=s}^{n-1} 2^j d_j + \sum_{j=s-r'}^{s-1} 2^j x'_j \tag{2.6}$$

are different. However, this is only guaranteed if $i = s - r'$. Otherwise, we may encounter a situation, where this does not hold. As a counterexample, consider the following situation, where we have $i = s - r$ for some $r \geq 0$. Let $d_{s-r-1} = 1$. This yields

$$\tilde{S} = \left(\prod_{j=0}^{s-r-1} M^{2^j d_j} \pm 2^{b_1} \right) \cdot \prod_{j=s-r}^{n-1} M^{2^j d_j} \qquad \text{for some} \quad 1 \leq s - r < n - 1,$$

for some $(M, \tilde{S}) \in \mathcal{S}$. Let $D_{s-r} := (d_{n-1}, d_{n-2}, \ldots, d_{s-r})$ denote an upper part of the binary expansion of d, and let $\circ$ denote concatenation of bit sequences. Assume that the bits $d_{n-1}, d_{n-2}, \ldots, d_s$ are known. Now let $x' = D_{s-r} \circ 0$ be a guess for the $n - s + r + 1$ most significant bits of d. Let w be defined as in Equation (2.6) for x', and let w_1 be defined as in Equation (2.5) for $i = s - r$. Obviously, since we assume that $d_{s-r-1} = 1$, x' is a wrong candidate. Nevertheless, we have

$$w = \sum_{j=s-r}^{n-1} 2^j d_j + \sum_{j=s-r-1}^{s-r-1} 2^j x'_j = \sum_{j=s-r}^{n-1} 2^j d_j + 0 = \sum_{j=s-r}^{n-1} 2^j d_j = w_1.$$

$$\Rightarrow \qquad S \pm 2^{b_1} M^w \equiv \tilde{S} \equiv S \pm 2^{b_1} M^{w_1} \bmod N,$$

where $S = M^d \bmod N$. This observation holds for all x' which satisfy the condition $x' = D_{s-r} \circ 0^\alpha$, where $\alpha > 0$. It is easy to see that as a zero does not add to w, we still have $w = w_1$. Yet, this does not yield a non-trivial factor of N. The implicit assumption made incorrectly in Lemma 2.7 is that different test patterns, which both satisfy the verification step yield different values for w and w_1, i.e., that "$\tilde{S}_v \equiv S_v \pm 2^{b_1} M_v^{w_1} \bmod N$ for some b_1, w_1 with $w_1 \neq w$" (Lemma 2.2 in [BDL01]). We will show in the following, that this is not guaranteed by the choices of x' made in Algorithm 2.5.

A Flaw in Algorithm 2.5. Algorithm 2.5 also contains a minor flaw: it is compelled to create candidate bit patterns x', which resemble the counterexample constructed above.

We consider the situation of Algorithm 2.5 at the beginning of round s. Again, we may assume that Algorithm 2.5 already recovered the $n - s$ most significant bits of d, i.e., $D_s =$

$(d_{n-1}, d_{n-2}, \ldots, d_s)$. According to the verification step, Line 12, this implies that these $n-s$ bits must have been recovered using the correct guess for the $n-s$ most significant bits and the correct guess for the error term $\pm 2^b$, together with some pair $(M_j, \tilde{S}_j) \in \mathcal{S}$, $1 \leq j \leq \#\mathcal{S}$, such that

$$\left(\tilde{S}_j \pm 2^{b_1} M_j^{w_1}\right)^e \equiv M_j \bmod N, \qquad \text{where} \quad w_1 := \sum_{j=s}^{n-1} 2^j d_j. \tag{2.8}$$

In the next round of Algorithm 2.5, all r-bit vectors $x = (x_{s-1}, x_{s-2}, \ldots, x_{s-r})$ are tested starting with $r = 1$. Naturally, one would enumerate all bit vectors starting with the bit vector $x = 0$, followed by $x = 1$, $x = 00$, $x = 01$, $x = 10$, $x = 11$, etc. In [BDL01], no restriction on the choice of the bit pattern x is mentioned. Algorithm 2.5 now starts with $x = 0$, guesses a value $\pm 2^b$ for the error and tests all faulty signatures $(M_j, \tilde{S}_j) \in \mathcal{S}$. In [BDL01], it is not mentioned that pairs, which have already been used to recover a fraction of d are removed from the set $\mathcal{S}$, therefore, we assume that we try all faulty pairs $(M_j, \tilde{S}_j) \in \mathcal{S}$ again. At one time, the same $(M_j, \tilde{S}_j)$ and the same $\pm 2^{b_1}$ as in Equation (2.8) are tested. In this case, we have that w is build with the bit pattern $D_s \circ 0$, which represents the counterexample above. If $d_{s-1} = 1$, a wrong bit pattern is accepted, since we have

$$\left(\hat{S} \pm 2^{b_1} M_j^{w}\right)^e \equiv M_j \bmod N, \qquad \text{where} \quad w := \sum_{j=s}^{n-1} 2^j d_j + \sum_{j=s-r}^{s-1} 2^j x_j = w_1 + 0 = w_1. \tag{2.9}$$

But even worse: if the algorithm continues in the way described, it will now guess that all successive bits are zero, because Equation (2.9) holds for any sequence x, which consists of zeros only. This implies that once some leading bits have been recovered, no more non-zero bits will be recovered correctly. Moreover, whenever a guess x satisfies the test in Equation (2.9), all other bit strings which are equal to x followed by an arbitrary number of zeros, would also be accepted as correct guesses.

The algorithm fails due to two problems. First, zero bit strings x may be tested, which lead to erroneously verified bit patterns. Second, faulty signatures $\tilde{S}_j$, which have been used before, stay in the set S and are tested again in the next round. This shows that Algorithm 2.5 is not correct in the compact form stated in [BDL01].

We will show now that both, the proof of Lemma 2.7 and Algorithm 2.5 can be fixed easily.

2.1.2. Correcting Algorithm 2.5 and Lemma 2.7.

In order to fix Algorithm 2.5, we need to ensure that no zero postfixes occur. Trivially, this can be achieved by demanding that only bit patterns x are tested which end with the digit 1. This choice guarantees that every new candidate x represents a new partial exponent w. Therefore, it may no longer happen that an old faulty value is redetected in the way described in Equation (2.9). We will show that this requirement also ensures that Lemma 2.7 holds. Although the new restriction on x appears as a trivial modification at first sight, a new problem is created. Assume that from bit position $s-1$ through $s-m$ all bits of d are equal to zero. In the case of such zero blocks, the correct bit pattern x would never be tested under the new requirement and Algorithm 2.5 would not come up with a valid guess. Fortunately, this is a problem, which can be solved with some additional considerations: we will show that setting the next bit to 0 if no matching pattern is found will solve this problem. However, this situation requires special attendance and will be formalized in Definition 2.8.

Definition 2.8 (Zero Block Failure for Algorithm 2.3).
Let $x = (x_{n-1}, x_{n-2}, \ldots, x_0)$ be a positive integer with $n := l(x)$, and let $x' = (x_{j-1}, x_{j-2}, \ldots, x_{j-m})$, $m \leq j \leq n$, be a block of m bits in x. Assume that faults are induced during the execution of the right-to-left repeated squaring algorithm, Algorithm 2.3. If all faults, which have been induced during iterations $j-1, j-2, \ldots, j-m$, are restricted to iterations $j-1, j-2, \ldots, j-r$, $1 \leq r \leq m$, where $x_{j-1} = x_{j-2} = \ldots = x_{j-r} = 0$, the situation is called a Zero Block Failure. *We assume that at least one fault was induced during the iterations $j-1, j-2, \ldots, j-m$.*

A Zero Block Failure is named after the fact that errors in a block of zeros will not be detected as errors within that block. Assume that $\tilde{S}_1$ and $\tilde{S}_2$ are two faulty values returned by Algorithm 2.3 while being under attack. Let i_1 and i_2 denote the iterations, in which faults were induced into the intermediate value y. We assume w.l.o.g. that $i_2 \leq i_1$. We further assume that the value y on the right side of Line 4 is affected by the induced fault, i.e., prior to the multiplication with z. We denote by e_1 and e_2 the errors inflicted during the computation of $\tilde{S}_1$ and $\tilde{S}_2$ respectively. We have

$$\tilde{S}_1 = \left(\prod_{j=0}^{i_1-1} m^{2^j d_j} + e_1 \right) \cdot \prod_{j=i_1}^{n-1} m^{2^j d_j} \tag{2.10}$$

$$\text{and} \qquad \tilde{S}_2 = \left(\prod_{j=0}^{i_2-1} m^{2^j d_j} + e_2 \right) \cdot \prod_{j=i_2}^{n-1} m^{2^j d_j}. \tag{2.11}$$

Equations (2.10) and (2.11) show that if $e_1 = e_2$ and $d_{i_2} = d_{i_2+1} = \ldots = d_{i_1-1} = 0$, it holds that $\tilde{S}_1 = \tilde{S}_2$. This is trivial since whenever $d_i = 0$, we have a factor of 1 in any of the products. Therefore, if two faults are induced within a zero block of bits and the error values e_1 and e_2 are equal, the two results cannot be distinguished. If the bits are recovered starting from the most significant bit, it is not possible to determine how many zero bits — if any — follow the last detected 1. Any error induced in iterations $j > i_2$ where it holds that $d_j = d_{j-1} = \ldots = d_{i_2} = 0$ will be detected as $\tilde{S}_2$ if the error e_2 is the same as e_1. Hence, tailing zeros must be neglected, because their number cannot be determined correctly. However, the fact that tailing zeros do not change the value of $\tilde{S}$ allows us to write

$$\tilde{S} = \left(\prod_{j=0}^{i-1} m^{2^j d_j} + e_2 \right) \cdot \prod_{j=i}^{n-1} m^{2^j d_j} \qquad \text{with} \quad d_i = 1, \tag{2.12}$$

for some unknown i. Moreover, if it is known that the error happened in a certain iteration k, we may specify i in greater detail: here, we have $i := \max\{j \mid d_j = 1 \wedge j \geq k\}$. The case where $\tilde{S} = S \pm 2^b$, i.e., when $k = n-1$ and the maximum does not exist, will be neglected, because it cannot yield any information. Any adversary can deliberately create values $S \pm 2^b$ without knowing or learning anything about the secret key.

The above considerations about Zero Block Failures show that whenever such a Zero Block Failure is encountered, our new algorithm would not be able to guess a correct value for the next block of m bits. However, we may interpret a missing valid guess as an indicator for a Zero Block Failure. As a Zero Block Failure implies that at least a single zero occurs, we are able to recover the first following bit, which must be 0. As before, our indicator will be wrong at least once during Algorithm 2.9 with probability at most 1/2. This happens in the case when none of the $c = (n/m)\log(2n)$ induced faults happened in the block of m bits under investigation.

The considerations explained above allow us to present the corrected Algorithm to attack the right-to-left repeated squaring algorithm. The modified parts compared to Algorithm 2.5 have been highlighted. We will prove the success probability afterwards.

Algorithm 2.9: Attack on Right-To-Left Repeated Squaring, Corrected Version

Input: Access to Algorithm 2.3, n the maximal length of the secret key d, e the RSA public key, m a parameter for acceptable amount of offline work.

Output: The secret key d with probability at least 1/2.

\# **Phase 1:** Collect Faulty Outputs

1 Set $c := (n/m) \cdot \log(2n)$

2 Create c faulty outputs of Algorithm 2.3 on inputs M_i by inducing faults according to Fault Model 1.7 in the intermediate variable y in random iterations.

3 Collect the set $\mathcal{S} = \{(M_i, \tilde{S}_i) |\ M_i$ an input to Algorithm 2.3 and $\tilde{S}_i$ a faulty output$\}$.

\# **Phase 2:** Inductive Retrieval of Secret Key Bits

4 Set $s := n$ representing the index of the highest known bit of d.

5 While $(s > 0)$ do

\# Compute the known MSB part of the exponent d.

6 Set $w_s := \sum_{j=s}^{n-1} 2^j d_j$

\# Try all possible bit pattern with length $r \leq m$.

7 For all lengths $r = 1, 2, \ldots m$ do

8 For all candidate r-bit vectors $x = (x_{s-1}, x_{s-2}, \ldots x_{s-r+1}, 1)$ of length r do

\# Compute the test candidate w

9 Set $w := w_s + \sum_{j=s-r}^{s-1} 2^j x_j$

\# Verification Step: Verify the test candidate using Equation (2.1)

10 for all $(M_j, \tilde{S}_j) \in \mathcal{S}$ do

11 For all b from 0 to $n-1$ do

12 if $\left(\tilde{S}_j \pm 2^b M_j^w\right)^e \equiv M_j \bmod N$ then

13 conclude that $d_j = x_j$ for all $s - r \leq j \leq s - 1$,

14 delete $(M_j, \tilde{S}_j)$ from the set $\mathcal{S}$,

15 set $s := s - r$ and continue at Line 5

\# Handle a Zero Block Failure

16 If no test candidate satisfied the verification step, then

17 assume that $d_{s-1} = 0$ and set $s := s - 1$.

18 **Output** d

Remark. Algorithm 2.9 has been abbreviated in a minor detail. We base the success of Algorithm 2.9 on the "lucky case" of Fact 2.4. However, we are not guaranteed that a fault was induced during the first iteration $i = 0$ of the repeated squaring algorithm. We can recover bits of d up to the lowest iteration, in which Algorithm 2.3 ever suffered a fault. This implies that even in the "lucky case" of Fact 2.4, we may not be able to recover the $m-1$ least significant bits of d in the manner described by Algorithm 2.9. However, since 2^m operations represent an acceptable amount of offline work, it is possible to recover those $m-1$ least significant bits with exhaustive search. Note that this consideration also applies to the original version, Algorithm 2.5.

Since including the special case mentioned above would add complexity at a significant loss in clarity, we decided to state the attack algorithm in the abbreviated form.

Algorithm 2.9 is very similar to the original version, Algorithm 2.5. The difference is the new choice of the test bit pattern x, which must end with the digit 1 now and, as a consequence, the handling of Zero Block Failures in Lines 16 and 17. We will now prove that this correction yields a correct and successful attack on the right-to-left repeated squaring algorithm. First, we investigate the situation encountered at each round of Algorithm 2.9, i.e., at each time, the loop counter s is decreased. We will first show that if no valid candidate pattern x is found which satisfies the verification step in Line 12 of Algorithm 2.9, a Zero Block Failure is correctly detected. Then we will show that a false pattern will satisfy the verification step with negligible probability only. We conclude this section with an analysis of the running time of Algorithm 2.9.

Lemma 2.10 (Correct Recovery).
In each round s of Algorithm 2.9, we have one of three cases.

1. *For some j, there is a block $j-1, j-2, \ldots, j-m$ of m contiguous iterations, during which no fault was induced into the targeted variable.*
2. *Algorithm 2.9 finds a candidate bit vector $x = (x_{s-1}, x_{s-2}, \ldots, x_{s-r+1}, 1)$ of length $1 \leq r \leq m$ which satisfies the verification step in Line 12.*
3. *A Zero Block Failure is detected correctly, revealing the bit $d_{s-1} = 0$.*

Proof:

The attack algorithm, Algorithm 2.9, relies on the assumption that Part 1 is false. If there is a block of m iterations, such that no pair $(M_v, \tilde{S}_v) \in \mathcal{S}$ resulted from a fault induced into the intermediate variable y during any of the m iterations, we do not claim that Algorithm 2.9 will be able to recover d. However, as Fact 2.4 shows, this event occurs with probability at most 1/2. Therefore, we now assume that we have a "lucky case" and Part 1 is not satisfied.

To show that a "lucky case" implies either Part 2 or Part 3, we now assume that Part 2 is not satisfied. We claim that this implies that Part 3 is satisfied.

We assume that a certain round of Algorithm 2.9 is given, i.e., assume that the $n-s$ most significant bits $d_{n-1}, d_{n-2}, \ldots, d_s$ of d are already known and Algorithm 2.9 now aims to recover the next $1 \leq r \leq m$ bits of d. If Part 2 is not satisfied, Algorithm 2.9 did not test any bit pattern of length $1 \leq r \leq m$ ending in the digit 1 successfully. We define

$$r' := \max\{r \mid s-1 \leq s-r \leq s-m \text{ and } y \text{ has been hit by a fault during iteration } r\}.$$

Since we assume that Part 1 is not satisfied, we know that at least one of the intermediate variables y computed during iterations $s-1, s-2, \ldots, s-m$ has been hit by a fault. This implies that r' exists and that $1 \leq r' \leq m$.

A Zero Block Failure is correctly detected if all bits $d_{s-1}, d_{s-2}, \ldots, d_{s-r'}$ are equal to zero, and in this case Algorithm 2.9 correctly recovers the next bit $d_{s-1} = 0$. If a Zero Block Failure is detected incorrectly, there is at least one bit $d_{s-k} = 1$ with $1 \leq k \leq r'$. We now

assume that a Zero Block Failure is detected incorrectly and choose k to be maximal, i.e., $s-k$ is minimal and closest to $s-r'$. This implies that

$$\sum_{j=s-r'}^{s-k-1} d_j 2^j = \sum_{j=s-r'}^{s-k-1} 0 \cdot 2^j = 0.$$

We consider the pair $(M_v, \tilde{S}_v) \in \mathcal{S}$, which resulted from a fault induced into the intermediate variable y during iteration $s-r'$. Consider the test bit string $x = (d_{s-1}, d_{s-2}, \ldots, d_{s-k+1}, 1)$ of length $1 \leq k \leq r' \leq m$. Obviously, x is one of the test patterns chosen in Line 8. Let $e_{s-r'}$ denote the fault which has been induced. Given $e_{s-r'}$, $\tilde{S}_v$, $S_v = M_v^d \bmod N$, and x, we have

$$\begin{aligned}
\tilde{S}_v &= \left(\prod_{j=0}^{s-r'-1} M_v^{2^j d_j} + e_{s-r'} \right) \cdot \prod_{j=s-r'}^{s-k-1} M_v^{2^j d_j} \cdot \prod_{j=s-k}^{n-1} M_v^{2^j d_j} \\
&= \left(\prod_{j=0}^{s-k-1} M_v^{2^j d_j} + e_{s-r'} \right) \cdot \prod_{j=s-k}^{n-1} M_v^{2^j d_j} = S_v + e_{s-r'} \cdot \prod_{j=s-k}^{n-1} M_v^{2^j d_j} = S_v + e_{s-r'} \cdot M_v^w, \\
&\quad \text{where } w := \sum_{j=s-k}^{n-1} 2^j d_j \quad \text{as in Line 9 of Algorithm 2.9.} \\
&\Rightarrow \left(\tilde{S}_v - e_{s-r'} \cdot M_v^w \right)^e \equiv M_v \bmod N. \qquad (2.14)
\end{aligned}$$

Equation (2.14) shows that the pattern x satisfies the verification step. However, this contradicts our assumption that Part 2 was false and Algorithm 2.9 did not find a satisfying bit pattern. Therefore, if Part 2 is not satisfied, then a Zero Block Failure cannot be detected incorrectly. This proves the lemma. □

Lemma 2.10 shows that if our basic assumption, the "lucky case" of Fact 2.4, is true, then Zero Block Failures are always detected correctly. It remains to show that Algorithm 2.9 also recovers bits of d correctly, if a satisfying bit pattern x is found. Although it is not impossible that a false bit pattern satisfies the verification step and is mistaken as the the correct pattern, Lemma 2.7 shows that such a *false positive* occurs with negligible probability only. Since we consider the original proof of Lemma 2.7 to be not convincing in several details, we present a different approach here. It relies on the following two lemmas, which we will use again later in this chapter.

Lemma 2.11 (Factorization with Polynomials Having Many Roots).
Let $N = p \cdot q$ be an n-bit RSA modulus. Let W be a set of polynomials $p(t) = t^w - a \in \mathbb{Z}_N[t]$, with $w \leq 2N$ for all w. Furthermore, let $\#W < n^\tau$ for some constant $\tau > 0$. Let $\delta > 0$ be a positive constant. In this case, at least one of the following cases holds:

1. *The probability that a random $M \in \mathbb{Z}_N$ is a root of at least one of the $\#W$ many polynomials $p(t)$ is less than $1/n^\delta$.*

2. *A non-trivial factor of N can be computed in expected polynomial time.*

Proof:

We assume that Part 1 of the lemma is false. We will show that this yields a non-trivial factor of N in expected polynomial time.

We assume that a random $M \in \mathbb{Z}_N$ is a root of at least one of the $\#W$ many polynomials $p(t) = t^w - a \bmod N$ with probability at least $1/n^\delta$. Here, M is chosen according to the uniform distribution, which will also be assumed for all random choices in the remainder of this proof. Since we have at most n^τ many different polynomials $p(t)$, we have at least one designated polynomial $\bar{p}(t) = t^{\bar{w}} - \bar{a} \bmod N$, such that a random M is a root of $\bar{p}(t)$ with probability at least $1/n^{\tau+\delta}$ by the pigeon hole principle.

Now let M_1 be a root of $\bar{p}(t)$. For every root M_2 of $\bar{p}(t)$, it holds that

$$M_1^{\bar{w}} - \bar{a} \equiv M_2^{\bar{w}} - \bar{a} \bmod N \qquad \Rightarrow \qquad \left(\frac{M_2}{M_1}\right)^{\bar{w}} \equiv 1 \bmod N.$$

Since we can write $M_2 = M_1 \cdot (M_2/M_1)$, we know that every root of $\bar{p}(t)$ is the same as M_1, multiplied by a $\bar{w}^{\text{th}}$ root of unity modulo N. Hence, we have as many $\bar{w}^{\text{th}}$ roots of unity modulo N as we have roots of $\bar{p}(t)$. This implies that with probability at least $1/n^{\tau+\delta}$, a random $M \in \mathbb{Z}_N$ is a $\bar{w}^{\text{th}}$ root of unity modulo N.

This is a very high probability, which allows to use a well-known algorithm from [MvOV96, § 8.2.2.] to factor N. The algorithm takes $\bar{w}$ and N as input and factors N with probability $1/2$ in polynomial time once a value $M \in \mathbb{Z}_N$ satisfying $M^{\bar{w}} \equiv 1 \bmod N$ is found. For the proof of the success probability of this algorithm, we refer the reader to [MvOV96].

In our case, we do not know the value $\bar{w}$, however, we know that $\bar{w}$ is from a set of at most n^τ many values. Hence, we can repeat the algorithm described in [MvOV96] n^τ times with all possible inputs until the correct one is used. This yields the following algorithm, which is a modified version of the algorithm from [MvOV96].

Algorithm 2.12: w^{th} Root of Unity Factorization

Input: A set W of pairs (w, a) with integers w and a composite integer N
Output: A non-trivial factor of N

1 Choose $\mathsf{M} \in \mathbb{Z}_\mathsf{N}$ uniformly at random
2 If $\gcd(\mathsf{M}, \mathsf{N}) \notin \{1, \mathsf{N}\}$ then **output** $\gcd(\mathsf{M}, \mathsf{N})$.
3 For all $\mathsf{p(t)} = \mathsf{t^w} - \mathsf{a} \in \mathsf{W}$ do
4 If $\mathsf{M^w} \equiv 1 \bmod \mathsf{N}$ then
5 Determine (s, r) such that $\mathsf{w} = 2^\mathsf{s} \cdot \mathsf{r}$ with r odd
6 Compute $\mathsf{M^r}, \mathsf{M^{2r}}, \mathsf{M^{2^2 r}}, \mathsf{M^{2^3 r}}, \ldots$ until $\mathsf{M^{2^k r}} \equiv 1 \bmod \mathsf{N}$
7 If $\mathsf{M^{2^{k-1} r}} \not\equiv \pm 1 \bmod \mathsf{N}$ then **output** $\gcd(\mathsf{M^{2^{k-1} r}} - 1, \mathsf{N})$
8 Repeat from Step 1

Algorithm 2.12 selects random $M \in \mathbb{Z}_N$. With probability at least $1/n^{\tau+\delta}$, M is a $\bar{w}^{\text{th}}$ root of unity as described above. Then, M yields a non-trivial factor of N with probability $1/2$ (see [MvOV96] for the proof of this property). Hence, we expect $2n^{\tau+\delta}$ many random choices in Line 1 until the selected M allows to factor N using $\bar{w}$. For all choices, we need to compute M^w for all w given by polynomials $p(t) \in W$. Since $\#W < n^\tau$ and $w \leq 2N$, we need to perform at most $O(2n^{\tau+1})$ many modular multiplications to check whether $M^w \equiv 1 \bmod N$ for any of the values w. For those choices, where we find that M is indeed

a w^{th} root of unity for some w, we need to compute the values in Line 6 and the gcd in Line 7. This requires at most n multiplications and a gcd, which can be computed using $O(n)$ multiplications. Therefore, we expect the algorithm to compute a non-trivial factor of N using at most

$$O\left(2n^{\tau+\delta} \cdot 2n^{\tau+1} + 2n\right) = O\left(n^{2\tau+\delta+1}\right)$$

many modular multiplications. This clearly is a polynomial running time. Obviously, the running time will be better, if not all $p(t) \in W$ share the same w. This holds since we are content with a value M to be a w^{th} root of unity for any of the at most n^{τ} values w. □

Lemma 2.13 (Factorization with Polynomials Having Many Roots for Random d). *Let $N = p \cdot q$ be an n-bit RSA modulus. Let d be an RSA decryption exponent, and let W_d be a set of polynomials $p(t) = t^w - a \in \mathbb{Z}_N[t]$, which can be computed in polynomial time (in n). Furthermore, let $w \leq 2N$ for all polynomials. Assume that there exists a fixed constant $\tau > 0$, such that $\#W_d < n^{\tau}$ for all d. Let $\delta > 0$ be a positive constant. In this case, at least one of the following cases holds.*

1. *The probability that a number $M \in \mathbb{Z}_N$ is a root of any polynomial $p(t)$ is less than $1/n^{\delta}$. The probability is over random choices of of numbers $M \in \mathbb{Z}_N$ and random choices of the decryption exponent d.*
2. *A non-trivial factor of N can be computed in expected polynomial time.*

Proof:

Let $\mathcal{A}$ be the event that a number $M \in \mathbb{Z}_N$ is a root of any of the polynomials $p(t)$ mod N. For a given choice of d, the polynomials $p(t)$ are from the set W_d of magnitude at most n^{τ}. We denote by $X(d) = X(\mathcal{A}|d)$ the probability that $\mathcal{A}$ occurs given d. We denote by $P(d)$ the probability that a specific d is chosen. For the whole attack, we assume that the secret key d is chosen randomly according to the uniform distribution from all possible values, i.e., from the set of all values smaller than $\varphi(N)$ and coprime to $\varphi(N)$. According to the standard total probability theorem, we have

$$\text{Prob}[\mathcal{A}] = \sum_{d} X(\mathcal{A}|d) \cdot P(d).$$

We now assume that Part 1 does not hold. In this case, we know that $1/n^{\delta} \leq \text{Prob}[\mathcal{A}]$.

Now we may divide the set of all possible secret keys d into two disjoint subsets. The first subset contains all those d, where a message M satisfies any of the modular polynomials from W_d with high probability, i.e., where $X(d) > 1/(2n^{\delta})$. The other subset contains all other values d, where $X(d) \leq 1/(2n^{\delta})$. We will show that if we have a d from the first set, then we can factor N with high probability. Hence, we first intend to show that a random d is from the first set with high probability. This would allow to factor N with high probability in polynomial time.

We have

$$\frac{1}{n^{\delta}} \leq \text{Prob}[\mathcal{A}] = \sum_{d:X(d)>1/(2n^{\delta})} X(\mathcal{A}|d) \cdot P(d) + \sum_{d:X(d)\leq 1/(2n^{\delta})} X(\mathcal{A}|d) \cdot P(d)$$

$$\Rightarrow \frac{1}{n^\delta} \leq \sum_{d:X(d)>1/(2n^\delta)} P(d) + \frac{1}{2n^\delta} \sum_{d:X(d)\leq 1/(2n^\delta)} P(d) \tag{2.18}$$

$$\Rightarrow \frac{1}{2n^\delta} \leq \sum_{d:X(d)>1/(2n^\delta)} P(d). \tag{2.19}$$

Equation (2.18) holds, since $X(d)$ is a probability, hence, $0 \leq X(d) \leq 1$. This implies that we can replace all occurrences of $X(d)$ by 1 in the first sum, which may not decrease the right hand side. For the second sum, we know that $X(d) \leq 1/(2n^\delta)$, hence, we do not decrease the sum, if we replace each occurrence by the upper bound $1/(2n^\delta)$. Equation (2.19) follows from replacing the rightmost sum of probabilities $P(d)$ in Equation (2.18) by the upper bound 1 and subtracting the resulting value $1/(2n^\delta)$. Equation (2.19) shows that with probability at least $1/(2n^\delta)$, a randomly chosen d has $X(\mathcal{A}|d) \geq 1/(2n^\delta)$ if Part 1 is false.

In order to factor N, we subsequently choose values d and compute the set W_d. Each polynomial can be computed in polynomial time and since there are at most n^τ polynomials in each set W_d, the set W_d can be computed in polynomial time. We know that w is less than $2N$. Given this data, we can apply Lemma 2.11, which shows that a non-trivial factor of N can be computed in expected polynomial time. In Lemma 2.11, Part 2 applies if the choice of d satisfies $X(d) > 1/(2n^\delta)$. We know from the considerations above that we expect $2n^\delta$ random choices of decryption exponents d until $X(d) > 1/(2n^\delta)$ is satisfied. This proves Part 2 of this lemma. □

We can now state the correct version of Lemma 2.7, which we will present as Lemma 2.14.

Lemma 2.14 (Accepting Wrong Candidates).
Let $\delta > 0$ be a fixed constant. For all n-bit RSA moduli $N = p \cdot q$ at least one of the following claims holds:

1. *The probability that a wrong candidate x' passes the test of Line 12 in Algorithm 2.9 is less than $1/n^\delta$. The probability is over the random choices of messages $M_i \in \mathbb{Z}_N$ given to the attack algorithm and the random choice of the decryption exponent d.*

2. *A non-trivial factor of N can be computed in expected polynomial time (in n and 2^m).*

Proof:

We show an algorithm that factors all RSA moduli N for which Part 1 is false. The algorithm works as follows: it picks a random exponent d and random messages $M_1, \ldots, M_c \in \mathbb{Z}_N$. We have $c = (n/m)\log(2n)$ as in Fact 2.4. It then computes erroneous signatures $\tilde{S}_i$ of the M_i by using the exponentiation algorithm (Algorithm 2.3) to compute $M_i^d \bmod N$ and deliberately simulating a bit fault according to Fault Model 1.7 at a random iteration. Let $(M_i, \tilde{S}_i)_{i=1}^c$ be the resulting set of faulty signatures. We show there is a polynomial time (in n and 2^m) algorithm that given this data succeeds in factoring N with probability at least $1/n^\delta$.

Computing the Set W_d. Suppose the attack algorithm was given $(M_i, \tilde{S}_i)_{i=1}^{c}$ as input. By assumption, with probability at least $1/n^{\delta}$, at some point during the algorithm a signature $\tilde{S}_v$ will incorrectly cause the wrong candidate x' to be accepted in Step 12. That is, $\tilde{S}_v \pm 2^b M_v^w \equiv S_v \bmod N$ even though $\tilde{S}_v$ was generated by a different fault (here w is defined as in Step 9 using the bits of x'). We know that $\tilde{S}_v \equiv S_v \pm 2^{b_1} M_v^{w_1} \bmod N$ for some b_1, w_1. The pair b_1, w_1 corresponds to the actual location of the fault during the computation of $\tilde{S}_v$.

Now we claim that we have $w \neq w_1$. We know that any bit pattern $x = (d_{n-1}, d_{n-2}, \ldots, d_s, x_{s-1}, x_{s-2}, \ldots, x_{s-r})$, which satisfies the verification step, ends in the digit 1. Since the leading digit is $d_{n-1} = 1$, no digits on either end vanish by not contributing to the value w. The value w_1 is computed as $w_1 = \sum_{j=i}^{n-1} d_j 2^j$. If $i > s-r$, we know that $w \neq w_1$, since the addend $2^{s-r} x_{s-r}$ with $x_{s-r} = 1$ is not added to w_1. This cannot be corrected by choosing any different pattern for the bits $(x_{s-1}, x_{s-2}, \ldots, x_{s-r+1})$ due to the nature of the binary expansion. If $i = s - r$, we have $w = w_1$ iff x represents the correct bit pattern of d, because both are of the same length. If $i < s - r$, we may have $w = w_1$ if all bits $d_j = 0$ for $i \leq j < s - r$. However, we only claim that x represents the correct bit pattern for the $n - s + r$ most significant bits, we do not recover any additional bits. Hence, we do not need to consider other bits as long as our guessed pattern is correct for the corresponding subpattern of d. If any of the bits $d_j = 1$ for $i \leq j < s - r$, we clearly have $w \neq w_1$. Hence it is true that if x' is a wrong bit pattern, w and w_1 are bound to be different.

As explained before, $w \neq w_1$ implies that we have

$$S \pm 2^{b_1} M^{w_1} \equiv S \pm 2^b M^w \bmod N.$$

Rearranging terms we get $M^{w-w_1} \equiv \pm 2^{b_1 - b} \bmod N$. In other words, M must be a root of a polynomial of the form

$$t^{w-w_1} \equiv a \bmod N$$

for some known constant $a = \pm 2^{b_1 - b}$. We now define the set W_d to be the set of all polynomials of the form $p(t) = t^{w-w_1} \pm 2^{b_1 - b} \bmod N$, which may occur during an attack. We ensure positive exponents, hence, if $w < w_1$, we consider the equation $t^{w_1 - w} \equiv \pm 2^{b - b_1} \bmod N$.

We wish to apply Lemma 2.13 to show that Part 2 of this lemma must hold if Part 1 is false. First, we need to bound the number of polynomials $p(t) = t^{\omega} - a \bmod N$, where $\omega = w - w_1$ and $a = \pm 2^{b_1 - b}$. The value of w_1 is essentially the prefix of d from the most significant bit to the fault location. We have at most $n + 1$ prefixes of a bit sequence of length n, however, at most n apply in our case, since the main loop in the right-to-left repeated squaring algorithm, Algorithm 2.3, is executed only n times, hence, only n different intermediate values y can be affected by a fault. The values of w are the ones tested in Step 12. There are at most $n \cdot (2^{m+1} - 2) < 2n \cdot 2^m$ possible values, since w comprises of known upper bits of d and at most m guessed bits. Hence, there are $2n^2 2^m$ possible values for $\omega = w - w_1$. For the values b and b_1, there are at most $2n$ possibilities. Hence, a can take at most $2n^2$ many values, since the signs of $\pm 2^b$ and $\pm 2^{b_1}$ may either be the same or not. Hence, there are at most $4n^4 2^m$ many different polynomials $p(t)$. Since every polynomial can be computed in polynomial time, the set W_d can be constructed in polynomial time.

Factoring N. By assumption, Part 1 of the lemma is false. Hence, with probability at least $1/n^{\delta}$ (over the choice of d and M) we have that a randomly chosen value $M \in \mathbb{Z}_N$ satisfies any of the polynomials $p(t) = t^{\omega} - a \bmod N$. Note that ω depends on the choice of d. We have shown that for any d, there are at most $4n^4 2^m$ many polynomials $p(t) = t^{\omega} - a \bmod N$ in W_d, and W_d can be computed in polynomial time given d. For $\tau = m+5$, we guarantee $\#W_d < n^{\tau}$. Clearly, we have $\omega \leq 2N$, since both w and w_1 satisfy $2^{n-1} \leq w, w_1 \leq 2^n - 1$, hence, $\omega = w - w_1 < 2^{n-1} < N$. Hence, the condition $\omega \leq 2N$ is satisfied. Since we have shown that acceptance of a wrong candidate implies that at least one of the polynomials $p(t)$ is satisfied, and we assume that this holds with probability at least $1/n^{\delta}$, we may use Lemma 2.13 now. Since Part 2 of Lemma 2.13 holds, we can factor N in expected polynomial time. This proves Part 2 of this lemma. □

The two lemmas 2.10 and 2.14 allow to state the following theorem, proving that the corrected attack algorithm has a high probability to recover the secret key d. Therefore, the flaw in [BDL01] can easily be fixed. The important contribution of [BDL01], starting off research in fault attacks, remains valid.

Theorem 2.15 (Corrected Attack on RSA by Boneh, DeMillo, and Lipton). *Let $N = p \cdot q$ be an n-bit RSA modulus. For any $1 \leq m \leq n$, given $(n/m)\log(2n)$ pairs $(M_i, \tilde{S}_i)$, the secret exponent d can be extracted from Algorithm 2.3 with probability at least $1/2$. The probability is over the location of the register faults and random messages $m_i \in \mathbb{Z}_N$. The algorithm's running time is dominated by the time it takes to perform $O((2^m n^3 \log(n))/m)$ full modular exponentiations mod N.*

Proof:

With probability 1/2 guaranteed by Fact 2.4, every contiguous m-bit interval is hit by a fault at least once. Lemma 2.10 guarantees that in each round of the main loop (Lines 4–17), either a candidate pattern is found or one bit is correctly recovered. Lemma 2.14 guarantees that with a negligible loss in the success probability, any candidate pattern found represents the correct bit pattern, thus revealing at least one more bit of d. Therefore, the main loop is executed at most $n = \lfloor \log(N) \rfloor + 1$ times. In every round, at most $2^{m+1} - 2$ different bit patterns x are tested together with at most $2n$ different errors and at most $c = (n/m)\log(2n)$ different pairs $(M_v, \tilde{S}_v) \in \mathcal{S}$. Each test requires two modular exponentiations, one with the value w and one with the public key e. Hence, we have a total of $n \cdot (2^{m+1} - 1) \cdot 2n \cdot (n/m)\log(2n) \cdot 2 = O(2^m n^3 \log(n)/m)$ full modular exponentiations. The generation of the set $\mathcal{S}$ requires $(n/m)\log(2n)$ modular exponentiations with the ability to induce a fault, which vanishes in the O-notation. □

Our analysis yields an asymptotic running time of $O((2^m n^3 \log(n))/m)$ modular exponentiations, while the original analysis presented as Theorem 2.6 yields an asymptotic running time of $O((2^m n^3 \log^2(n))/m^2)$ modular exponentiations. This implies that our analysis yields a better asymptotic running time than Theorem 2.6 if $(\log(n)/m)^2 > \log(n)/m$, i.e., if $\log(n) > m$, which is true for all practical choices of parameters. We achieve a better result due to a more careful analysis: there are c different faulty results $\tilde{S}$, each of which may be used to reveal a part of d. However, at most n of them may yield different sub-patterns of d. If two faulty results are generated by a fault attack during the same iteration, but with a different error, one of them will never be used to recover a part of d. Obviously, if the first one has been used, the second one cannot yield additional information about d. Hence, we can replace one factor of $c = n \cdot (\log(2n)/m)$ by n. This explains the improved result.

2.2. An Attack on The Left-To-Right Repeated Squaring Algorithm

The two versions of repeated squaring presented as Algorithms 2.2 and 2.3 look similar, however, there are important differences. The right-to-left version needs an additional variable z which computes the appropriate binary powers M^{2^i} of the base M. The number of operations is identical, yet the first squaring in the left-to-right version is a squaring of 1, while the last squaring in the right-to-left version is a squaring of $M^{l(d)-1}$. Both are not needed to compute the correct result, and it may be interpreted as a slight advantage in runtime for the left-to-right version. However, both could be avoided using more lines of code. Nevertheless, the left-to-right version is usually the method of choice, preferred over its twin.

Surprisingly, there are attacks which do not apply to both versions alike. The most popular example is [FV03], presenting a differential power attack which searches for collisions in the power profile of the computations of M^d and $(M^2)^d$. Note that [FV03] describe the attack in the context of scalar multiplication for elliptic curves. This so-called *doubling attack* only applies to the left-to-right version and not to the right-to-left version, prompting the authors to state that "upwards is better than downwards". Contrary to that belief, we will present an attack with a similar selective applicability in Chapter 7, which indicates the converse of the claim from [FV03], i.e., where downwards is better than upwards.

The attack from [FV03] shows that it is not self-evident that an attack which applies to one of the two repeated squaring versions also applies to the other one. Boneh, DeMillo, and Lipton do not comment on how their attack on the right-to-left version carries over to the left-to-right version of repeated squaring. We are able to show that the attack from [BDL01] can also be mounted on the left-to-right repeated doubling algorithm, and we will present this result in the following. We will prove the applicability of our attack for the case, when -1 is a quadratic non-residue modulo N only. We will discuss the details of this choice later. It is possible to recover all bits of d either starting from the MSBs (Section 2.2.1) or from the LSBs (Section 2.2.2). Since both approaches are different, we present both in the next two sections. To the best of our knowledge, these results have not been published before.

We first present a slightly modified version of the standard left-to-right repeated squaring Algorithm 2.2. To avoid ambiguity when referring to intermediate variables, we change the variable names to indexed variables. This is the only modification we make.

Algorithm 2.16: Left-To-Right Repeated Squaring (indexed version)

Input: a message $M \in \mathbb{Z}_N$, a secret RSA key $1 \leq d$, where $l(d)$ denotes the number of bits of d, and an RSA modulus N
Output: $M^d \bmod N$
\# **init**
1 Set $y_{l(d)} := 1$
\# **main**
2 For i from $l(d) - 1$ downto 0 do
3 Set $y'_i := y^2_{i+1} \bmod N$
4 If $d_i = 1$ then set $y_i := y'_i \cdot M \bmod N$ else set $y_i := y'_i$
5 **Output** $S := y_0$

For both attacks, starting either from the MSBs or from the LSBs, we will use the same fault model as the original attack, namely the Single Bit Fault Model 1.7. In both cases, we assume that we target the intermediate variable y'_i in Line 4 of Algorithm 2.16. The fault type used in

the Single Bit Fault Model is that of bit flip faults following Definition 1.3, which implies that we can write the effect of the fault as $y'_i \nmapsto y'_i \pm 2^b$, for some random $0 \leq b < l(y'_i)$. For simplicity, we assume that $l(y'_i) = l(N)$, i.e., even if y'_i is a number with less digits than the modulus, we assume that leading zeros can be flipped to 1 as well. If this is impossible in practice, our algorithm tests too many candidates for b, which is a negligible overhead. For the analysis of our attack, we will use the following definitions.

Definition 2.17 (High and Low Parts of an Integer).
Let $d > 0$ be an n-bit integer with binary expansion $d = (d_{n-1}, d_{n-2}, \ldots, d_0)$. As $d \geq 1$, we have $d_{n-1} = 1$. We define the high part $h_i(d)$ *and the* low part $l_i(d)$ *of d, build by the $n - i$ most significant and $i + 1$ least significant bits of d respectively, as*

$$l_i(d) := \sum_{j=0}^{i} 2^j d_j \qquad \text{and} \qquad h_i(d) := \sum_{j=i}^{n-1} 2^{j-i} d_j \tag{2.22}$$

$$\Rightarrow \quad d = 2^{i+1} h_{i+1}(d) + l_i(d) \qquad \forall\ i \in \mathbb{Z}. \tag{2.23}$$

We will now derive some basic equations to describe the values of the intermediate variables used in Algorithm 2.16 in greater detail.

Proposition 2.18 (Properties of the Variables Used in Algorithm 2.16).
Let $S := M^d \bmod N$ and let $\tilde{S}$ denote a faulty output of Algorithm 2.16 resulting from a fault induced into the intermediate variable y'_i for some $0 \leq i \leq n-1$. Let $n = l(d)$ be the length of the secret key $d \geq 1$, and let e_i denote the error induced into the variable y'_i. In this case, we have the following equations for some of the variables computed by Algorithm 2.16:

$$y'_i \equiv (((M^{2d_{n-1}} \cdot M^{d_{n-2}})^2 \cdot M^{d_{n-3}})^2 \cdot \ldots \cdot M^{d_{i+1}})^2 \equiv M^{2h_{i+1}(d)} \bmod N \tag{2.24}$$

(the intermediate value y'_i from Algorithm 2.16 which is attacked)

$$y_i \equiv (((M^{2d_{n-1}} \cdot M^{d_{n-2}})^2 \cdot M^{d_{n-3}})^2 \cdot \ldots \cdot M^{d_{i+1}})^2 \cdot M^{d_i} \equiv y'_i \cdot M^{d_i} \equiv M^{h_i(d)} \bmod N$$

(the intermediate value y_i from Algorithm 2.16 which is the result of round i)

$$S \equiv y_i'^{2^i} \cdot M^{l_i(d)} \equiv \left(M^{2h_{i+1}(d)}\right)^{2^i} \cdot M^{l_i(d)} \equiv M^{2^{i+1}h_{i+1}(d) + l_i(d)} \equiv M^d \bmod N \tag{2.25}$$

(the correct final result of Algorithm 2.16)

$$\tilde{S} \equiv (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N \tag{2.26}$$

(the faulty final result of Algorithm 2.16)

Proof:

The formulas are easily derived from Algorithm 2.16. It is

$$y_i = M^{\sum_{j=i}^{n-1} 2^{j-i} d_j},$$

as y_i always contains the result $M^{h_i(d)}$, since $h_i(d) = (d_{n-1}, d_{n-2}, \ldots, d_i)$, i.e., the high part of d. This value is squared in the next iteration to yield $y'_{i-1} = y_i^2$. Given y_i and y'_i, Equations (2.25) and (2.26) follow directly using Equation (2.23). □

In the following, we will show that it is possible to recover all bits of d either starting from the MSBs or from the LSBs. The MSB approach is very similar to the original attack described in [BDL01]. However, the LSB version requires some additional work.

For both attacks, we assume that the RSA modulus N has been chosen to guarantee that -1 is a quadratic non-residue modulo N. This means that there is no integer $a \in \mathbb{Z}_N$ such that $a^2 \equiv -1 \bmod N$. We will briefly discuss the possibility to generalize our attack to arbitrary moduli N in Section 2.3. The restriction to these special moduli is practical. For example, the property that -1 is a quadratic non-residue holds for *Blum integers*, i.e., integers $N = p \cdot q$, such that both primes satisfy $p, q \equiv 3 \bmod 4$. If p and q are chosen as *strong primes*, i.e., primes p and q with the property that $(p-1)/2$ and $(q-1)/2$ are prime, N is guaranteed to be a Blum integer. Choosing strong primes offers security against certain factorization algorithms, e.g., Pollards $p-1$ factoring algorithm [MvOV96, § 3.2.3]. We will discuss in Section 3.3.1 that such strong primes can be found efficiently. Hence, restricting our analysis to this case is reasonable.

Given the fact that -1 is a quadratic non-residue modulo N, we can use the following fact about $2^{i\text{th}}$ congruences to ease our analyses.

Lemma 2.19 ($2^{i\text{th}}$ Congruence Factorization).
Let x and y be integers and let N be a composite integer such that -1 is a quadratic non-residue modulo N. If there is an integer $i \geq 1$, such that

$$x^{2^i} \equiv y^{2^i} \bmod N,$$

and $x \not\equiv \pm y \bmod N$, then there is a $0 \leq k < i$ such that $\gcd(x^{2^k} - y^{2^k}, N)$ yields a non-trivial factor of N.

Proof:

Assume that $x \not\equiv \pm y \bmod N$. Since $x^{2^i} \equiv y^{2^i} \bmod N$, there must be a minimal $k \geq 0$, such that $x^{2^{k+1}} \equiv y^{2^{k+1}} \bmod N$ and $x^{2^k} \not\equiv y^{2^k} \bmod N$. This yields two cases, one where $x^{2^k} \not\equiv \pm y^{2^k} \bmod N$, and a second where $x^{2^k} \equiv -y^{2^k} \bmod N$.

Case 1. Assume that we have $x^{2^k} \not\equiv \pm y^{2^k} \bmod N$. Let $a := x^{2^k}$ and $b := y^{2^k}$. We have $a \not\equiv \pm b \bmod N$ but $a^2 \equiv b^2 \bmod N$. The latter implies that $N \mid (a-b) \cdot (a+b)$. By assumption, both factors are not equal to zero and both factors are not multiples of N. Therefore, N must contain a non-trivial factor q, which divides $(a-b)$ but not $(a+b)$. This factor can be computed by $\gcd(a-b, N) = \gcd(x^{2^k} - y^{2^k}, N)$.

Case 2. Assume that we have $x^{2^k} \equiv -y^{2^k} \bmod N$. This implies that $(x/y)^{2^k} \equiv -1 \bmod N$. The power k must be larger than 0, since we assume that $x \not\equiv \pm y$. Therefore, a square root of -1 is given by $(x/y)^{2^{k-1}}$. However, we assume that -1 is a quadratic non-residue modulo N. Hence, Case 2 is impossible. □

2.2.1. Fault Attack Starting From the MSBs

In order to recover the bits of the secret key d starting from the MSBs, we need to derive a description of a faulty final result $\tilde{S}$, which depends only on the most significant bits of d. This will be the basis for our recovery. We state such an equation in the following lemma. In the remainder of this section, we will denote errors induced by a fault by e_i, where i denotes the affected iteration, and guessed values for the "correct error" e_i by ε.

Lemma 2.20 (Description of Faulty Results in Terms of Most Significant Bits). *Let $M \in \mathbb{Z}_N$ be a message and $\tilde{S}$ a faulty final result, which was returned by Algorithm 2.16, while a fault affected the intermediate variable y'_i in Line 3. Let $0 \leq i \leq l(d) - 1$ denote the affected iteration. We assume that the attack resulted in $y'_i \not\mapsto y'_i + e_i$, where e_i denotes the error. Let $h_i(d)$ be defined as in Definition 2.17.*

In this case, $\tilde{S}$ can be described in terms of $S = M^d \bmod N$, M, and the $n - i - 1$ most significant bits of d as

$$\tilde{S} \equiv S \cdot \left(1 + e_i \cdot M^{-2h_{i+1}(d)}\right)^{2^i} \bmod N. \tag{2.28}$$

If the RSA public key e is given, $\tilde{S}^e$ can be described in terms of M, e, and the $n - i - 1$ most significant bits of d as

$$\tilde{S}^e \equiv M \cdot \left(1 + e_i \cdot M^{-2h_{i+1}(d)}\right)^{2^i e} \bmod N. \tag{2.29}$$

Proof:

According to Proposition 2.18, we have

$$\frac{\tilde{S}}{S} \equiv \frac{(y'_i + e_i)^{2^i} \cdot M^{l_i(d)}}{(y'_i)^{2^i} \cdot M^{l_i(d)}} \equiv \left(\frac{M^{2h_{i+1}(d)} + e_i}{M^{2h_{i+1}(d)}}\right)^{2^i} \equiv \left(1 + e_i \cdot M^{-2h_{i+1}(d)}\right)^{2^i} \bmod N.$$

□

Equation (2.29) shows that knowledge of a correct signature is not needed to identify $\tilde{S}$, however, if it is given, Equation (2.28) can be used, which allows much more efficient computations.

Both, Equation (2.28) and Equation (2.29), use multiplicative inverses modulo N. If any of these multiplicative inverses should not exist, N can be factored. In this case, our attack succeeds right away, and all secrets can be computed. Since we assume that factoring N is a hard problem, we expect such an event to occur with negligible probability. Therefore, we will neglect this possibility and assume that all needed multiplicative inverses exist.

Given Equations (2.28) and (2.29), an adversary can guess a bit pattern $y = (y_{n-i-2}, \ldots, y_0)$ of length $n - i - 1$ and an error term ε_i in order to describe $\tilde{S}$. If y correctly represents the $n - i - 1$ most significant bits $(d_{n-1}, d_{n-2}, \ldots, d_{i+1})$ of d and ε_i is equal to the correct error term e_i, Equations (2.28) and (2.29) are satisfied using the guessed pair (y, ε_i). If the adversary guesses a false pair (y, ε_i), where y does not represent $(d_{n-1}, d_{n-2}, \ldots, d_{i+1})$, we will show that Equations (2.28) and (2.29) are not satisfied with overwhelming probability. Hence, the two equations offer a way to check whether a guessed bit pattern y and a guessed error ε_i represent the correct values.

The two equations are the basis for the key recovery algorithm, which works in the same manner as introduced by Boneh, DeMillo, and Lipton in [BDL01]. However, we cannot completely rule out the possibility that false patterns satisfy Equations (2.28) and (2.29). If this happens, a false guess would be identified as the correct pattern. However, we will show that such *false positives* do not prevent to recover the secret key in expected polynomial time.

We will now state the complete attack algorithm.

Algorithm 2.21: Attack on Left-To-Right Repeated Squaring, MSB Version

Input: Access to Algorithm 2.16, n the maximal length of the secret key d, e the RSA public key, m a parameter for acceptable amount of offline work.

Output: d with probability at least 1/2.

\# **Phase 1:** Collect Faulty Outputs

1 Set $c := (n/m) \cdot \log(2n)$

2 Create c faulty outputs of Algorithm 2.16 on inputs M_j by inducing faults according to Fault Model 1.7 in the intermediate variable y'_i in random iterations.

3 Collect the set $\mathcal{S} = \{(M_j, \tilde{S}_j) |\ M_j$ an input to Algorithm 2.16 and $\tilde{S}_j$ a faulty output$\}$.

\# **Phase 2:** Inductive Retrieval of Secret Key Bits

4 Set $s := 0$ representing the current candidate bit length between $s \cdot m + 1$ and $(s+1) \cdot m$

5 Set $K := \{\epsilon\}$ to contain candidates for verified/"known" MSB-parts of d

6 While $(s \cdot m < n - 1)$ do

7 for all patterns $y = (y_{u-1}, y_{u-2}, \ldots, y_0) \in K$ with length $u := l(y)$ do

\# The algorithm ensures that $(s-1) \cdot m < u \leq s \cdot m$

8 Set $K' = \emptyset$ # to contain candidates of length between $s \cdot m + 1$ and $(s+1) \cdot m$

9 For all r with $s \cdot m < u + r \leq (s+1) \cdot m$ do

10 For all test bit patterns $x = (x_{r-1}, x_{r-2}, \ldots x_0)$ of length $r < 2m$ do

\# Compute the test candidate w using the pattern $y \circ x$

11 Set $w := 2^r \sum_{j=0}^{u-1} y_j 2^j + \sum_{j=0}^{r-1} x_j 2^j$

\# Verification Step: Verify the test candidate using Equation (2.29)

12 For all $(M, \tilde{S}) \in \mathcal{S}$ do

13 For all b from 0 to $n-1$ do

14 if $\tilde{S}^e \equiv M \cdot \left(1 \pm 2^b \cdot M^{-2w}\right)^{2^{u+r-1}e} \bmod N$ then

15 Add $y \circ x = (y_{u-1}, y_{u-2}, \ldots, y_0, x_{r-1}, \ldots, x_0)$ to the set K'

16 Set $K := K'$ and $s := s + 1$ and continue at Line 6

\# **Phase 3:** Compute k from all candidates of length $u > n - 1 - m$

17 For all patterns $y = (y_{u-1}, y_{u-2}, \ldots, y_0) \in K$ with length $u := l(y)$ do

18 Set $r := n - 1 - u$

19 For all test bit patterns $x = (x_{r-1}, x_{r-2}, \ldots x_0)$ of length $r < m$ do

20 Set $w := 2^r \sum_{j=0}^{u-1} y_j 2^j + \sum_{j=0}^{r-1} x_j 2^j$

21 If $M^{w \cdot e} \equiv M \bmod N$ then **Output** $(y_{u-1}, y_{u-2}, \ldots, y_0, x_{r-1}, \ldots, x_0)$

22 **Output** FAILURE

The symbol ϵ used in Line 5 denotes the empty bit string of length 0.

Algorithm 2.21 consists of three phases. In the first phase, we collect up to $c = (n/m) \cdot \log(2n)$ faulty values $\tilde{S}$. In Phase 2, the algorithm tries to recover the secret key d starting from the most significant bits. Here, a "known" upper part $y = (y_{u-1}, y_{u-2}, \ldots, y_0)$ is assumed to represent the u most significant bits $(d_{n-1}, d_{n-2}, \ldots, d_{n-u})$ of d. These known upper bits will be combined with all possible r-bit extensions to form test bit patterns. Each test bit pattern is tested using

all possible faulty results from Phase 1, all possible error terms ε_i, and Equation (2.29). Since we assume the Single Bit Fault Model 1.7, we can enumerate all possible error values in polynomial time. If Equation (2.29) is satisfied for any combination, a new "known" upper part of d has been recovered.

However, a fundamental difference to the attack on right-to-left repeated squaring presented in the beginning of this chapter is the existence of false positives. Since we will not show that false positives only occur with negligible probability, we must take the existence of false positives into account. This implies that a "known" upper part may be a false guess. We solve this problem by collecting all known upper parts and by trying all of them to derive new longer patterns. Later, we will show that this strategy is feasible.

Phase 3 captures the special case when no error was induced into the last $r < m$ iterations. At this time, at most m bits are unknown, hence, they can be recovered by exhaustive search.

Remark. Algorithm 2.21 has been abbreviated for clarity in one minor detail. Since we always assume that $d > 0$, we know that $d_{n-1} = 1$. This implies that the algorithm should only test candidate patterns, which start with the digit 1. This consideration could be incorporated by starting the main loop of Lines 6–16 with setting $K := \{1\}$ in Line 5. However, this feature requires a more complicated description of the loop invariants, e.g., of $(s-1) \cdot m < u \leq s \cdot m$. Hence, omitting this consideration adds clarity. Nevertheless, the fact that $d_{n-1} = y_{u-1} = 1$ will be used in the analysis later.

Algorithm 2.21 shows the same basic pattern as Algorithm 2.9, which was used for an attack on the right-to-left repeated squaring algorithm. The main difference is the use of a set K to contain verified upper parts of d, allowing the occurrence of multiple verified candidate bit patterns.

Another difference to Algorithm 2.9 is the absence of any Zero Block Failure handling. Algorithm 2.21 has no problem equivalent to the Zero Block Problem as defined in Definition 2.8, because the value

$$w = 2^r \sum_{j=0}^{u-1} y_j 2^j + \sum_{j=0}^{r-1} x_j 2^j$$

used in Line 10 of Algorithm 2.21 is build from the left to the right with the most significant bit $y_{u-1} = 1$. A tailing zero bit $x_0 = 0$ does not vanish like in the case of Zero Block Failures, it shifts the leading bits by 1 to the left, thus changing the value of w. Hence, the pattern $y \circ x$ is a unique bit pattern, which uniquely encodes a positive integer w. Therefore, it is not possible that two different bit patterns yield the same integer w. This would only be possible if a pattern $y \circ x$ was allowed to contain leading, i.e., leftmost, zeros. Therefore, no problem similar to the Zero Block Failure introduced in Definition 2.8 may occur.

In the following, we will first investigate the occurrence of false positives in detail. This analysis will allow us to state results about the success probability and the running time of Algorithm 2.21. We start with defining valid guesses and formalizing the notion of false positives.

Definition 2.22 (Valid Guesses and False Positives in Algorithm 2.21).
Let $S = M^d \bmod N$ be an RSA signature and let $\tilde{S}$ be a faulty result as defined in Proposition 2.18. Let $y = (y_{u-1}, y_{u-2}, \ldots, y_0)$ be a bit pattern of length u with $y_{u-1} = 1$. This pattern yields a value w as used in Algorithm 2.21 according to the following formula:

$$w := \sum_{j=0}^{u-1} y_j 2^j.$$

Let $\varepsilon = \pm 2^b$, $0 \leq b < n$, be a guessed value for the error $e(y'_{n-u-1})$. If the verification step of Algorithm 2.21 yields

$$\frac{\tilde{S}^e}{M} \equiv \left(1 + \varepsilon \cdot M^{-2w}\right)^{2^{n-u-1}e},$$

then (y, ε) is called a valid guess *for the bit pattern $(d_{n-1}, d_{n-2}, \ldots, d_{n-u})$ and the correct error $e(y'_{n-u-1})$. If for at least one $n - 2 \geq j \geq n - u$ it holds that $d_j \neq y_{j-n+u}$, the valid guess (y, ε) is called a* false positive.

A false positive satisfies the verification step in Line 14 of Algorithm 2.21, yet it represents a false bit pattern. A false positive only requires the bit pattern y to be wrong. We do not consider the correctness of the error value ε. This is due to the fact that we do not need to learn the correct error value e_i for ε in order to recover the bit pattern of the secret RSA key d.

We will now show that during the execution of Algorithm 2.21, the number of all bit patterns, which ever occur in the round sets K', does not exceed a certain bound with overwhelming probability.

Lemma 2.23 (Bounding the Number of Special Valid Guesses).
Let K_s be the round set K computed in Line 16 at iteration s of the main loop of Algorithm 2.21 (Lines 6–16). Let -1 be a quadratic non-residue modulo N, $c = (n/m)\log(2n)$, and let

$$\kappa := \sum_j \#K_j.$$

If $\kappa > 2nc$, then at least one of the following cases holds:

1. *There are two valid guesses (x, ε_x) and (y, ε_y) with $x \neq y$ such that*

$$\left(1 + \varepsilon_x \cdot M^{-2w_x}\right) \equiv \left(1 + \varepsilon_y \cdot M^{-2w_y}\right) \bmod N$$

$$\text{where} \quad w_x := \sum_{j=0}^{l(x)-1} x_j 2^j \quad \text{and} \quad w_y := \sum_{j=0}^{l(y)-1} y_j 2^j.$$

2. *There are two valid guesses (x, ε_x) and (y, ε_y) with $x \neq y$ such that*

$$\gcd\left(\left(1 + \varepsilon_x \cdot M^{-2w_x}\right)^{2^k} - \left(1 + \varepsilon_y \cdot M^{-2w_y}\right)^{2^k}, N\right) \in \{p, q\},$$

for a $0 \leq k \leq n - u - 1$ with $u = l(x) = l(y)$, and w_x and w_y as defined in Case 1.

Proof:

We assume that $\kappa \geq 2nc + 1$. This implies that there are at least $2nc + 1$ different bit patterns y in the union of all K_s. This is true since elements from different round sets K_s have a different bit length, hence, there cannot be one pattern in multiple round sets. For every such y, there exists an ε_y, such that (y, ε_y) is a valid guess, which has been generated and tested by Algorithm 2.21 in Line 14. In this case, we can use the pigeon hole principle to show that there must be at least one pair $(M, \tilde{S}) \in \mathcal{S}$ such that $2n + 1$ of these valid guesses were verified using $(M, \tilde{S})$. We now fix this pair $(M, \tilde{S})$.

Since we have at least $2n + 1$ many valid guesses, but only n possible values for the bit lengths of the bit patterns y, we have at least three valid guesses with the same bit length u. We denote these three valid guesses by (x, ε_x), (y, ε_y), and (z, ε_z). We have $l(x) = l(y) = l(z) = u$, and we denote the bits of all three patterns starting with the index $u - 1$ through 0. In this case, Definition 2.22 shows that we have

$$\begin{aligned}\frac{\tilde{S}^e}{M} &\equiv \left(1 + \varepsilon_x \cdot M^{-2w_x}\right)^{2^{n-u-1}e} \\ &\equiv \left(1 + \varepsilon_y \cdot M^{-2w_y}\right)^{2^{n-u-1}e} \\ &\equiv \left(1 + \varepsilon_z \cdot M^{-2w_z}\right)^{2^{n-u-1}e} \bmod N.\end{aligned}$$

Since $\gcd(e, \varphi(N)) = 1$, exponentiation with e is a bijection, hence, we have

$$\left(1 + \varepsilon_x \cdot M^{-2w_x}\right)^{2^{n-u-1}} \equiv \left(1 + \varepsilon_y \cdot M^{-2w_y}\right)^{2^{n-u-1}} \equiv \left(1 + \varepsilon_z \cdot M^{-2w_z}\right)^{2^{n-u-1}}. \quad (2.32)$$

Equation (2.32) allows to factor N unless we have

$$\left(1 + \varepsilon_x \cdot M^{-2w_x}\right) \equiv \pm\left(1 + \varepsilon_y \cdot M^{-2w_y}\right) \equiv \pm\left(1 + \varepsilon_z \cdot M^{-2w_z}\right), \quad (2.33)$$

according to Lemma 2.19. Note that we assume that -1 is a quadratic non-residue modulo N, hence, Lemma 2.19 can be applied. This shows that either Equation (2.33) or Case 2 of Lemma 2.23 holds, i.e., N can be factored with the claimed gcd computation. Hence, we now assume that Equation (2.33) holds. In this case, at least two of the three valid guesses share the same sign. Hence, without loss of generality, we have

$$\left(1 + \varepsilon_x \cdot M^{-2w_x}\right) \equiv \left(1 + \varepsilon_y \cdot M^{-2w_y}\right) \bmod N.$$

□

The bound on κ established by Lemma 2.23 allows us to bound the overall running time of Algorithm 2.21. The proof follows substantially the proof of Lemma 2.14, based on [BDL01].

Theorem 2.24 (Probability of False Positives).
Let $\delta > 0$ be a fixed constant. Let N be an n-bit RSA modulus $N = p \cdot q$, and let $c = (n/m) \cdot \log(2n)$, where 2^{2m} is an acceptable amount of offline work. In this case, at least one of the following claims holds:

1. *The probability that Algorithm 2.21 encounters two valid guesses (x, ε_x) and (y, ε_y) with $x \neq y$ defined as in Definition 2.22, which satisfy*

$$\left(1 + \varepsilon_x \cdot M^{-2w_x}\right) \quad \equiv \quad \left(1 + \varepsilon_y \cdot M^{-2w_y}\right) \bmod N$$

$$\textit{where} \qquad w_x := \sum_{j=0}^{l(x)-1} x_j 2^j \qquad \textit{and} \qquad w_y := \sum_{j=0}^{l(y)-1} y_j 2^j$$

is less than $1/n^\delta$. The probability is over the random choices of messages $M_i \in \mathbb{Z}_N$ given to the attack algorithm and the random choice of the decryption exponent d.

2. *A non-trivial factor of N can be computed in expected polynomial time (in n and 2^{2m}).*

Proof:

The proof of this theorem is very similar to the proof of Lemma 2.14. As before, we prove the existence of an algorithm that factors all RSA moduli N for which Part 1 is false. The algorithm works as follows: it picks a random exponent d and random messages $M_1, \ldots, M_c \in \mathbb{Z}_N$. We have $c = (n/m)\log(2n)$ as in Fact 2.4. It then computes erroneous signatures $\tilde{S}_i$ of the M_i by using Algorithm 2.16 to compute $M_i^d \bmod N$ and deliberately simulating a bit fault according to Fault Model 1.7 at a random iteration. Let $(M_i, \tilde{S}_i)_{i=1}^c$ be the resulting set of faulty signatures. We show there is a polynomial time (in n and 2^{2m}) algorithm that given this data succeeds in factoring N with high probability.

Computing the Set W_d. We assume that Case 1 is false. In this case, Algorithm 2.21 encounters two valid guesses (x, ε_x) and (y, ε_y) with $x \neq y$, and w_x and w_y as defined above with probability at least $1/n^\delta$, such that

$$\left(1 + \varepsilon_x \cdot M^{-2w_x}\right) \equiv \left(1 + \varepsilon_y \cdot M^{-2w_y}\right) \bmod N \tag{2.34}$$

$$\Rightarrow \qquad M^{2w_y - 2w_x} \equiv \frac{\varepsilon_y}{\varepsilon_x} \bmod N. \tag{2.35}$$

Obviously, Equation (2.35) resembles the case already analyzed for the original attack, presented in Lemma 2.14. Similar to the proof of Lemma 2.14, we want to use Lemma 2.13 to show that N can be factored. Therefore, we construct all polynomials $p(t) = t^\omega - a \bmod N$, which occur for a given secret exponent d. Each polynomial has $\omega = 2(w_y - w_x)$ and $a = \varepsilon_y/\varepsilon_x$. Hence, each polynomial can be constructed in polynomial time. Let W_d be the set of all possible polynomials $p(t)$ constructed as described above. We ensure positive exponents, hence, if $w_y < w_x$, we consider $p(t) = t^{2(w_x - w_y)} - \varepsilon_x/\varepsilon_y \bmod N$.

We bound the number of such polynomials $p(t)$ by the following consideration. We have $a = \pm 2^{b_y - b_x}$, and there are $2n$ values for ε_x and $2n$ values for ε_y. Since the signs may either match or be different, this yields at most $2n^2$ possible combinations for a. For $\omega = 2w_y - 2w_x$, we need to bound the number of possible different values w_x and w_y. We assume that at some point, we get the desired pair of valid guesses (x, ε_x) and (y, ε_y) satisfying Equation (2.34). We know from Lemma 2.23, that there are at most $2nc$ many valid guesses which do not ensure that at least one polynomial $p(t)$ is satisfied. Hence, there are at most $2nc$ many bit patterns y in any K. Therefore, we may have to test at most $2nc \cdot 2^{2m}$ many test patterns in each round, with a maximum of n/m rounds. Hence, there are at most $2n^2c2^{2m}/m \leq 2n^2c2^{2m}$ many different values for both w_x and w_y, hence, we get at most $4n^4c^2 2^{4m}$ many values for ω. Therefore, we have at most $8n^6c^2 2^{4m}$ many different polynomials in W_d. This can be bounded from above by n^τ with $\tau = 9$ for $n \gg m$ large enough, and especially for practical bit lengths of N. Since we know that $w_x - w_y < N$, we have $\omega \leq 2N$.

Factoring N. The above considerations show that we may use Lemma 2.13. Since we know that the probability that a random message M satisfies any of the polynomials $p(t)$ is at least $1/n^{\delta}$, we know that Part 2 of Lemma 2.13 applies. Hence, a non-trivial factor of N can be found in polynomial time, proving this theorem. □

Theorem 2.24 allows to conclude that Algorithm 2.21 will recover d in expected polynomial time.

Theorem 2.25 (Success and Running Time of Algorithm 2.21).
Algorithm 2.21 succeeds to recover the secret key d of bit length n in the time it takes to perform

$$O\left(\left(\frac{n^2\log(n)2^m}{m}\right)^2\right)$$

full modular exponentiations or full modular exponentiations with the ability to induce a fault with probability at least $1/2$.

Proof:

We will investigate the costs of all of the three phases of Algorithm 2.21 separately.

Phase 1. In the first phase, the total costs are fixed. Here, $c = (n/m)\log(2n)$ full modular exponentiations must be performed, each of which suffers a fault induced by the adversary.

Phase 2. For the second phase, we will count the maximal number of operations starting from the innermost loop.

In the verification step, i.e., in Line 14, Algorithm 2.21 has to perform one comparison of two precomputed values $\tilde{S}^e$ and $M \cdot (1 \pm 2^b \cdot M^{-2w})^{2^{u-r-1}e}$. Since the values $\tilde{S}^e$ are used in every iteration of the main loop (Lines 6–16), these values should be computed once in Phase 1, and they should not be computed repeatedly. Here, c full modular exponentiations are required, which may be counted towards Phase 1. Given w, b, the sign of 2^b, and the values e and $u-r-1$, the value $M \cdot (1 \pm 2^b \cdot M^{-2w})^{2^{u-r-1}e}$ can be computed by 2 modular exponentiations for 2^b and M^{-2w}, 1 multiplication and 1 addition or subtraction, the computation of $2^{u-r-1}e$, the final exponentiation with $2^{u-r-1}e$, and a last multiplication with M. The computation of $-2w$ from w is negligible. All these operations can surely be done in the time required to perform 6 full modular exponentiations. Compared to this, the cost of the comparison in Line 14 is negligible and will not be counted.

The verification step has to be performed for all choices of $\tilde{S}^e$ and all choices of $\pm 2^b$, hence, Lines 12–15 require at most $6 \cdot 2n \cdot c$ full modular exponentiations.

For a given pattern $y \circ x$, the computation of w requires $(u+r)$ additions and $2(u+r)+1$ multiplications, i.e., it can be computed in the time required to compute one full modular exponentiation. Note that w is computed over the integers. The loop from Line 9 to Line 15 has to be computed for all choices of $y \in K$. We will first derive the costs for an arbitrary but fixed choice of $y \in K$ with $u := l(y)$. It is clear from Algorithm 2.21 and a trivial application of induction that all $y \in K$ satisfy $(s-1) \cdot m < u \leq s \cdot m$. Algorithm 2.21 starts with $K = \{\epsilon\}$, i.e., $u = 0$. For any y, all bit patterns $y \circ x$, with $r := l(x)$, are tested, such that $s \cdot m < u + r \leq (s+1) \cdot m$. Hence, all y have a number of bits from a

certain interval of length m. This implies that the longest x, which might be tested, has length $2m-1$. This happens if $u = (s-1)\cdot m + 1$ and $u + r = (s+1)\cdot m$. Since we test all bit patterns of all sizes less or equal to $2m-1$, we need to test at most $2^{2m}-2$ different bit patterns for each choice of y. Therefore, the loop of Lines 9–15 requires at most $(2^{2m}-2)\cdot(1+12\cdot n\cdot c)$ full modular exponentiations.

The total number of times the loop of Lines 9–15 has to be executed depends on the overall number of y in all round sets K. Following the results of Lemma 2.23 and Theorem 2.24, we expect to have at most $2nc$ many bit patterns y with overwhelming probability, since N can be factored otherwise. Hence, Phase 2 requires at most

$$2nc\cdot(2^{2m}-2)\cdot(1+12\cdot n\cdot c) = O(n^2c^2 2^{2m}) = O\left(\left(\frac{n^2\log(n)2^m}{m}\right)^2\right)$$

full modular exponentiations.

Phase 3. Phase 3 requires to test at most 2^m different bit patterns x for all $y \in K$. Every pattern $y \circ x$ requires to compute w and $M^{w\cdot e}$. Then, the value $M^{w\cdot e}$ must be compared to M. All three tasks can be computed in the time it takes to perform two full modular exponentiations. Since the last K contains at most $2nc$ candidates for y, we have total costs of

$$O\left(2^m\cdot\frac{n^2}{m}\cdot\log(n)\right)$$

times the cost to perform a full modular exponentiation.

This implies that all 3 phases together can be performed in the time required to compute

$$O\left(\left(\frac{n^2\log(n)2^m}{m}\right)^2\right)$$

full modular exponentiations. Here, we assume that a modular exponentiation requires the same time as a modular exponentiation, where a fault is induced.

Correctness. With probability $1/2$, each block of m contiguous bits has been targeted by a fault attack according to Fact 2.4. Hence, one of the candidate patterns $y \circ x$ represents the correct bit pattern for at least one choice of $\tilde{S}$ and $\pm 2^b$. Hence, the correct bit pattern will always be a member of the round set K. Therefore, it will also be in the final round set K at the beginning of Phase 3. Since it correctly represents d, it will satisfy the verification step of Phase 3 in Line 20. Moreover, any bit pattern $y \circ x$, which yields a value w such that $M^{w\cdot e} \equiv M \bmod N$ yields the correct value for d, if it holds for random M.

Faster Recovery. Lemma 2.23 and Theorem 2.24 have shown that the attack algorithm can recover d much faster than normal if two bit patterns x and y as defined in Theorem 2.24, Case 1, occur. In this case, N can be factored efficiently, which allows to compute d easily. However, since the results from Lemma 2.23 and Theorem 2.24 indicate that the probability of this event is negligible, Algorithm 2.21 does not feature any mechanism to detect such a case. In the unlikely event that such a case occurs, we assume that Algorithm 2.21 fails. Hence, the success probability is negligibly below $1/2$. □

Theorem 2.25 shows that the recovery of d can be done in expected polynomial time for an attack on the left-to-right repeated squaring algorithm. However, the asymptotic running time is almost the square of the running time for the attack on the right-to-left repeated squaring algorithm (see Theorem 2.15).

In practice, Algorithm 2.21 behaves much better than anticipated by the result of Theorem 2.24. Numerous experiments never witnessed the occurrence of a false positive. Therefore, the result of Theorem 2.24 might not be tight. For practical purposes, we propose a simplified algorithm presented as Algorithm 2.26, which proved sufficient in practice.

Algorithm 2.26: Attack on Left-To-Right Repeated Squaring, simplified MSB version

Input: Access to Algorithm 2.16, n the maximal length of the secret key d, e the RSA public key, m a parameter for acceptable amount of offline work.

Output: d with probability at least 1/2.

\# **Phase 1:** Collect Faulty Outputs

Set $c := (n/m) \cdot \log(2n)$

Create c faulty outputs of Algorithm 2.16 on inputs M_j by inducing faults according to Fault Model 1.7 in the intermediate variable y'_i in random iterations.

Collect the set $\mathcal{S} = \{(M_j, \tilde{S}_j) |\ M_j$ an input to Algorithm 2.16 and $\tilde{S}_j$ a faulty output$\}$.

\# **Phase 2:** Inductive Retrieval of Secret Key Bits

Set $s := n - 1$ representing the index of the highest known bit of d, where $d_{n-1} = 1$.

While $(s > 0)$ do

\# Compute the known MSB part of the exponent d.

    Set $w_s := h_s(d)$

\# Try all possible bit patterns with length $r \leq m$.

    For all lengths $r = 1, 2, \ldots m$ do

        For all possible test bit patterns $x = (x_{s-1}, x_{s-2}, \ldots x_{s-r})$ of length r do

\# Compute the test candidate w

            Set $w := 2^r w_s + \sum_{j=s-r}^{s-1} x_j 2^{j-s+r}$

\# Verification Step: Verify the test candidate using Equation (2.29)

            for all $(M, \tilde{S}) \in \mathcal{S}$ do

                For all b from 0 to $n - 1$ do

                    if $\tilde{S}^e \equiv M \cdot \left(1 \pm 2^b \cdot M^{-2w}\right)^{2^{s-r-1}e} \mod N$ then

                        conclude that $d_j = x_j$ for all $s - r \leq j \leq s - 1$,

                        set $s := s - r$ and continue at Line 5

Output k

2.2.2. Fault Attack Starting From the LSBs

In this section, we also assume that an adversary attacks Algorithm 2.16, the left-to-right repeated squaring algorithm. However, this time we assume that contrary to the last section, the adversary wants to recover the bits of the secret key d starting from the least significant bit d_0. This direction of recovering the bits requires a more careful analysis than the MSB version presented in the last section. However, since the approach is very different from the MSB approach, and its asymptotic running time is faster than the running time of the MSB approach, it is of independent interest. As before, we base our analysis on the assumption that -1 is a quadratic non-residue modulo N.

As a basis for our attack, we assume that the adversary targets the intermediate variable y'_i after it has been computed in Line 3 of Algorithm 2.16 and before it is used in Line 4. This is the same setting as for the attack presented in the last section. We also assume the same fault model as before, namely the Single Bit Fault Model 1.7. According to Proposition 2.18, we have

$$\left.\begin{array}{rcl} y'_{i-1} & \stackrel{(2.24)}{=} & M^{2h_i(d)} \bmod N \\ y'_i & \stackrel{(2.24)}{=} & M^{2h_{i+1}(d)} \bmod N \end{array}\right\} \Rightarrow \quad y'_{i-1} \stackrel{(2.22)}{=} M^{2(2h_{i+1}(d)+d_i)} = y_i'^2 \cdot M^{2d_i}. \tag{2.37}$$

Equation (2.37) will be the basis for the recovery of the bits of d. This will be done by not only recovering all bits of d one after the other, but also by recovering all intermediate values of the variable y'_i. Equation (2.25), stating that

$$S = y_i'^{2^i} \cdot M^{l_i(d)} \equiv M^d \bmod N, \tag{2.25}$$

and Equation (2.37) allow to recover these values inductively. The first value y'_0 can be computed very easily. It is $y'_0 = S/M^{d_0} \bmod N$. In the RSA system, we have $d_0 = 1$, since $\gcd(d, \varphi(N)) = 1$ must hold and $\varphi(N)$ is even. Therefore, the first secret bit of d is d_1, and we know that $y'_0 = S/M \bmod N$ by Equation (2.25). We denote by n the maximal binary length $l(d)$ of d as in the previous sections. Since we assume that $d_{n-1} = 1$, we also know the values $y'_{n-1} = 1$ and $y'_{n-2} = M^2 \bmod N$. This follows directly from Algorithm 2.16. Hence, we do not have to compute these values during our attack. For other values of i, Equation (2.37) yields

$$y_i'^2 = y'_{i-1}/M^{2d_i} \bmod N. \tag{2.38}$$

In order to compute any of the remaining y'_i, it is necessary to compute the square roots of $y'_{i-1}/M^{2d_i} \bmod N$. Since N is a composite number with unknown factorization, this is a hard problem. However, given the faulty results $\tilde{S}$, collected by the adversary, these roots can be computed with high probability. Although we use divisions, which are not guaranteed to be applicable modulo N, we will assume that every multiplicative inverse modulo N exists. This is justified by the fact that N can be factored otherwise, which allows to compute d much faster.

Given a faulty signature $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$ according to Equation (2.26), the following lemma shows that y_i can be computed in polynomial time if $M^{l_i(d)}$, i, the error value e_i, and the value y'_{i-1} are known.

Lemma 2.27 (Computation of y'_i From a Faulty Signature).
Let $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$, for some $0 < i < n-1$, $e_i \in \mathbb{Z}_N$ an error term, and $l_i(d)$ defined as in Definition 2.17. We have

$$y'_i = \left(\frac{\tilde{S}}{M^{l_i(d)}} - B \right) \Big/ A \bmod N \tag{2.39}$$

$$\textit{where} \qquad A = \sum_{k=0}^{2^{i-1}-1} \binom{2^i}{2k+1} \cdot \left(y_i'^2\right)^k \cdot e_i^{2^i-(2k+1)} \tag{2.40}$$

$$\textit{and} \qquad B = \sum_{k=0}^{2^{i-1}} \binom{2^i}{2k} \cdot \left(y_i'^2\right)^k \cdot e_i^{2^i-2k}. \tag{2.41}$$

Equation (2.39) can be computed in the time required to perform 7 full modular exponentiations, if all the values $\tilde{S}$, M, $l_{i-1}(d)$, d_i, i, e_i, and y'_{i-1} are known.

Proof:

The values A and B are derived from the binomial coefficients of $(y'_i + e_i)^{2^i}$, where A contains all odd powers and B contains all even powers. In A, one y'_i has been divided off, leaving only even powers of y'_i.

Equation (2.39) follows directly from Equation (2.25), which states that

$$\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} = (Ay'_i + B) \cdot M^{l_i(d)} \bmod N.$$

If y'_{i-1} and d_i are known, we can compute $y'^2_i = y'_{i-1}/M^{2d_i} \bmod N$ according to Equation (2.37). In this case, A and B can be computed using the following variant of repeated squaring.

Algorithm 2.28: Computation of A and B

Input: The values y'^2_i, e_i, and the number i
Output: The values A and B
1 Set $A_0 := 1$ and $B_0 := e_i$
2 For k from 1 to i do
3 $\quad A_k := 2 \cdot A_{k-1} \cdot B_{k-1}$
4 $\quad B_k := A^2_{k-1} \cdot y'^2_i + B^2_{k-1}$
5 **Output** A_i, B_i

Algorithm 2.28 uses the fact that

$$\left(A_{k-1} \cdot y'_i + B_{k-1}\right)^2 = \left(2A_{k-1}B_{k-1} \cdot y'_i + (A^2_{k-1} \cdot y'^2_i + B^2_{k-1})\right),$$

and both $A_k := 2A_{k-1}B_{k-1}$ and $B_k := A^2_{k-1} \cdot y'^2_i + B^2_{k-1}$ are known values. Algorithm 2.28 requires 3 modular multiplications, 1 modular addition and 2 modular squarings (y'^2_i is known in advance) in each loop iteration. As i is bounded by $n = \lfloor \log(N) \rfloor + 1$, Algorithm 2.28 needs at most $6n$ modular multiplications, if the cost of a modular addition and of a modular squaring are assumed to be bounded by the cost of a modular multiplication. Since we assume that a full modular exponentiation requires up to $2n$ multiplications and squarings, the computation of A and B can be bounded by the cost to compute 3 full modular exponentiations.

Additionally, the value $M^{l_i(d)}$ accounts for another full modular exponentiation, while we may count the two divisions and the subtraction in (2.39) as an additional two exponentiations. The computation of $y'^2_i = y'_{i-1}/M^{2d_i}$ can be done in the time it takes to perform one full modular exponentiation. This yields a total of at most 7 full modular exponentiations. □

Getting All Parameters. Lemma 2.27 shows how y'_i can be computed if all the values $\tilde{S}$, M, $l_{i-1}(d)$, d_i, i, e_i, and y'_{i-1} are known. Since we do not know any of these values if $i > 0$, we will recover them using an inductive approach similar to the approaches presented in the last sections. For the recovery, we will compute the values y'_i, d_i, and e_i given the values $\tilde{S}$, M, $l_{i-1}(d)$, and y'_{i-1}. We will recover the parameters for all values of $0 \leq i < n$ starting with $i = 0$. As explained above, we know that $y'_0 = S/M^{d_0}$ and $d_0 = 1$ in the RSA scheme. Therefore, we can start seeking for the tuple (y'_1, d_1, e_1), which we need to identify in order to proceed to the next round $i = 2$.

Similar to the attacks in the previous sections, we will start by choosing a random message M and we will collect a set $\mathcal{S}$ of faulty signatures $\tilde{S}$. However, we assume that all faulty signatures have been created using the same input message M. We also assume that the faulty signatures have been created by a fault induced into y'_i at random iterations $0 \leq i < n$. Here, we assume that subsequent faults are i.i.d. according to the uniform distribution. We collect $c = (n/m) \cdot \log(2n)$ many faulty results. However, we will restrict our analysis to the choice $m = 1$. This guarantees that with probability at least $1/2$, every iteration $0 \leq i < n$ has been hit by a fault at least once (see Fact 2.4). We will always assume this "lucky case" in the following.

For the approach to recover all bits of d starting from the least significant bits, our inductive approach allows us to assume that at a specific step i all the values y'_{i-1}, $l_{i-1}(d)$ and i are known. Unfortunately, the value $l_i(d) = l_{i-1}(d) + 2^i d_i$ requires knowledge about d_i, which is the value, we try to recover in the first place. Additionally, given a faulty signature $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$, the error term e_i is not known either. This implies that Lemma 2.27 cannot be used to compute a unique value y'_i without further considerations.

However, there are at most $4n$ possible pairs $(d_i, e_i = \pm 2^b)$, since $0 \leq b \leq n-1$. Hence, we may try all possible pairs in the same manner as demonstrated for the attacks described in the previous sections. Differently from before, the verification step, which eliminates false guesses, is more complicated than, e.g., for the MSB approach described in the last section. We start with defining some notation.

Definition 2.29 (Solution Candidates and the Set of Guesses).
Let $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$ be a faulty signature. Given i, M, y'_{i-1}, and $l_{i-1}(d)$, we define the set of guesses

$$\mathcal{G}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S}) := \left\{ (\gamma, x, \varepsilon) \;\middle|\; \begin{array}{l} x \in \{0,1\} \text{ and } \varepsilon = \pm 2^b, 0 \leq b \leq n-1, \text{ and } \gamma \\ \text{is the value } y'_i \text{ computed in Lemma 2.27 on} \\ \text{inputs } \tilde{S},\ i,\ M,\ y'_{i-1},\ l_{i-1}(d), \text{ and the guessed} \\ \text{values } (x, \varepsilon) \text{ replacing } (d_i, e_i) \end{array} \right\}. \quad (2.43)$$

If the input values i, M, y'_{i-1}, and $l_{i-1}(d)$ are not important or clear from the context, we will abbreviate $\mathcal{G}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S})$ as $\mathcal{G}_i(\tilde{S})$. Any tuple $(\gamma, x, \varepsilon) \in \mathcal{G}_i(\tilde{S})$ is called a solution candidate *for the correct tuple (y'_i, d_i, e_i). Let*

$$\mathcal{S} = \left\{ \tilde{S} \;\middle|\; \tilde{S} = (y'_j + e_j)^{2^j} \cdot M^{l_j(d)} \bmod N \text{ for some } 0 \leq j \leq n-1,\ e_j = \pm 2^b,\ 0 \leq b \leq n-1 \right\}$$

be a set of faulty signatures. We define

$$\mathcal{G}_i(\mathcal{S}) := \bigcup_{\tilde{S} \in \mathcal{S}} \mathcal{G}_i(\tilde{S}). \quad (2.44)$$

Since there are $4n$ possible values for (x, ε), the set $\mathcal{G}_i(\tilde{S})$ contains up to $4n$ tuples for any $\tilde{S}$.

The Verification Step

The idea for the verification step is quite obvious. If we compute a candidate for y'_i using the faulty result $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)}$ with the correct value for i according to Equation (2.39), we may use the correct and known value y'_{i-1} to check if the computed value is a possible solution. We know from Equation (2.38), that the correct guess for y'_i and d_i must satisfy

$$y_i'^2 = y'_{i-1}/M^{2d_i} \bmod N. \tag{2.45}$$

Therefore, we may sort out all those guesses $(\gamma, x, \varepsilon) \in \mathcal{G}_i(\tilde{S})$, which do not satisfy Equation (2.45). This idea is formalized in the following definition.

Definition 2.30 (Test 1 and the Set of Guesses Surviving Test 1).
Let $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$ be a faulty signature. Let $\mathcal{G}_i(\tilde{S})$ be the set of guesses computed according to Lemma 2.27, given the known values $\tilde{S}$, M, i, $l_{i-1}(d)$, and y'_{i-1}, and all choices of $x \in \{0, 1\}$ and $\varepsilon = \pm 2^b$, $0 \leq b < n$.

Let $(\gamma, x, \varepsilon) \in \mathcal{G}_i(\tilde{S})$. We define the set of guesses surviving Test 1, *denoted by $\mathcal{G}^{(1)}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S})$, as the set of all tuples from the set $\mathcal{G}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S})$ satisfying*

$$\textit{Test 1:} \qquad \gamma^2 \equiv y'_{i-1}/M^{2x} \bmod N, \tag{2.46}$$

i.e.,

$$\mathcal{G}^{(1)}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S}) := \left\{ (\gamma, x, \varepsilon) \in \mathcal{G}_i(\tilde{S}) \;\middle|\; \gamma^2 \equiv y'_{i-1}/M^{2x} \bmod N \right\}. \tag{2.47}$$

If the input values i, M, y'_{i-1}, and $l_{i-1}(d)$ are not important or clear from the context, we will abbreviate $\mathcal{G}^{(1)}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S})$ as $\mathcal{G}^{(1)}_i(\tilde{S})$. For a set $\mathcal{S}$ of faulty signatures as defined in Definition 2.29, we define

$$\mathcal{G}^{(1)}_i(\mathcal{S}) := \bigcup_{\tilde{S} \in \mathcal{S}} \mathcal{G}^{(1)}_i(\tilde{S}). \tag{2.48}$$

We will show that the $4n$ values given by the set $\mathcal{G}_i(\tilde{S})$ can be reduced to a single solution candidate, at least with high probability. Once we can achieve this, an inductive approach similar to the attacks described in the previous sections, can be used for an attack. We will first show that Test 1 effectively reduces the number of remaining solution candidates. The properties of the set $\mathcal{G}^{(1)}_i(\tilde{S})$ are stated by the following lemma.

Lemma 2.31 (Eliminating False Solution Candidates: Test 1).
Let $\mathcal{G}^{(1)}_i(\tilde{S})$ be the set defined in Definition 2.30 for some faulty signature $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$ and the variable choices required by Definition 2.30. Then the set $\mathcal{G}^{(1)}_i(\tilde{S})$ satisfies the following claims:

1. *If $(\gamma, x, \varepsilon) \in \mathcal{G}^{(1)}_i(\tilde{S})$, then $(-\gamma, x, -\varepsilon) \in \mathcal{G}^{(1)}_i(\tilde{S})$ as well.*
2. *For $(\gamma, d_i, e_i) \in \mathcal{G}^{(1)}_i(\tilde{S})$, it holds that $\gamma = y'_i$. For tuples $(\gamma, x, \varepsilon) \notin \mathcal{G}^{(1)}_i(\tilde{S})$, it holds that $(\gamma, x, \varepsilon) \neq (y'_i, d_i, e_i)$.*

3. *We have* $\#\mathcal{G}_i^{(1)}(\tilde{S}) \geq 2$.

4. *For any two tuples* $(\gamma_1, x_1, \varepsilon_1)$, $(\gamma_2, x_2, \varepsilon_2) \in \mathcal{G}_i^{(1)}(\tilde{S})$, *one of the following cases holds:*

   a) *We have* $\gamma_1 \cdot M^{x_1} \equiv \pm\gamma_2 \cdot M^{x_2} \bmod N$.
   For any tuple $(\gamma, x, \varepsilon) \in \mathcal{G}_i^{(1)}(\tilde{S})$, *it holds that* $\gamma \equiv \pm y_i' \cdot M^{d_i - x} \bmod N$.

   b) *The value* $\gcd(\gamma_1 \cdot M^{x_1} - \gamma_2 \cdot M^{x_2}, N)$ *is a non-trivial factor of* N.

Proof:

Part 1. We first show that $(\gamma, x, \varepsilon) \in \mathcal{G}_i^{(1)}(\tilde{S})$ implies that $(-\gamma, x, -\varepsilon) \in \mathcal{G}_i^{(1)}(\tilde{S})$ as well. Suppose that $(\gamma, x, \varepsilon) \in \mathcal{G}_i^{(1)}(\tilde{S})$. Since all possible combinations of $x \in \{0, 1\}$ and $\varepsilon = \pm 2^b$, $0 \leq b \leq n-1$, are used to compute tuples, there is also a tuple $(\gamma', x, -\varepsilon)$ in the set of guesses $\mathcal{G}_i(\tilde{S})$. The value γ has been computed using x and ε according to Lemma 2.27 as

$$\gamma = \left(\frac{\tilde{S}}{M^{l_{i-1}(d)+x2^i}} - B \right) \Big/ A \bmod N$$

$$\text{where} \qquad A = \sum_{k=0}^{2^{i-1}-1} \binom{2^i}{2k+1} \cdot \left(\frac{y_{i-1}'}{M^{2x}} \right)^k \cdot \varepsilon^{2^i-(2k+1)} \tag{2.49}$$

$$\text{and} \qquad B = \sum_{k=0}^{2^{i-1}} \binom{2^i}{2k} \cdot \left(\frac{y_{i-1}'}{M^{2x}} \right)^k \cdot \varepsilon^{2^i-2k}. \tag{2.50}$$

Let A' and B' be the values computed using the same prerequisite values $\tilde{S}$, i, $l_{i-1}(d)$, y_{i-1}', M, and x, but the negated error value $-\varepsilon$. The error term ε is only used to compute the two values A and B. Since all occurrences of ε in Equation (2.49) are factors with odd powers, it is $A' = -A$. For B, as defined in Equation (2.50), all occurrences of ε are factors with even powers, hence, $B' = B$. This implies that $\gamma' = -\gamma$, hence, $(-\gamma, x, -\varepsilon) \in \mathcal{G}_i(\tilde{S})$ as well.

If the tuple (γ, x, ε) satisfies Test 1, then $(-\gamma, x, -\varepsilon)$ satisfies the test as well, since the right hand side of Equation (2.46) uses the same values in both cases, and the left hand side is squared, thus eliminating the sign of $-\gamma$. Therefore, both tuples satisfy Test 1.

Part 2. According to Lemma 2.27, y_i' can be computed given the correct parameters. Hence, (y_i', d_i, e_i) is in the set $\mathcal{G}_i(\tilde{S})$, if $\varepsilon = e_i$ is among the possible choices for ε. From Equation (2.37), we know that the correct tuple (y_i', d_i, e_i) satisfies Equation (2.46), therefore, pairs not satisfying this equation cannot yield the correct values.

Part 3: Lower Bound. For the lower bound on the number of tuples satisfying Test 1, consider the correct tuple (y_i', d_i, e_i). Since we know that Lemma 2.27 holds for the correct values, we know that $(y_i', d_i, e_i) \in \mathcal{G}_i(\tilde{S})$. We also know that the correct guess satisfies Test 1, hence, $(y_i', d_i, e_i) \in \mathcal{G}_i^{(1)}(\tilde{S})$. With the considerations in Part 1, we know that $(-y_i', d_i, -e_i) \in \mathcal{G}_i^{(1)}(\tilde{S})$ as well. Moreover, since $e_i \neq 0$, we have $(y_i', d_i, e_i) \neq (-y_i', d_i, -e_i)$. This establishes the lower bound of 2.

Part 4. Let $(\gamma_1, x_1, \varepsilon_1)$, $(\gamma_2, x_2, \varepsilon_2) \in \mathcal{G}_i^{(1)}(\tilde{S})$. Since both tuples satisfy Test 1, we have

$$(\gamma_1 \cdot M^{x_1})^2 \equiv y'_{i-1} \equiv (\gamma_2 \cdot M^{x_2})^2 \bmod N.$$

If $\gamma_1 \cdot M^{x_1} \not\equiv \pm\gamma_2 \cdot M^{x_2}$, a non-trivial factor of N can be computed as $\gcd(\gamma_1 \cdot M^{x_1} - \gamma_2 \cdot M^{x_2}, N)$ (see the proof of Lemma 2.19 for details).

We know that every tuple $(\gamma, x, \varepsilon) \in \mathcal{G}_i^{(1)}(\tilde{S})$ and the correct tuple (y'_i, d_i, e_i) satisfy the equation

$$(\gamma \cdot M^x)^2 \equiv y'_{i-1} \equiv (y'_i \cdot M^{d_i})^2 \bmod N \Rightarrow \gamma^2 \equiv (y'_i \cdot M^{d_i - x})^2 \bmod N.$$

Since $(y'_i, d_i, e_i) \in \mathcal{G}_i^{(1)}(\tilde{S})$, we must have $\gamma \equiv \pm y'_i \cdot M^{d_i - x} \bmod N$. Otherwise, a non-trivial factor of N can be found as explained above. □

Lemma 2.31 has provided some insight into the set of solution candidates surviving Test 1. Basically, the value y'_i is known up to the sign and the dependency on the secret bit d_i. The results still allow for multiple tuples in $\mathcal{G}_i^{(1)}(\tilde{S})$ with the same values γ and x, but with various values for ε. Since our goal is to compute y'_i and d_i, we do not need to consider tuples, which only differ in the guess for the error value. We can simply choose one of these tuples as a representative for the others and delete all others. Hence, such a situation is not a problem for our recovery strategy.

However, intuitively, due to the special structure of the error e_i, the occurrence of several tuples $(\gamma, x, \varepsilon_i) \in \mathcal{G}_i^{(1)}(\tilde{S})$ for several i should be improbable. This is indeed the case, as the following lemma shows.

Lemma 2.32 (Multiple Possible Error Values Are Improbable).
Let $\delta > 0$ be a fixed constant. Let $\mathcal{G}_i^{(1)}(\tilde{S})$ be defined as in Definition 2.30. Then at least one of the following claims holds for the set $\mathcal{G}_i^{(1)}(\tilde{S})$.

1. *The probability that $\mathcal{G}_i^{(1)}(\tilde{S})$ contains two solution candidates $(\gamma, x, \varepsilon_1)$ and $(\gamma, x, \varepsilon_2)$ with $\varepsilon_1 \neq \varepsilon_2$ is less than $1/n^\delta$. The probability is over the random choice of the message $M \in \mathbb{Z}_N$ given to the attack algorithm and the random choice of the decryption exponent d.*

2. *A non-trivial factor of N can be computed in expected polynomial time (in n).*

Proof:

The proof of this theorem is very similar to the proof of Lemma 2.14. As before, we prove the existence of an algorithm that factors all RSA moduli N for which Part 1 is false. The algorithm works as follows: it picks a random exponent d and random messages $M_1, \ldots, M_c \in \mathbb{Z}_N$. We have $c = n\log(2n)$ as in Fact 2.4. It then computes erroneous signatures $\tilde{S}_i$ of the M_i by using Algorithm 2.16 to compute $M_i^d \bmod N$ and deliberately simulating a bit fault according to Fault Model 1.7 at a random iteration. Let $(M_v, \tilde{S}_v)_{v=1}^c$ be the resulting set of faulty signatures. Then, it computes the set $\mathcal{G}_i^{(1)}(\tilde{S}_v)$ for all $\tilde{S}_v$ until two solution candidates satisfying Part 1 are found. We show there is a polynomial time (in n) algorithm that given this data succeeds in factoring N with high probability.

Computing the Set W_d. Let $(\gamma, x, \varepsilon_1)$ and $(\gamma, x, \varepsilon_2)$ be two tuples in the set $\mathcal{G}_i^{(1)}(\tilde{S})$ with $\varepsilon_1 \neq \varepsilon_2$. In this case, we know that they satisfy

$$\begin{aligned}(\gamma+\varepsilon_1)^{2^i} \cdot M^{2^i x} \cdot M^{l_{i-1}(d)} \equiv \tilde{S} \equiv (\gamma+\varepsilon_2)^{2^i} \cdot M^{2^i x} \cdot M^{l_{i-1}(d)} \bmod N \\ \Rightarrow (\gamma+\varepsilon_1)^{2^i} \equiv (\gamma+\varepsilon_2)^{2^i} \bmod N. \end{aligned} \tag{2.53}$$

Equation (2.53) implies that we can either compute a non-trivial factor of N using the approach of Lemma 2.19, or we must have $\gamma+\varepsilon_1 \equiv \pm(\gamma+\varepsilon_2) \bmod N$. This holds according to Lemma 2.19 if -1 is a quadratic non-residue modulo N. As explained at the beginning of this section, we assume this as a property of the modulus N.

However, $\gamma+\varepsilon_1 \equiv \gamma+\varepsilon_2 \bmod N$ implies $\varepsilon_1 \equiv \varepsilon_2 \bmod N$. Since $\varepsilon_1, \varepsilon_2 < N$, this implies that the two tuples are equal contrary to our assumption. Hence, unless N can be factored right away, we must have

$$\begin{aligned}\gamma+\varepsilon_1 \equiv -\gamma-\varepsilon_2 \bmod N \\ \Rightarrow \gamma \equiv -\frac{\varepsilon_1+\varepsilon_2}{2} \bmod N. \end{aligned} \tag{2.54}$$

From Lemma 2.31, Part 4, we know that $\gamma \equiv \pm M^{2h_{i+1}(d)+d_i-x} = \pm M^{h_i(d)-x} \bmod N$. Hence, the situation presented by Equation (2.54) may only occur if the chosen message M is a root of the polynomial

$$p(t) = t^\omega - a \bmod N \qquad \text{with} \quad \omega = h_i(d) - x \quad \text{and} \quad a = \pm(\varepsilon_1+\varepsilon_2)/2. \tag{2.55}$$

This yields the desired result with the same considerations as explained in detail in the proof of Theorem 2.24: We determine all possible polynomials $p(t)$ and collect these polynomials in the set W_d.

We know that ω can take at most two fixed values for a given set $\mathcal{G}_i^{(1)}(\tilde{S})$, however, there are up to $2\log(N)$ possible choices for ω for all possible values of i. This is due to the fact that w is build from prefixes of the secret key d. The value $a = \pm(\varepsilon_1+\varepsilon_2)/2$ can take less than $4\log(N)^2$ values. Hence, an adversary must construct at most $8\log(N)^3 = 8n^3$ polynomials $p(t)$ to compute W_d. Hence, we have $\#W_d < n^\tau$ with $\tau = 4$. Each polynomial can be constructed in polynomial time. We obviously have $\omega < N$ for all ω.

Factoring N. The above considerations show that the prerequisites for using Lemma 2.13 are satisfied. We now assume that Part 1 of Lemma 2.32 is false, i.e., the probability that $\mathcal{G}_i^{(1)}(\tilde{S})$ contains two candidates of the desired form is at least $1/n^\delta$. Since this implies that Part 1 of Lemma 2.13 is false as well, we can prove the existence of an algorithm, which computes a non-trivial factor of N in expected polynomial time. This concludes the proof. □

Lemma 2.31 and Lemma 2.32 show that with overwhelming probability, we have

$$\mathcal{G}_i^{(1)}(\tilde{S}) \subseteq \left\{ \begin{matrix} (\ \gamma,\ 1,\ \ \varepsilon_1),\ (\ \ \gamma \cdot M\ ,\ 0,\ \ \varepsilon_2), \\ (-\gamma,\ 1,\ -\varepsilon_1),\ (-(\gamma \cdot M),\ 0,\ -\varepsilon_2) \end{matrix} \right\}, \tag{2.56}$$

for $\gamma \equiv \pm y_i' \cdot M^{d_i-1} \bmod N$ and some $\varepsilon_1 = \pm 2^{b_1}$ and $\varepsilon_2 = \pm 2^{b_2}$, with $0 \leq b_1, b_2 < n$.

Although the solution candidates have been narrowed down to only 4 possibilities, this is not enough to guarantee a polynomial running time. Therefore, we need a second test, which effectively reduces the set $\mathcal{G}_i^{(1)}(\tilde{S})$ to size 1.

However, first, we will investigate another problem. If an adversary attacks the left-to-right repeated squaring algorithm, Algorithm 2.16, he has no control over the attacked iteration. This holds since we assume that the errors are induced according to Fault Model 1.7. Therefore, an adversary will collect a set $\mathcal{S}$ of at most c faulty signatures. Although he can compute the sets $\mathcal{G}_i^{(1)}(\tilde{S})$ for all $\tilde{S} \in \mathcal{S}$, he still does not know which resulting set contains the correct tuple (y_i', d_i, e_i). Hence, we need a result about the set $\mathcal{G}_i^{(1)}(\mathcal{S})$, containing all solution candidates computed for any $\tilde{S}$, which survive Test 1. Fortunately, as we will see in the next lemma, the set $\mathcal{G}_i^{(1)}(\tilde{S})$ for the right $\tilde{S}$ can be replaced by the set $\mathcal{G}_i^{(1)}(\mathcal{S})$ without many problems.

Lemma 2.33 (Multiple Faulty Signatures).
Let $\mathcal{S}$ be a set of faulty signatures. Let i, M, y_{i-1}', and $l_{i-1}(d)$ be given and let $\mathcal{G}_i^{(1)}(\mathcal{S})$ be defined as in Definition 2.30. For any two tuples $(\gamma_1, x_1, \varepsilon_1), (\gamma_2, x_2, \varepsilon_2) \in \mathcal{G}_i^{(1)}(\mathcal{S})$, one of the following cases holds.

1. *We have $\gamma_1 \cdot M^{x_1} \equiv \pm y_i' \cdot M^{d_i} \equiv \pm\gamma_2 \cdot M^{x_2} \bmod N$.*
2. *A non-trivial factor of N is given by $\gcd(\gamma_1 \cdot M^{x_1} - \gamma_2 \cdot M^{x_2}, N)$.*

Proof:

Let $(\gamma_1, x_1, \varepsilon_1), (\gamma_2, x_2, \varepsilon_2) \in \mathcal{G}_i^{(1)}(\mathcal{S})$ with $(\gamma_1, x_1, \varepsilon_1) \neq (\gamma_2, x_2, \varepsilon_2)$. We know from Definition 2.30 that both tuples from $\mathcal{G}_i^{(1)}(\mathcal{S})$ must satisfy the equation

$$(\gamma_1 \cdot M^{x_1})^2 \equiv y_{i-1}' \equiv \left(y_i' \cdot M^{d_i}\right)^2 \equiv (\gamma_2 \cdot M^{x_2})^2 \bmod N,$$

since every tuple is from a set $\mathcal{G}_i^{(1)}(\tilde{S})$ for some choice of $\tilde{S} \in \mathcal{S}$, and all such tuples must satisfy Test 1, which is independent of ε and $\tilde{S}$. We know that the correct tuple $(y_i, d_i, e_i) \in \mathcal{G}_i^{(1)}(\mathcal{S})$ as well. This implies that unless $\gamma_1 \cdot M^{x_1} \equiv \pm\gamma_2 \cdot M^{x_2} \equiv \pm y_i' \cdot M^{d_i} \bmod N$, a non-trivial factor of N can be computed as

$$\gcd(\gamma_1 \cdot M^{x_1} - \gamma_2 \cdot M^{x_2}, N).$$

□

Now we will show how to reduce the number of solution candidates to 1. In order to do so, we proceed to the next bit position $i+1$. At this time, we know that we have at most four different combinations of the values $\gamma = \pm y_i'$ and $x \in \{0, 1\}$ in $\mathcal{G}_i^{(1)}(\mathcal{S})$. We now consider all four pairs as valid guesses for y_i' and d_i and compute the set of guesses surviving Test 1 for the next step, i.e., the set $\mathcal{G}^{(1)}_{[i+1,M,\gamma,l_{i-1}(d)+2^i x]}(\mathcal{S})$. If this set is non-empty, then there is a tuple

$$(\gamma', x', \varepsilon') \in \mathcal{G}^{(1)}_{[i+1,M,\gamma,l_{i-1}(d)+2^i x]}(\mathcal{S})$$

such that

$$(\gamma' \cdot M^{x'})^2 \equiv \gamma. \tag{2.60}$$

Equation (2.60) implies that γ is a quadratic residue modulo N. Hence, by computing the set $\mathcal{G}^{(1)}_{[i+1,M,\gamma,l_{i-1}(d)+2^i x]}(\mathcal{S})$, we can learn a root of γ. We will therefore define Test 2 as the test of checking whether γ is a quadratic residue modulo N.

Definition 2.34 (Set of Guesses Surviving Test 2).
Let $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$ be a faulty signature. Let $\mathcal{G}_i^{(1)}(\tilde{S})$ be the set of guesses computed according to Lemma 2.27, given the known values $\tilde{S}$, M, i, $l_{i-1}(d)$, and y'_{i-1}, and all choices of $x \in \{0,1\}$ and $\varepsilon = \pm 2^b$, $0 \leq b < n$.

Let $(\gamma, x, \varepsilon) \in \mathcal{G}_i^{(1)}(\tilde{S})$. We define the set of guesses surviving Test 2, *which we denote by $\mathcal{G}^{(2)}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S})$, as the set of pairs (γ, x) given by tuples from the set $\mathcal{G}^{(1)}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S})$ satisfying*

$$\textit{Test 2:} \qquad \exists \quad a \in \mathbb{Z}_N : a^2 \equiv \gamma \bmod N, \tag{2.61}$$

i.e.,

$$\mathcal{G}^{(2)}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S}) := \left\{ (\gamma, x) \;\middle|\; \begin{array}{l} (\gamma, x, \varepsilon) \in \mathcal{G}_i^{(1)}(\mathcal{S}) \text{ for some } \varepsilon, \\ \gamma \text{ is a quadratic residue modulo } N \end{array} \right\}.$$

If the input values i, M, y'_{i-1}, and $l_{i-1}(d)$ are not important or clear from the context, we will abbreviate $\mathcal{G}^{(2)}_{[i,M,y'_{i-1},l_{i-1}(d)]}(\tilde{S})$ as $\mathcal{G}_i^{(2)}(\tilde{S})$. For a given tuple $(\gamma, x, \varepsilon) \in \mathcal{G}_i(\tilde{S})$, we refer to the tuple $(-\gamma, x, -\varepsilon)$ as the twin tuple. *For a set $\mathcal{S}$ of faulty signatures as defined in Definition 2.29, we define*

$$\mathcal{G}_i^{(2)}(\mathcal{S}) := \bigcup_{\tilde{S} \in \mathcal{S}} \mathcal{G}_i^{(2)}(\tilde{S}). \tag{2.62}$$

We define the set $\mathcal{G}_i^{(2)}(\mathcal{S})$ as the set of pairs (γ, x) rather than the set of tuples (γ, x, ε). This is reasonable, since we do not need to know the exact error value to continue recovering more significant bits. We are only interested to recover y'_i and d_i exactly. We would like to emphasize the fact that we may not know the exact value for e_i. This is due to the fact that $\mathcal{S}$ may contain more than one faulty signature $\tilde{S}$ with a fault induced into y'_i. Hence, there may be several correct tuples (y_i, d_i, e_i) for several values of e_i and different values $\tilde{S}$.

We will now show that we can determine whether γ is a quadratic residue modulo N or not by computing the set $\mathcal{G}^{(1)}_{[i+1,M,\gamma,l_{i-1}(d)+2^i x]}(\mathcal{S})$. If this set is non-empty, γ is a quadratic residue modulo N as explained above. However, we will use two assumptions to prove that this reduces the number of solution candidates to 1. First, we assume that -1 is a quadratic non-residue. This is the overall assumption in this section. It will be used to show that either a given tuple or its twin tuple will survive Test 2. Second, we will assume that both M and $-M$ are also quadratic non-residues modulo N. This assumption will show that of the remaining two tuples (assuming twin tuples have been sorted out already) only one may survive Test 2. Since we may choose M uniformly at random, we expect both M and $-M$ to be quadratic non-residues modulo N with probability 1/2. Hence, this assumption is practical, since an adversary may start over if M is not of the desired type.

Lemma 2.35 (Eliminating False Solution Candidates: Test 2).
Let $\mathcal{S}$ be a set of faulty signatures, containing $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$. Let $\mathcal{G}_i^{(1)}(\mathcal{S})$ be defined as in Definition 2.30, and let $\mathcal{G}_i^{(2)}(\mathcal{S})$ be defined as in Definition 2.34. If -1, M, and $-M$ are quadratic non-residues modulo N, then we have $\mathcal{G}_i^{(2)}(\mathcal{S}) = \{(y_i, d_i)\}$, unless N can be factored according to Lemma 2.31, Part 4, Lemma 2.32, or Lemma 2.33.

Proof:

Let $(\gamma, x, \varepsilon) \in \mathcal{G}_i^{(1)}(\mathcal{S})$. According to Lemma 2.33, we know that $\gamma \cdot M^x \equiv \pm y_i \cdot M^{d_i} \bmod N$. Hence, there are only four possible values for (γ, x), namely $(\pm y_i \cdot M^{d_i - x}, x)$ for $x \in \{0, 1\}$. Otherwise, N can be factored right away.

We now treat the solution candidate (γ, x, ε) as the correct solution, i.e., we assume that $y_i' = \gamma$, $d_i = x$ and $e_i = \varepsilon$. This allows us to compute solution candidates for y_{i+1}' and d_{i+1}. We compute the set $\mathcal{G}^{(1)}_{[i+1,M,\gamma,l_{i-1}(d)+2^i x]}(\mathcal{S})$ according to Definition 2.30. This is based on the computation of solution candidates for all single faulty signatures $\tilde{S} \in \mathcal{S}$ according to Lemma 2.27. We will write $\mathcal{G}_{i+1}^{(1)}(\mathcal{S})$ short for $\mathcal{G}^{(1)}_{[i+1,M,\gamma,l_{i-1}(d)+2^i x]}(\mathcal{S})$.

Since all solution candidates from the set $\mathcal{G}_{i+1}^{(1)}(\mathcal{S})$ must satisfy Test 1 for the given choice of γ and x, we have for every $(\gamma', x', \varepsilon') \in \mathcal{G}_{i+1}^{(1)}(\mathcal{S})$ that

$$\left(\gamma' \cdot M^{x'}\right)^2 \equiv \gamma \bmod N. \tag{2.63}$$

Equation (2.63) implies that γ must be a quadratic residue modulo N. If γ is the correct value for $y_i' = M^{2h_{i+1}(d)}$, this is obviously satisfied. Therefore, we have $(y_i', d_i) \in \mathcal{G}_i^{(2)}(\mathcal{S})$.

However, we have seen that the initial set $\mathcal{G}_i^{(1)}(\mathcal{S})$ contains twin pairs of values, i.e., pairs of tuples of the form (γ, x, ε) and $(-\gamma, x, -\varepsilon)$. If both choices of γ and $-\gamma$ yield non-empty solution sets $\mathcal{G}_{i+1}^{(1)}(\mathcal{S})$, both values are quadratic residues modulo N. This implies that there are values ρ, $\sigma \in \mathbb{Z}_N$ such that $\rho^2 \equiv \gamma \bmod N$ and $\sigma^2 \equiv -\gamma \bmod N$. In this case, we know that $(\rho/\sigma)^2 \equiv -1 \bmod N$. Since we assume that -1 is a quadratic non-residue modulo N, this is impossible. Therefore, at most one of the two tuples of a twin pair may yield a non-zero solution set $\mathcal{G}_{i+1}^{(1)}(\mathcal{S})$. Since the correct tuple (y_i', d_i, e_i) yields a non-zero solution set, we can detect and eliminate at least half of all elements from $\mathcal{G}_i^{(1)}(\tilde{S})$ as false guesses.

Now assume that two tuples $(\pm\gamma \cdot M, 0, \varepsilon_1)$ and $(\gamma, 1, \varepsilon_2)$ both yield non-zero solution sets $\mathcal{G}_{i+1}^{(1)}(\mathcal{S})$. We know that every tuple from $\mathcal{G}_i^{(1)}(\mathcal{S})$ satisfies one of these two patterns with $\gamma = \pm y_i' \cdot M^{d_i - 1} \bmod N$ according to Lemma 2.33. Since both tuples yield non-zero solution sets, we know that both $\pm\gamma \cdot M$ and γ are quadratic residues modulo N. Therefore, we have two values ρ, $\sigma \in \mathbb{Z}_N$ such that $\rho^2 \equiv \pm\gamma \cdot M \bmod N$ and $\sigma^2 \equiv \gamma \bmod N$. This implies that

$$(\rho/\sigma)^2 \equiv \pm M \bmod N, \tag{2.64}$$

i.e., ρ/σ is a square root of $\pm M$. Only one of the two values $\gamma \cdot M$ and $-\gamma \cdot M$ may be present, otherwise -1 would be a quadratic residue as well, contradicting the overall assumption in this section.

For a randomly chosen message M, the probability that both M and $-M$ are quadratic non-residues modulo N is approximately $1/2$: For a random $M \in \mathbb{Z}_p^*$, the probability that M is a square modulo p is $1/2$, the same holds for the probability that $M \in \mathbb{Z}_q^*$ is a square modulo q. Since 0 is always a square modulo both p and q, the probability that a randomly chosen message $M \in \mathbb{Z}_N$ is a square modulo p is $1/2 + 1/(2q)$ and a similar consideration holds modulo q. Hence, the probability that a given M is a square modulo both p and q, and hence modulo N, is negligibly higher than $1/4$. More precisely, there are about $N/4$

many messages $M \in \mathbb{Z}_N$, which are squares modulo N. Therefore, there are also about $N/4$ many messages M', such that $M := -M'$ is a square modulo N. Consequently, the probability that for a randomly chosen message M, both M itself and its additive inverse $-M$ are both quadratic non-residues is about $1/2$.

Hence, with probability negligibly lower than $1/2$, a message M chosen uniformly at random from $\mathbb{Z}_N$ cannot yield Equation (2.64). In this case, the set $\mathcal{G}_i^{(2)}(\mathcal{S})$ contains at most one pair (γ, x) from $\mathcal{G}_i^{(1)}(\mathcal{S})$. This identifies the correct pair (y_i', d_i). □

The result from Lemma 2.35 shows that the correct pair (y_i', d_i) can be recovered with probability negligibly far from $1/2$, if M is chosen uniformly at random and the values y_{i-1}', i, and l_{i-1} are known. The results gathered so far allow for the following inductive algorithm.

Algorithm 2.36: Attack on Left-To-Right Repeated Squaring, LSB Version

Input: Access to Algorithm 2.16, n the maximal length of the secret key d, e the RSA public key
Output: d with probability at least 1/4.

\# **Phase 1:** Collect Faulty Outputs
Set $c := n \cdot \log(2n)$
Choose $M \in \mathbb{Z}_N$ uniformly at random.
Create c faulty outputs of Algorithm 2.16 on input M by inducing faults according to Fault Model 1.7 into the intermediate variable y_i' at random iterations.
Collect the set $\mathcal{S} = \{ \tilde{S} \mid M$ the input to Algorithm 2.16 and $\tilde{S}$ a faulty output$\}$.
Compute the correct signature $S \equiv M^d \bmod N$ by **not** inducing a fault into Algorithm 2.16
\# **Phase 2:** Inductive Retrieval of Secret Key Bits
Set $Y = \{(S/M \bmod N, 1)\}$ holding the element $(y_0', l_0(d))$
For s from 1 to n-2 do
    Set $Y' := \{\}$
    For all verified/"known" candidates $(y_{s-1}', l_{s-1}(d))$ in Y do
        For all $x \in \{0, 1\}$ do
            For all $\varepsilon = \pm 2^b$ with $0 \leq b < n$ do
                For all $\tilde{S} \in \mathcal{S}$ do
\# Compute a solution candidate γ for y_i' using A and B according to Lemma 2.27

Set $$A := \sum_{k=0}^{2^{s-1}-1} \binom{2^s}{2k+1} \cdot \left(\frac{y_{s-1}'}{M^{2x}}\right)^k \cdot \varepsilon^{2^s-(2k+1)} \bmod N.$$

Set $$B := \sum_{k=0}^{2^{s-1}} \binom{2^s}{2k} \cdot \left(\frac{y_{s-1}'}{M^{2x}}\right)^k \cdot \varepsilon^{2^s-2k} \bmod N.$$

Set $$\gamma := \left(\frac{\tilde{S}}{M^{l_{s-1}(d)+2^s x}} - B\right) \Big/ A \bmod N.$$

\# Verify the solution candidate using Test 1 according to Definition 2.30
If $(\gamma \cdot M)^2 \equiv y_{s-1}' \bmod N$ then add $(\gamma, l_{s-1}(d) + 2^s \cdot x)$ to the set Y'
    Set $Y := Y'$
\# **Phase 3:** Inductive Retrieval of the Most Significant Key Bit
For all verified/"known" candidates $(y_{n-2}', l_{n-2}(d))$ in Y do
    Set $d := l_{n-2}(d) + 2^{n-1}$
    If $M^{d \cdot e} \equiv M \bmod N$ then **output** d
Output FAILURE

Algorithm 2.36 is similar to the attack algorithms presented in previous sections. It tries to recover the bits of d one by one starting from the least significant bit.

As previous algorithms stated before, Algorithm 2.36 has been abbreviated in one minor detail. We base the success probability and, more importantly, the expected running time of Algorithm 2.36 on the fact that the occurrence of special solution candidates allow the adversary to factor N easily, thus rendering the remainder of our attack algorithm needless (see Lemma 2.31, Part 4, Lemma 2.32, and Lemma 2.33). However, we have omitted any checking procedures, which might detect the occurrence of such a special case when stating Algorithm 2.36 above. First, we do not expect such special cases to occur with non-negligible probability, hence, an adversary may rely on the fact that these cases do not occur. Second, we are confident that omitting these details add to clarity at negligible loss of completeness.

The following theorem summarizes the results of this section, showing that Algorithm 2.36 successfully recovers the secret key d in expected polynomial time.

Theorem 2.37 (Success and Running Time of Algorithm 2.36).
Let $N = p \cdot q$ be an n-bit RSA modulus. Given $n \cdot \log(2n)$ faulty signatures $\tilde{S}$ for some message M, chosen uniformly at random from $\mathbb{Z}_N$, the secret exponent d can be extracted from Algorithm 2.2 with probability at least $1/4$. The probability is over the location of the register faults and the random message $M \in \mathbb{Z}_N$. The algorithm's running time is dominated by the time it takes to perform $O(n^3 \log(n))$ full modular exponentiations mod$N$.

Proof:

We will investigate the costs of all of the three phases of Algorithm 2.36 separately.

Phase 1. In the first phase, the total costs are fixed. Here, $c = n \cdot \log(2n)$ full modular exponentiations must be performed, each of which is subject to a fault induced by the adversary. This is followed by one full modular exponentiation, where no fault is induced, which is used to derive the correct signature S.

Phase 2. In Phase 2, we first count the number of operations required for each iteration of Lines 10–16. Here, we have 3 nested loops, which loop through all possible tuples $(x, \varepsilon, \tilde{S})$. Therefore, the computations in Lines 13–16 have to be performed at most $2 \cdot 2n \cdot c$ times. The computation of A, B, and γ can be performed using Algorithm 2.28 in the time required to perform 7 full modular exponentiations. This has been shown in Lemma 2.27.

The computations in Line 16 require one multiplication, one squaring, one comparison, and one set operation. For the set operation, the insertion into a set may be assumed to require logarithmic time in the number of elements already present in the set (using a standard binary search tree structure). However, Lemma 2.33 combined with Lemma 2.35 show that Y' may contain at most 4 elements. This holds since there are at most 2 possible values for x, and also for $l_{i-1} + 2^i x$, and at most 4 possible values for $\gamma \equiv \pm y'_i \cdot M^{d_i - x} \bmod N$, which may satisfy Test 1 and only one choice of $(y'_{i-1}, l_{i-1}(d))$, which satisfies Test 2. If more values are present, N can be factored according to Lemma 2.33, which instantly terminates our algorithm. Therefore, the cost of the set operation can be neglected. The remaining costs can easily be bounded by the cost of one full modular exponentiation.

Therefore, for each pair $(y'_{i-1}, l_{i-1}(d)) \in Y$, the Loop of Lines 10–16 accounts for at most $32 \cdot n \cdot c = O(nc)$ full modular exponentiations.

The computation for one given pair $(y'_{i-1}, l_{i-1}(d)) \in Y$, which is done in Lines 10–16, computes the set $\mathcal{G}_i^{(1)}(\mathcal{S})$ defined in Definition 2.30. We know by Lemma 2.33, that $\mathcal{G}_i^{(1)}(\mathcal{S})$ contains at most 4 different combinations (γ, x), unless N can be factored. Hence, the set Y' contains at most 4 elements. However, we have seen in Lemma 2.35 that if Y contains 4 elements, only 1 of them can yield a non-empty set Y'. This depends on the choice of M, which has to be a quadratic non-residue modulo N, as well as $-M$. Since M is chosen uniformly at random in Line 2, this happens with probability 1/2. We only claim that our algorithm is successful in this case. Hence, the set Y may be at most of size 4 in any round. This implies that in each iteration of the loop of Lines 7–16, at most $4 \cdot 32 \cdot nc = O(nc)$ full modular exponentiations have to be performed. Since we have $n-2$ rounds of the main loop, Lines 7–16, we have a total of at most

$$128 \cdot nc \cdot (n-2) = O(cn^2) = O(n^3 \log(n))$$

full modular exponentiations.

Phase 3. In Phase 3, there are at most 4 pairs in the set Y, all of which are being tested. For each of these 4 pairs, one exponentiation (of 2^i), one multiplication, and one addition has to be performed in order to compute d. Then, a multiplication and a full modular exponentiation tests the derived value for d. This requires at most $4 \cdot 2 = O(1)$ full modular exponentiations.

Summary. Adding up the results for the three phases, we can conclude that Algorithm 2.36 runs in the time required to compute at most $O(n^3 \log(n))$ full modular exponentiations or full modular exponentiations with the ability to induce faults.

Correctness. We have seen in Lemma 2.27 that the correct value y'_i is computed in Line 15, if the correct values i, M, $\tilde{S}$, y'_{i-1}, l_{i-1}, d_i, and e_i are used to compute A, B, and γ. We assume by induction that eventually, one of the values from Y represents $(y'_{i-1}, l_{i-1}(d))$. We test all possible values for x and ε, i.e., at one time, we also test d_i and e_i. The correct values for i and M are obviously given at all times. However, in order to guarantee that the correct value y'_i is ever computed, we rely on the fact that there is a faulty signature $\tilde{S} \in \mathcal{S}$, such that $\tilde{S} = (y'_i + e_i)^{2^i} \cdot M^{l_i(d)} \bmod N$. We know that this assumption is justified with probability 1/2 according to Fact 2.4. We do not claim that our algorithm succeeds otherwise. Since the correct values for $\gamma = y'_i$ and $x = d_i$ satisfy Test 1 performed in Line 16, we know that $(y'_i, l_i(d)) \in Y'$. This ensures that the correct values $(y'_{n-2}, l_{n-2}(d))$ are tested in Phase 3. Here, we exploit the fact that $d_{n-1} = 1$. Hence, $d = 2^{n-1} + l_{n-2}(d)$. Therefore, the algorithm will output d with probability 1/2 given the fact that both M and $-M$ are quadratic non-residues modulo N.

Since the polynomial running time depends on the message M, which satisfies the condition that both M and $-M$ are quadratic non-residues modulo N with probability 1/2, we may conclude that Algorithm 2.36 recovers d in the time required to perform $O(cn^2) = O(n^3 \log(n))$ full modular exponentiations or full modular exponentiations with the ability to induce faults with probability 1/4. □

In practice, Algorithm 2.36 can be enhanced in several ways, e.g., exchanging Lines 13–14 with Line 12 increases the speed. However, this enhancement is only of practical interest, since it only lowers the constants hidden in the O-notation. Compared to the original attack due to Boneh, DeMillo, and Lipton on the right-to-left repeated squaring algorithm (see Theorem 2.15), our algorithm requires the same number of modular exponentiations (assuming $m = 1$). Therefore, the susceptibility of both variants of repeated squaring to fault attacks is the same. We have only presented an attack with the same running time for $m = 1$, for $m > 1$, the attack has to be adapted, or the slower attack starting from the most significant bits presented as Algorithm 2.21 has to be used.

Remark. In this section, we have assumed that $m = 1$, hence, that $c = n \cdot \log(2n)$ and every iteration i in Algorithm 2.16 has suffered a fault with probability 1/2. However, it should be clear that a similar analysis would also show that our algorithm can be modified to work with larger values of m. This is desirable, if an adversary can only induce a limited number of faults into Algorithm 2.16.

2.3. Concluding Remarks and Open Problems

In this chapter, we have presented the important algorithm of Boneh, DeMillo, and Lipton, starting off research in the area of fault attacks. We have identified two minor flaws in the original attack presented in [BDL01], which could easily be fixed. Afterwards, we have solved the open problem of transferring the attack on the right-to-left repeated squaring algorithm to its twin, the left-to-right repeated squaring algorithm. We have presented two different approaches, one starting from the most significant bits (in Section 2.2.1), and one starting from the least significant bits (in Section 2.2.2). Both attacks show that the left-to-right repeated squaring algorithm is susceptible to fault attacks in the same manner and at the same computational costs as the right-to-left repeated squaring algorithm. Here, we have seen that for our analyses, the LSB recovery strategy presented in Section 2.2.2 is asymptotically faster than the MSB recovery strategy.

However, our attacks have only been analyzed for the special case of RSA moduli N, which ensure that -1 is a quadratic non-residue modulo N. Since strong primes, which ensure that $(p-1)/2$ and $(q-1)/2$ are primes, are often recommended for use for RSA, and this choice ensures the assumed property of $N = p \cdot q$, our constraints are reasonable. However, we would like to emphasize that our attack algorithms also worked for randomly chosen primes p and q in numerous experiments. However, proving the success probability for general RSA moduli N still is an open problem. Our analyses heavily depend on Lemma 2.19, which ensures that congruences of the form $x^{2^i} \equiv y^{2^i} \bmod N$ allow to factor N unless $x \equiv \pm y \bmod N$. Certain results from Schnorr [Sch96] indicate that this property can be generalized to allow to state more general proofs. However, our goal to show that the attacks from [BDL01] can be applied to the left-to-right repeated squaring algorithm does not require this generalization.

Remark. Similar to the original attack on right-to-left repeated squaring as presented in [BDL01], we have assumed in our attacks that both the message M and the decryption exponent d were chosen at random. However, we agree with the remark of Boneh, DeMillo, and Lipton [BDL01, p. 110], that it should be clear that heuristically, our algorithms will work for

almost any set of chosen messages and any secret key d. Especially, this holds for secret keys d corresponding to low public exponents, such as $e = 2^{16} + 1$.

3. A New Countermeasure Against Attacks on CRT-RSA

In the last section, we have presented attacks on RSA with a standard variant of repeated squaring, which is the classic method to perform modular exponentiations. It has been shown that these variants are susceptible to fault attacks. In this section, we will investigate another method for modular exponentiation in RSA, CRT-RSA. On small mobile devices such as smartcards, time is of much greater importance than on desktop computers. Here, even small speedups may result in an advantage over the products of other vendors, which is of great importance in practice. For such environments, CRT-RSA offers a faster version of modular exponentiation than ordinary repeated squaring. Here, the computations are sped up using the Chinese Remainder Theorem. However, this fast method has been proven to be highly susceptible to fault attacks. In this section, we will first introduce CRT-RSA and the attack from [BDL01]. As this attack is a serious threat to many modern implementations of RSA, one major goal of this thesis has been to develop a countermeasure, which protects such implementations from fault attacks. In Section 3.3, we present a modified algorithm for CRT-RSA, which secures CRT-RSA against fault attacks in the most realistic fault model at the expense of few additional operations.

3.1. CRT-RSA and the Bellcore Attack

In RSA, the modulus N is the product of two large primes p and q. The main operation, the modular exponentiation with either the secret or the public key, can be computed modulo N using repeated squaring in either variant presented in Chapter 2. For an exponent d, this requires $l(d) + \nu(d) - 2 = O(\log(d)) = O(\log(N))$ multiplications, where $l(d) = \lfloor \log(d) \rfloor + 1$ is the binary length of the exponent d, and $\nu(d)$ is the binary Hamming-weight of d. The complexity of the multiplication is usually assumed to be quadratic in the size of the modulus, although faster multiplication methods exist (for details confer [GG99]). There are several faster methods than repeated squaring, e.g., using optimal addition chains (cf. [Ott01]), but they are hardly used in practice. This is mainly due to the fact that computing the optimal addition chain is a hard problem, considered to be NP-hard (cf. [DLS81]). Moreover, repeated squaring is extremely easy to implement, and even the optimal addition chain has a runtime of $O(l(d))$. For an overview of fast exponentiation methods, confer [Gor98].

Although repeated squaring is not a slow operation in general, its computation time is still the bottleneck for mobile devices such as smartcards. A user does not tolerate a card with a noticeable time delay. The solution to this problem given increasing RSA key lengths is to both incorporate RSA accelerators in hardware (see Figure 1.2 on page 13) as well as to use the Chinese Remainder Theorem (CRT) to speed up RSA [CQ82]. All speedups are negligible from the view of computational complexity, i.e., in big O-notation, non of them manages to be asymptotically faster than $\log(d)$ — yet in practice, constants matter much. We present the CRT-RSA algorithm in the context of RSA signatures, i.e., with the secret RSA key d as the exponent.

Algorithm 3.1: CRT-RSA Algorithm

Input: $\mathsf{m} \in \mathbb{Z}_\mathsf{N}$ the message
Output: $\mathsf{s} = \mathsf{m}^\mathsf{d} \bmod \mathsf{N}$
In Memory: the secret RSA key d with $\mathsf{d_p} = \mathsf{d} \bmod (\mathsf{p}-1)$ and $\mathsf{d_q} = \mathsf{d} \bmod (\mathsf{q}-1)$, the RSA modulus $\mathsf{N} = \mathsf{p} \cdot \mathsf{q}$, and both primes p and q

1 $\mathsf{S_p} := \mathsf{m}^{\mathsf{d_p}} \bmod \mathsf{p}$
2 $\mathsf{S_q} := \mathsf{m}^{\mathsf{d_q}} \bmod \mathsf{q}$
3 $\mathsf{s} := \mathsf{CRT}(\mathsf{S_p}, \mathsf{S_q})$

The CRT combination in Line 3 of Algorithm 3.1 is usually done using Garner's Scheme [Gar59], which computes s as

$$s = S_p + X \cdot (S_q - S_p) \bmod N, \qquad \text{where } X = p \cdot (p^{-1} \bmod q). \tag{3.1}$$

Note that the value X can be precomputed. As an alternative method to Garner's Scheme, a method usually referred to as *Gauß's Scheme* could be used [MvOV96, p. 68]. However, Gauß's Scheme is computationally inferior and allows the same attacks as Garner's Scheme.

CRT-RSA is on average four times faster than a single modular exponentiation using repeated squaring [CQ82]. The reason for the speedup is the decreased length of the modulus, which is now p or q instead of N, yielding intermediate results which are only half as long as usual. With classical arithmetic, a speedup of factor four follows directly, but we need to compute two results S_p and S_q. Moreover, the exponent d can be reduced modulo $p-1$ or $q-1$, which gives an additional speedup of 2 on average for random d. The secret key d must be a large number in order to be secure against a variety of lattice attacks (cf. [BD00]). Hence, this yields a total speed-up factor of 4 on average. Additionally, if the architecture supports parallelism, the two values S_p and S_q may be computed in parallel, which adds an additional speedup of factor 2, yielding a total factor of 8 in this case.

CRT-RSA proved to be highly susceptible to fault attacks. In [BDL97], Boneh, DeMillo, and Lipton describe an extremely simple fault attack on CRT-RSA. Named the *Bellcore attack*, this attack reveals the secret factorization of the RSA modulus N by introducing a single fault. It requires a signature that is correct modulo one of the secret prime factors of N, but faulty modulo the other prime factor. This attack is particularly devastating because the type of fault induced is irrelevant.

The Bellcore Attack. We now describe this attack in greater detail. Assume that "some error" happens during the computation of S_p, such that $S_p \nmapsto \tilde{S}_p = S_p + e(S_p)$ with $e(S_p) \not\equiv 0 \bmod p$. Here, the fault model for $e(S_p)$ is irrelevant as long as $\tilde{S}_p \not\equiv S_p \bmod p$. We have captured this notion in Fault Model 1.10. While the correct value S_q is used, the CRT combination yields a faulty final signature $\tilde{s}$. For $\tilde{s}$ it holds that

$$\begin{aligned} &\tilde{s} \bmod p \equiv \tilde{S}_p \not\equiv S_p \equiv s \bmod p \quad \text{and} \\ &\tilde{s} \bmod q \equiv S_q \equiv s \bmod q. \end{aligned}$$

Consequently, given a correct signature s, it is obvious that

$$\begin{aligned} (s - \tilde{s}) \equiv 0 \bmod q \quad &\Leftrightarrow \quad q \mid (s - \tilde{s}) \quad \text{but} \\ (s - \tilde{s}) \not\equiv 0 \bmod p \quad &\Leftrightarrow \quad p \nmid (s - \tilde{s}). \end{aligned}$$

This fact allows to derive one of the secret prime factors. We have

$$q = \gcd(s - \tilde{s}, N).$$

A. K. Lenstra observed that the attack also works even if no correct signature s is given (cf. [BDL01]), using the public key e. We have

$$q = \gcd(m - \tilde{s}^e, N).$$

The attack described above also works if it is applied at different locations. Due to symmetry, it is clear that if a faulty value for S_q is computed in Line 2 of Algorithm 3.1, we get $p = \gcd(s - \tilde{s}, N)$.

In the remainder of this chapter, we will refer to *Bellcore-like attacks* as attacks, which recover the secret key by inducing just a single fault into some intermediate variable.

Other Bellcore-like Attacks. Attacks during the CRT combination may also yield a non-trivial factor of N. As a Bellcore-like attack, we consider attacks, which yield a final result, that is correct modulo exactly one of the two secret prime factors of N and faulty modulo the other. In the following, we present the results for faults induced into all parameters used in Garner's Scheme (3.1). To the best of our knowledge, this summary has not been presented in the literature before. However, the main ideas have already been described in [ABF+02] and [YMH03]. We will state success probabilities for all reasonable fault models, i.e., the Single Bit Fault Model 1.7, the Byte Fault Model 1.8, and the Random Fault Model 1.9. We do not consider the Arbitrary Fault Model 1.10, since this model does not allow to bound the probability that any desired value will be created. We also do not consider the Chosen Bit Fault Model 1.6 either, since this model is too strong. An adversary capable of mounting a fault attack according to this fault model does not need to create special values, he can always mount an oracle attack. We base our analysis on the following observations.

Proposition 3.2 (Probability of Special Error Values).
Let x be a faulty variable of maximal length $n := l(x)$, and let $e(x)$ be the error induced into x. Let $p, q > 2^8$ be two distinct large primes, and let $N = p \cdot q$.

If $e(x)$ is an error according to the Single Bit Fault Model 1.7, or to the Byte Fault Model 1.8, then we have

$$\text{Prob}[p \text{ divides } e(x) \text{ and } q \text{ does not divide } e(x)] = 0.$$

If $e(x)$ results from a fault according to the Random Fault Model 1.9, then

$$\text{Prob}[p \text{ divides } e(x) \text{ and } q \text{ does not divide } e(x)] < \frac{2}{p}.$$

Proof:

If $e(x)$ results from a bit error, we have $e(x) = \pm 2^k \bmod N$ according to Definition 1.7. As $p \mid N$, we have $e(x) \equiv \pm 2^k \bmod p$ as well. If $e(x) \equiv 0 \bmod p$, p must divide 2^k. However, as p is a prime larger than 2, dividing any power of 2 is impossible. If $e(x)$ results from a byte error, we have $e(x) = \pm b \cdot 2^k \bmod N$ according to Definition 1.8. Again, since $b < 2^8$, p cannot divide $e(x) \bmod N$. If $b = 0$, we have $e(x) = 0$, hence, p divides $e(x)$. However, in this case, q divides $e(x)$ as well.

Let $e(x)$ be a random fault according to the Random Fault Model 1.9, which takes any value from $\{-x, -x+1, \ldots, -x+2^n-1\}$ with uniform probability. In this case, $e(x)$ can

take at most $2^n/p+1$ many values which are multiples of p. By not subtracting the number of those values, which are also multiples of q, we may at most increase the probability. This yields a probability of at most $(2^n/p+1)/2^n = 1/p + 1/2^n$. If $2^n \geq p$, this yields a probability at most $2/p$. Clearly, it is less than $2/p$, since the value $e(x) = 0$ does not yield a desired result. If 2^n is smaller than p, then there is at most one multiple of p in the interval $\{-x, -x+1, \ldots, -x+2^n-1\}$, which is always $e(x) = 0$. As this is explicitly excluded, the probability is 0, i.e., less than $2/p$. □

Proposition 3.3 (About the Value $S_q - S_p$).
Let $N = p \cdot q$ be an RSA modulus with $p < q$ and $l(p) = l(q)$. Additionally, let d be a secret RSA key, and let $m \in \mathbb{Z}_N$ be a random message. For $S_p = m^d \bmod p$ and $S_q = m^d \bmod q$, we have

$$\text{Prob}[(S_q - S_p) \equiv 0 \bmod p \text{ and } (S_q - S_p) \not\equiv 0 \bmod q] \leq \frac{1}{p}$$

and $$\text{Prob}[(S_q - S_p) \equiv 0 \bmod q \text{ and } (S_q - S_p) \not\equiv 0 \bmod p] = 0.$$

Proof:

To compute the desired probabilities, we use the fact that S_p and S_q may be seen as independent random values. This is justified by the fact that m is random and it is reasonable to assume that p and q are independent. To be exact, $p \neq q$ is required for RSA, hence, p and q are not completely independent, however, this may be neglected. We have

$$\begin{aligned}\text{Prob}[(S_q - S_p) = k] &= \sum_{c=0}^{q-1} \text{Prob}[(S_q - S_p) = k \mid S_q = c] \cdot \text{Prob}[S_q = c] \\ &= \sum_{c=0}^{q-1} \text{Prob}[(c - S_p) = k] \cdot \text{Prob}[S_q = c] \\ &\leq q \cdot \frac{1}{p} \cdot \frac{1}{q} = \frac{1}{p}.\end{aligned}$$

We know that $(S_q - S_p) \in \{-p+1, -p+2, \ldots, q-1\}$, hence, the case where $S_q - S_p \neq 0$ but $S_q - S_p \equiv 0 \bmod p$ or $S_q - S_p \equiv 0 \bmod q$ may only occur at all if $|S_q - S_p| = \min(p, q)$. If we assume that $p < q$, we may have $|S_q - S_p| = p$, and we get $\text{Prob}[(S_q - S_p) \equiv 0 \bmod p \text{ and } (S_q - S_p) \not\equiv 0 \bmod q] \leq 1/p$ and $\text{Prob}[(S_q - S_p) \equiv 0 \bmod q \text{ and } (S_q - S_p) \not\equiv 0 \bmod p] = 0$. □

Now, we investigate attacks on every variable used in Garner's Scheme (3.1). Some variables can be attacked successfully with a Bellcore attack.

1. **Attack targeting the leftmost S_p: Bellcore-like attack is possible.** If the leftmost use of the variable S_p is affected by an induced fault, we differentiate between transient and permanent errors.

   In case of a permanent error, the second use of S_p would be faulty as well. This case represents the same situation as if S_p would have been affected in Line 1 of Algorithm 3.1. Hence, any nonzero error value allows the Bellcore attack. The probability of inducing a nonzero error term is 1 in the Single Bit Fault Model, it is $1 - 1/256$ in the Byte Fault Model and $1 - 1/2^{l(p)}$ in the Random Fault Model (where we assume that $l(S_p) = l(p)$).

In case of a transient error, the final result will be $\tilde{s} = s + e(S_p)$. For the Single Bit Fault Model and the Byte Fault Model, an adversary can easily construct all possible values even without access to a smartcard. This is possible if he knows the correct signature s. Moreover, according to Proposition 3.2, there is no bit flip or byte value, which may yield a value $e(S_p)$, which is zero modulo one prime and non-zero modulo the other. Consequently, we have a zero success probability for a Bellcore-like attack for the Single Bit Fault Model and the Byte Fault Model. For the Random Fault Model, we have a success probability of less than $4/\min(p, q)$ according to Proposition 3.2. This success probability basically resembles exhaustive trying of divisors of N.

2. **Attack targeting the precomputed value X: Bellcore-like attack is infeasible.** If X is affected by a fault, the final result is $\tilde{s} = s + e(X) \cdot (S_q - S_p)$. In order to apply the Bellcore attack, we need that either $e(X)$ is zero modulo exactly one of p and q, or that the difference $(S_q - S_p)$ is zero modulo either p or q.

   In the first case, which states that $e(X)$ must be divided by exactly one of the two secret primes, we know that the success probability is zero for the Single Bit Fault Model and the Byte Fault Model. For the Random Fault Model, we have a success probability of less than $4/\max(p, q)$. All these results follow immediately from Proposition 3.2.

   In the second case, which requires that $(S_q - S_p) \equiv 0$ modulo exactly one of the two primes, every non-zero error term $e(X)$ allows a Bellcore-like attack. This depends on the message m. For randomly chosen messages m, the probability is $1/\min(p, q)$ according to Proposition 3.3.

3. **Attacks targeting the term $(S_q - S_p)$: Bellcore-like attack is possible.** If an adversary targets $(S_q - S_p)$, he might either induced a fault into any of the two values S_q and S_p or the result. In any case, such a fault yields $\tilde{s} = s + X \cdot \varepsilon$, for $\varepsilon \in \{e(S_p), e(S_q), e(S_q - S_p)\}$, depending on which of the three values has been hit. In Garner's Scheme (3.1), we have $X = p \cdot (p^{-1} \bmod q)$, hence, $X \equiv 0 \bmod p$ and $X \equiv 1 \bmod q$. Therefore, we always have $\tilde{s} \equiv s \bmod p$, as any fault is multiplied with zero modulo p. Modulo q however, any nonzero error will yield a faulty result $\tilde{s} \not\equiv s \bmod q$ unless $\varepsilon \equiv 0 \bmod q$. Proposition 3.2 shows that the condition $\varepsilon \equiv 0 \bmod q$ holds with probability 0 for the Single Bit Fault Model and for the Byte Fault Model, and with probability less than $2/\min(p, q)$ for the Random Fault Model. In all other cases, the adversary can recover the factors of N using

   $$p = \gcd(m - \tilde{s}^e, N).$$

4. **Attack targeting the computation of X: Bellcore-like attack is possible.** As noted before, X can be precomputed in advance, because it does not depend on the message m. However, long term memory needs to be specially secured to ensure that the values used are correct at all times. If one wants to save long term memory and compute X each time Algorithm 3.1 is invoked, an adversary can try to induce faults into another computation. In Garner's Scheme (3.1), the variable p or the computation of $p^{-1} \bmod q$ can be targeted. If the final product is affected by a fault, we have the same situation as described above for a fault induced into X.

   If p is faulty, the result is $\tilde{s} = s + e(p) \cdot (p^{-1} \bmod q) \cdot (S_q - S_p)$ in case of a transient fault. In order to apply the Bellcore attack, an adversary needs to ensure that this term is zero modulo exactly one of the two primes p and q with a sufficiently high probability. For bit

faults and byte faults, this is impossible according to Proposition 3.2, for random faults, the probability is less than $4/p$. However, any nonzero $e(p)$ allows a Bellcore attack if $(S_q - S_p)$ is zero modulo either p or q. This depends on the message m and it occurs with probability at most $1/\min(p,q)$ according to Proposition 3.3. For a permanent error, we get the same result with $\tilde{s} = s + e \cdot (S_q - S_p)$, where $e = (p + e(p)) \cdot ((p + e(p))^{-1} \bmod q) - p \cdot (p^{-1} \bmod q)$.

It is more successful to induce faults into $p^{-1} \bmod q$. Here, the effect of any nonzero fault will be eliminated modulo p, as it is multiplied by p. Modulo q, however, $p \cdot (p^{-1} \bmod q + e)$ will no longer yield 1. Hence, p can be recovered by $\gcd(s^e - m, N)$.

A fault induced into $p^{-1} \bmod q$ is also successful. If $(p^{-1} \bmod q)$ is attacked, the final faulty result is $\tilde{s} = s + e(p^{-1} \bmod q) \cdot p \cdot (S_q - S_p)$. Therefore, we have $\tilde{s} \equiv s \bmod p$ and $\tilde{s} \not\equiv s \bmod q$ unless $e(p^{-1} \bmod q) \cdot (S_q - S_p) \equiv 0 \bmod q$. For a random message m, the probability that $(S_q - S_p) \equiv 0 \bmod q$ is less than $2/q$. This follows from the proof of Proposition 3.3. Hence, p can be recovered by $\gcd(s^e - m, N)$.

If an adversary cannot target any of the variables precisely, which allow a successful Bellcore attack, he can run the algorithm multiple times. This approach is successful, as long as any of the variables, e.g., $(S_q - S_p)$ or $(p^{-1} \bmod q)$, is hit with a sufficiently high probability.

3.2. Countermeasures

All attacks presented in the last section only need a single faulty output $\tilde{s}$ to break the given instance of RSA. Hence, they are a very dangerous threat to devices using the CRT-RSA scheme. Because of the speed enhancement of CRT-RSA compared to plain RSA, the scheme has been widely deployed. Therefore, countermeasures are needed. In the following, we will give a brief overview over previously proposed countermeasures. We do not consider hardware countermeasures, since we aim at algorithmic protection mechanisms.

Full Verification. The most trivial countermeasure is to use full verification of the final result on the card. Given the public key e, the card could just check whether a computed signature s satisfies $s^e \equiv m \bmod N$. As long as no correlated faults can be induced, every fault attack is thwarted by this approach. However, it is only an efficient option if the the public key e is small. Otherwise, the computation time is doubled and the speed advantage of CRT-RSA is significantly reduced.

Doubled Computation. Another approach may be to compute a result twice, say s_1 and s_2 using the same data. If the two results are equal, s_1 is returned, otherwise, an error is detected. However, this approach has two drawbacks. First, the secret key d is usually very large, therefore, doubled computations suffer at least the same speed slowdown as full verification. Second, if an adversary manages to induce permanent faults into variables, both computations may use the same faulty variable and would not detect an error. Such a scenario might occur in the case of a naive implementation, which computes the message-independent value X used in Garner's scheme (3.1) beforehand. If the value $p^{-1} \bmod q$ is affected by a fault, a faulty value X would be used twice, yielding the same faulty result. The same considerations apply to schemes, where a result is computed three or more times to get a majority vote on a correct result.

Redundant Information. All system parameters are stored in ROM, where physical attacks are extremely difficult, as only destructive faults may change bits in ROM. However, such faults are possible and values stored in ROM must be loaded into the CPU before they are used. Hence, it is always possible that such values are being tampered with. Since all checking procedures must rely on system parameters, it is crucial to check the integrity of the used parameters during the computation. If error-correcting codes or CRC sums are used, permanent errors induced into system parameters can be easily detected. However, this does not prevent transient faults.

Shamir's Countermeasure. A more sophisticated countermeasure has been described by Shamir at the Rump session of EuroCrypt '97 (cf. [BDL01]) and in a patent [Sha99]. Shamir suggests to choose a small random integer r of about 32 bits, which can then be used to verify intermediate results with a high probability of detecting all faults. Shamir's algorithm is presented as Algorithm 3.4.

Algorithm 3.4: CRT-RSA Algorithm with Shamir's Countermeasure

Input: $\mathsf{m} \in \mathbb{Z}_\mathsf{N}$ the message
Output: $\mathsf{s} = \mathsf{m}^\mathsf{d} \bmod \mathsf{N}$
In Memory: the secret RSA key d, a random 32-bit integer r, the RSA modulus $\mathsf{N} = \mathsf{p} \cdot \mathsf{q}$, and both primes p and q

1 Set $\mathsf{d_{pr}} := \mathsf{d} \bmod \varphi(\mathsf{pr})$
2 Set $\mathsf{d_{qr}} := \mathsf{d} \bmod \varphi(\mathsf{qr})$
3 $\mathsf{S_p} := \mathsf{m}^{\mathsf{d_{pr}}} \bmod \mathsf{pr}$
4 $\mathsf{S_q} := \mathsf{m}^{\mathsf{d_{qr}}} \bmod \mathsf{qr}$
5 If $\mathsf{S_p} \not\equiv \mathsf{S_q} \bmod \mathsf{r}$ then return FAILURE
6 $\mathsf{s} := \mathsf{CRT}(\mathsf{S_p}, \mathsf{S_q})$

The additional costs of Shamir's countermeasure are negligible and the number of undetectable faults depends on the number r, which acts as a security parameter. An obvious choice for r is to use a 32-bit prime, which allows to compute $\varphi(pr)$ and $\varphi(qr)$ very fast. Moreover, d_{pr} and d_{qr} can be precomputed and stored on the card as in Algorithm 3.1. The major drawback of Shamir's method is, as already noted in [BDL01] and [ABF+02], that faults induced during the CRT step are not checked. This shows that any countermeasure must aim at checking the final result, it is not sufficient to check intermediate results only. Moreover, an explicit check yields a single point of failure, where attacks might easily circumvent countermeasures. We will elaborate more on this subject in Section 3.2.1. In [ABF+02], the authors try to enhance Shamir's countermeasure, mainly by adding a large number of explicit checking procedures. In [YMH03], however, it was shown that the resulting scheme is still not secure. As a consequence, a better countermeasure is needed, which protects the CRT step as well. We will present such a countermeasure in Section 3.3. But before, we introduce "infective errors".

3.2.1. Infective Errors

The main problem of the CRT-RSA algorithm is the extremely high susceptibility of the algorithm for faults. Any fault, which is induced into either S_p or S_q allows a complete break of the given instance of RSA. Hence, it is crucial for the security of the system, that such a situation is prevented. The aforementioned countermeasures aim at checking intermediate results explicitly and discarding them in favor of an error message if the check is not satisfied.

In [YKLM01b], later published in an extended version as [YKLM03], Yen, Kim, Lim, and Moon suggest a different approach. They promote the idea that a computation could ensure that the CRT computation will only result in values, which are either correct modulo both p and q, or which are wrong modulo both p and q, but never wrong modulo only one of the two primes. Therefore, any error in either S_p or S_q must "spread" into the other value and "infect" it with its fault. This is a concept of implicit checking procedures, which was named "fault infective CRT computation" in [YKLM03].

Trying to output faulty values seems counterintuitive at first. However, such an approach might overcome several drawbacks of explicit checking procedures. After all, if every error is guaranteed to result in an output, which does not leak secret information, such errors, if any, are good errors.

The main disadvantage of an explicit checking procedure is the fact, that it is usually relied on to be error-free. However, this assumption is questionable. In Shamir's countermeasure presented in Algorithm 3.4, the check in Line 5 usually depends on the zero flag of the CPU. Hence, an attack must change just a single bit to render this check useless. Although we assume in this thesis, that an adversary cannot mount two attacks during one run of an algorithm, the extreme susceptibility of the CRT-RSA scheme suggests to consider attacks, which are improbable, but may be possible by chance. Moreover, other explicit checking procedures may be susceptible to timing or power analyses. Infective computations do not raise any of these doubts. Therefore, they may serve as an excellent replacement for explicit checking procedures.

However, infective computations also have their limitations. They try to randomize the final result such that the adversary cannot derive any helpful values. However, since infective computations rely only on the induced error, the "amount of randomness" given by the induced fault must suffice to randomize the output. If the adversary uses a fault model, which induces errors from an efficiently sampleable set, e.g., the Single Bit Fault Model 1.7 or the Byte Fault Model 1.8, an adversary may do an exhaustive search for the correct error value. Details about such an approach will be presented in Section 3.6.

The authors of [YKLM03] propose two schemes, claimed to be secure against Bellcore attacks. However, this claim does not hold for either scheme. We will show this for their first proposed scheme only, since the results carry over to the second scheme as well.

The schemes in [YKLM03] use two dependent RSA keys, which we will refer to as a "main key" and a "small key". The keys are generated as follows:

Protocol 3.5 (CRT-1 Protocol from [YKLM03]: Key Generation).

System parameters:
- *Choose two large primes p and q and generate an RSA modulus $N = p \cdot q$.*
- *Fix a bound $B \ll N$.*

Key generation:
- *Choose a small RSA key:*
- *Choose a random $r < B$ and a random $e_r < B$ coprime to $\varphi(N)$, such that $\gcd(e_r, \varphi(N)) = 1$.*
- *Choose d_r such that $e_r \cdot d_r \equiv 1 \bmod \varphi(N)$.*
- *Choose a main RSA key:*
- *Let $d = d_r + r$. If $\gcd(d, \varphi(N)) \neq 1$ then choose a new small key.*
- *Otherwise, compute e such that $e \cdot d \equiv 1 \bmod \varphi(N)$.*

Public Key:
- (e, N)

The desired advantage of having two RSA key pairs from Protocol 3.5 is that a modular exponentiation can be computed with the small key, where a small public key allows to verify the computed result very efficiently. Since the main key is related to the small key, the equation $m^d = m^{d_r+r}$ allows to compute m^d from m^{d_r} quickly as well. Since the verification should not be carried out as an explicit checking procedure, the authors of [YKLM03] use these verifications implicitly to offer infective computations. They propose the following algorithm.

Algorithm 3.6: Infective CRT-RSA Computation, Proposal CRT-1 from [YKLM03]

Input: A message $\mathsf{m} \in \mathbb{Z}_\mathsf{N}$
Output: A signature $\mathsf{s} = \mathsf{m}^\mathsf{d} \bmod \mathsf{N}$
In Memory: The RSA modulus N, its factors p and q, the small RSA key $((\mathsf{e_r}, \mathsf{N}), \mathsf{d_r})$ and the integer r

\# Preliminaries
1 Set $\mathsf{k_p} := \lfloor \mathsf{m}/\mathsf{p} \rfloor$ and $\mathsf{k_q} := \lfloor \mathsf{m}/\mathsf{q} \rfloor$
\# Computation of the two signature parts
2 Set $\mathsf{S_p} := \mathsf{m}^{\mathsf{d_r}} \bmod \mathsf{p}$
3 Set $\hat{\mathsf{m}} := ((\mathsf{S_p}^{\mathsf{e_r}} \bmod \mathsf{p}) + \mathsf{k_p} \cdot \mathsf{p}) \bmod \mathsf{q}$
4 Set $\mathsf{S_q} := \hat{\mathsf{m}}^{\mathsf{d_r}} \bmod \mathsf{q}$
\# Combination of the signature
5 Set $\tilde{\mathsf{m}} := (\mathsf{S_q}^{\mathsf{e_r}} \bmod \mathsf{q}) + \mathsf{k_q} \cdot \mathsf{q}$
6 Set $\mathsf{s} := \mathsf{CRT}(\mathsf{S_p}, \mathsf{S_q}) \cdot (\tilde{\mathsf{m}}^{\mathsf{r}}) \bmod \mathsf{N}$
7 Return s

It can be easily seen that in a correct computation, we have $\hat{m} = m \bmod q$ and $\tilde{m} = m$ in Algorithm 3.6. Hence, the correct result is returned. If a fault occurs, the idea is that the final result will be sufficiently randomized, such that an attacker does not learn anything from a faulty output. To achieve this, the authors claim that an error induced into S_p in Line 2 will spread into S_q via $\hat{m}$ and that an error induced into S_q in Line 4 will spread into s via $\tilde{m}$.

However, the concept of infective computations has not been implemented to its full effectiveness. In particular, the presented scheme does not make all parameters infective. This leaves Algorithm 3.6 susceptible to an attack that resembles the classic Bellcore attack.

We will now show that both the key generation scheme presented as Protocol 3.5 and the actual signature scheme presented as Algorithm 3.6 have flaws, which show that both are insecure.

Flaw 1: Publishing Too Many Parameters

In [YKLM03, p. 466], the authors claim that the security of their schemes "does not require the integers e_r and r to be secret parameters." Unfortunately, this is not true as the following lemma from [BM04] shows.

Lemma 3.7 (The Parameters e_r And r Must Not Be Public).
Let (e, N) be a public RSA key generated by Protocol 3.5. If e_r and r, as defined by Protocol 3.5, are known, the modulus N can be factored in expected polynomial time.

Proof:

Consider the following set of equations:

$$\begin{aligned} & d \cdot e \equiv 1 \bmod \varphi(N) \\ \Rightarrow \quad & (d_r + r) \cdot e \equiv 1 \bmod \varphi(N) \\ \Rightarrow \quad & (1 + e_r \cdot r) \cdot e \equiv e_r \bmod \varphi(N) \\ \Rightarrow \quad & e \cdot (1 + e_r \cdot r) - e_r = k \cdot \varphi(N) \qquad \text{for some } k \in \mathbb{N} \end{aligned} \tag{3.2}$$

The complete left side of Equation (3.2) is public knowledge, therefore an unknown multiple of $\varphi(N)$ is known. Once a multiple of $\varphi(N)$ is known, $\varphi(N)$ can be computed in expected polynomial time using a well known probabilistic algorithm, which is briefly described in [MvOV96, §8.2.2] and in Algorithm 2.12. Given N and $\varphi(N)$, the factorization of N follows immediately since $\varphi(N) = N - p - N/p + 1$. □

Flaw 2: Generating Insecure Keys

Since publishing the parameters r and e_r compromises the security of the proposed system, we may assume that these parameters are secret. Unfortunately, even keeping these parameters secret will not secure the proposed schemes. This is due to an inappropriate choice of the two key pairs. In Protocol 3.5, the "small" RSA key pair is chosen in order to be able to implicitly check intermediate variables very efficiently. This efficiency is the reason for the insecurity of the complete key generation scheme.

Clearly, the number of possible keys is extremely limited. Since e_r is a small integer and r is a small integer, there are less than B^2 many pairs (e_r, r) and consequently at most B small key pairs (e_r, d_r). The main RSA key (e, d) depends on e_r and r, hence, there are less than B^2 many key pairs (e, d). The bound B is assumed to be very small, since it is intended for very efficient implicit checking of the intermediate parameters. If B is too small, an adversary may be able to guess the two values r and e_r with non-negligible probability. Intuitively, if a cryptosystem only offers a very small number of key pairs, this property might be susceptible to attacks. In the case of Protocol 3.5, the system can be broken by lattice attacks. This has been described by Blömer and May in [BM04]. It follows that given an RSA key (e, N) generated according to Protocol 3.5, the modulus N can be factored in expected polynomial time if $1 + e_r \cdot r \leq 1/3 \cdot N^{1/4}$ and $e_r \leq 1/2 \cdot c \cdot N^{1/4}$. Hence, the keys proposed in [YKLM03] are insecure. [YKLM03] proposes two different schemes, however, both use the same key generation protocol.

Flaw 3: Failure to Protect all Intermediate Variables

We have demonstrated that the key generation process is insecure. However, even if this issue could be fixed, and two secure pairs of RSA keys would be used, Algorithm 3.6 is not secure. The algorithm itself is still susceptible to fault attacks. While the computations of S_p and S_q are infective, other values, i.e., k_p and k_q, are not infective. This fact allows a successful Bellcore-like attack.

Let us consider the Arbitrary Fault Model presented in Definition 1.10. This assumes a very weak adversary and it can be replaced by any other fault model. Here, an adversary is assumed to be able to induce an unknown error into a specific line of code. Let us assume that the attacker targets the computation of k_q in Line 1 of Algorithm 3.6. The only time this variable is used is in Line 5, hence, we do not need to differentiate between transient and permanent faults in k_q. In

Line 5, k_q is multiplied with q, therefore, this term vanishes modulo q. This implies that if k_q is faulty, we have $\tilde{m} \equiv m \bmod q$, but $\tilde{m} \not\equiv m \bmod p$. The final result $s = \mathrm{CRT}(S_p, S_q) \cdot \tilde{m}^r \bmod N$ is therefore equal to $m^{d_r+r} = m^d \bmod q$, but not equal to $m^d \bmod p$. Consequently, we have $q = \gcd(s^e - m, N)$, which represents the Bellcore attack. The attack also applies to the second algorithm proposed in [YKLM03]. This attack does not yield the same result if the value k_p is attacked, since $\hat{m}$ is computed modulo q.

Flaw 4: Failure to Protect Against Oracle Attacks

Let us assume again that an adversary is able to induce faults according to the Arbitrary Fault Model 1.10. Since the values of k_p and k_q depend on m and a secret factor of N, this setting can be used for an oracle attack as defined in Section 1.6.1 together with chosen messages. If $m < p$, we have $k_p = 0$. Therefore, an adversary may try to induce a fault affecting the variable p used in Line 3 of Algorithm 3.6. In this case, any error will be eliminated by the multiplication with $k_p = 0$. However, we need to ensure that p is attacked after the computation of $S_p^{e_r} \bmod p$, which must still be correct. After that computation, p is used last in the second summand in Line 3, hence, transient errors as well as permanent errors have the same effect. Since a fault affecting p only yields a faulty value $\hat{m}$ if $m < p$, this allows for a binary search yielding p. An adversary will choose increasing values for $m < p$ until he knows p. For every choice of m, the adversary induces faults into p in Line 3. If this leads to any faulty behaviour — a faulty result or an error message — the adversary knows that $m \geq p$, otherwise, he knows that $m < p$. This allows to recover p using $O(\log(p))$ many runs of Algorithm 3.6 while inducing faults. The same result also applies to attacks, which try to induce faults affecting q in Line 5.

The proposed attack assumes a weak adversary, since any fault will yield the desired result. However, an adversary needs to be able to induce $O(\log(p))$ faults in subsequent runs of the algorithm. He must also be able to choose the input m, which is different from other attacks, where he is just assumed to know m. Hence, this setting is stronger.

It might be argued that since p is a variable stored in long term memory, it is harder to target than inputs or intermediate variables. However, since p must be loaded into the CPU at least once, faults induced into bus lines and cache memory are possible for p and q as well. If the fault is permanent, however, the smartcard could detect the attack by checking p and q after their use in Lines 3 and 5.

The proposed oracle attack is also possible if an adversary uses a fault model based on the random fault type with an asymmetric distribution. Assume that an adversary can induce single bit faults, such that an affected bit is reset to 0 with a significantly higher probability than set to 1. In this case, an adversary can target the variable k_p at any time prior to its use in Line 3. If $m < p$, we have $k_p = 0$ and attacks with the asymmetric random fault type will be more likely to leave k_p unchanged than inducing a fault. Several faulty final results collected with the same setting will allow the adversary to conclude whether a chosen input message m is smaller than p or not. With an increased number of runs per input, this attack also works if a bounded number of bits is attacked with an asymmetric random fault type. This result also applies to k_q, and the second scheme proposed in [YKLM03] is also susceptible to an oracle attack in the described way.

The proposed binary search approach has been imagined by the authors of [YKLM03] themselves for timing attacks. However, we have shown that this approach is also possible for fault attacks. A blinding technique suggested in [YKLM03] as a countermeasure against a binary search approach for timing attacks does not thwart our fault attack, since the blinding is done

with a multiple of p or q respectively.

Flaw 5: Failure to Protect the CRT Combination

In [YKLM03], the authors do not explicitly require $\tilde{m}$ to be computed prior to the CRT combination. The order of the operations as presented in Algorithm 3.6 is intruding. However, this leaves the CRT step unprotected. If any attack on the CRT combination yields a value, which is correct modulo exactly one of the two primes p and q, the final multiplication by $\tilde{m}^r$ in Line 6 will not prevent an adversary to mount a successful Bellcore-like attack.

In [YKLM03], the authors promote the use of Garner's Scheme (3.1) and insist that $X = p \cdot \left(p^{-1} \bmod q\right)$ be precomputed and stored on the card. Otherwise, any fault induced into $p^{-1} \bmod q$ would allow a successful Bellcore attack. Unfortunately, an adversary may also try to attack the value S_q used in Garner's Scheme. In this case, a Bellcore-like attack is possible as well.

Summary. The five flaws presented above show that developers of cryptosystems must be careful not to trade security for efficiency. For fault attacks, one must not rely on any computation to be immune against fault attacks. In general, any line of code and any variable used may be affected by a fault. Although Algorithm 3.6 proved insecure against fault attacks, the concept of infective computations is very promising. Therefore, we will use this concept to develop a new, secure countermeasure for CRT-RSA modular exponentiations in the following section.

3.3. A New Countermeasure Against Random Faults

The drawbacks of Shamir's small prime verification countermeasure explained in Section 3.2 and the insecure infective computation approach presented in Section 3.2.1 show that better methods are needed. As Shamir's basic idea is very promising, we extend this countermeasure to the whole CRT-RSA computation. We also use infective computations as introduced in Section 3.2.1 to eliminate the single point of failure of a checking step. We propose the following new algorithm, which is secure against the Bellcore attack for the most realistic fault model. We will prove this claim in the remainder of this section. The results of this section have been published as a joint work together with Johannes Blömer and Jean-Pierre Seifert as [BOS03].

Algorithm 3.8: Infective CRT-RSA

Input: A message $\mathsf{m} \in \mathbb{Z}_\mathsf{N}$
Output: $\mathsf{Sig} := \mathsf{m}^\mathsf{d} \bmod \mathsf{N}$ or a random number in $\mathbb{Z}_\mathsf{N}$
In Memory: $\mathsf{p} \cdot \mathsf{t}_1, \mathsf{q} \cdot \mathsf{t}_2, \mathsf{N}, \mathsf{N} \cdot \mathsf{t}_1 \cdot \mathsf{t}_2, \mathsf{d}_\mathsf{p}, \mathsf{d}_\mathsf{q}, \mathsf{t}_1, \mathsf{t}_2, \mathsf{e}_{\mathsf{t}_1}$, and $\mathsf{e}_{\mathsf{t}_2}$

1 Let $\mathsf{S}_\mathsf{p} := \mathsf{m}^{\mathsf{d}_\mathsf{p}} \bmod \mathsf{p} \cdot \mathsf{t}_1$
2 Let $\mathsf{S}_\mathsf{q} := \mathsf{m}^{\mathsf{d}_\mathsf{q}} \bmod \mathsf{q} \cdot \mathsf{t}_2$
3 Let $\mathsf{S} := \mathsf{CRT}(\mathsf{S}_\mathsf{p}, \mathsf{S}_\mathsf{q}) \bmod \mathsf{N} \cdot \mathsf{t}_1 \cdot \mathsf{t}_2$
4 Let $\mathsf{c}_1 := (\mathsf{m} - \mathsf{S}^{\mathsf{e}_{\mathsf{t}_1}} + 1) \bmod \mathsf{t}_1$
5 Let $\mathsf{c}_2 := (\mathsf{m} - \mathsf{S}^{\mathsf{e}_{\mathsf{t}_2}} + 1) \bmod \mathsf{t}_2$
6 Let $\mathsf{Sig} := \mathsf{S}^{\mathsf{c}_1 \cdot \mathsf{c}_2} \bmod \mathsf{N}$
7 **Output** Sig

The basic algorithm for CRT-RSA consists of three steps, the computation of the two parts S_p and S_q and their combination to the signature S using the CRT. We modify the computation of all three values implementing a variant of Shamir's idea. Then we introduce a detection mechanism that is not required to be error free in order to prevent a fault attack on the whole smartcard. If an error was induced at any step of the algorithm, this countermeasure will change the final result in a way unpredictable to an adversary. The resulting algorithm looks extremely simple, but it proves to be very effective in the most practical attack model assuming the strongest adversary.

Selecting The Parameters. As a precomputation step that can be performed for any smartcard at production time, generate a valid RSA key with (N, e), $N = p \cdot q$, as the public key and d as the corresponding private key satisfying $e \cdot d \equiv 1 \bmod \varphi(N)$.

Additionally, select two integers t_1 and t_2 of sufficiently large bitlength to withstand exhaustive search (see Section 3.4.3 for concrete suggestions) which must satisfy several conditions in order to allow a secure scheme:

1.	t_1 and t_2 must be coprime
2.	$gcd(d, \varphi(t_1)) = 1$ and $gcd(d, \varphi(t_2)) = 1$
3.	t_1 and t_2 are squarefree
4.	$t_i \equiv 3 \bmod 4$ for $i \in \{1, 2\}$
5.	$t_2 \not\mid X = pt_1 \cdot ((pt_1)^{-1} \bmod qt_2)$

Note that the use of two small moduli instead of a single one has already been described in [Sha99] and in [YKLM03], but for a different use. Let $d_p := d \bmod \varphi(p \cdot t_1)$, $d_q := d \bmod \varphi(q \cdot t_2)$. Afterwards, compute two corresponding public keys e_{t_1} and e_{t_2} such that $d \cdot e_{t_i} = 1 \bmod \varphi(t_i)$. Store $p \cdot t_1$, $q \cdot t_2$, N, $N \cdot t_1 \cdot t_2$, d_p, d_q, t_1, t_2, e_{t_1} and e_{t_2} on the smartcard. It is easy to see that the algorithm computes the correct signature if no error occurs. In this case, the two infective variables c_1 and c_2 computed in Lines 4 and 5 are both equal to 1, hence, $\text{Sig} \equiv S \equiv m^d \bmod N$.

Selecting The Small Primes t_1 and t_2. Let us comment on the five conditions for the small primes t_i:

1. *Condition* 1 is needed to ensure that the CRT combination of S_p and S_q works, because it requires the two moduli to be coprime. There is also a variant of the Chinese Remainder Theorem, where this condition is not required (cf. [FS02]), however, choosing primes is practical as we will show later and if both moduli are coprime, the standard CRT can be used. This is the better choice as it requires a smaller number of modifications of the plain CRT-RSA algorithm, Algorithm 3.1.

2. *Condition* 2 is required to ensure that the small keys e_{t_i} $(i = 1, 2)$ can be generated.

3. *Condition* 3 must hold, because otherwise the equation $m^{d \cdot e_{t_i}} \equiv m \bmod t_i$ may not hold. This condition can be further relaxed: we have $m^{d \cdot e_{t_i}} \equiv m$ already if

$$\gcd\left(\frac{t_i}{\gcd(t_i, m)}, \gcd(t_i, m)\right) = 1.$$

   However, we suggest choosing primes for both t_i.

4. *Condition* 4 ensures a good resistance against attacks on the exponents d_p and d_q. The security analysis will further explain this condition.

5. Finally, *Condition* 5 provides security against attacks on the CRT combination (see the analysis of attacks on Line 3 below).

Summarizing the five conditions, it is an obvious idea to choose both t_1 and t_2 to be strong primes. Not all primes satisfy Condition 4. Section 3.3.1 will show that choosing both t_i to be strong primes is indeed a useful recommendation.

3.3.1. Efficiency Of The New Algorithm

We will first show that the new algorithm is indeed an efficient algorithm to compute RSA signatures. The efficiency depends on the keys t_i. The additional costs compared to the plain CRT-RSA method are mainly an increased number of operations in Lines 1 and 2 due to the larger moduli. This increases the size of the exponent and of the intermediate results. They have to be computed in a larger group now. The additional costs of the CRT (due to a larger modulus) and the costs of the two modular exponentiations modulo t_1 and t_2 (in Lines 4 and 5) do not contribute to the overall costs significantly. If we choose an RSA modulus N of 1024 bits, t_1 and t_2 such that $l(t_i) \in \{60, 80\}$, and assume quadratic time complexity for the modular exponentiations in $\mathbb{Z}_{pt_1}$ and $\mathbb{Z}_{qt_2}$, the savings compared to the plain RSA scheme is lowered to 1/3 instead of 1/4 for the unsecured CRT-RSA scheme, Algorithm 3.1. In our Algorithm 3.8, the intermediate values are of length about 1/1.7 instead of 1/2 compared to N. For the next generation of RSA key recommendations, i.e., using a 2048 bit modulus to compare to a 112-bit exhaustive search security level [Kal03], the small primes have to have at least a bit length of 112 bits, which cause intermediate results to be of length 1/1.8 compared to a 2048-bit N. This represents a speedup of 3.25 compared to plain repeated squaring. Increasing bit sizes narrow the gap between the original performance advantage of CRT-RSA and the performance advantage of the secure CRT-RSA algorithm, Algorithm 3.8. Summarizing the above considerations, Algorithm 3.8 still offers a significant speedup compared to plain repeated squaring.

Efficiently Computing t_1 and t_2. For the key generation process, which is usually performed on the card at production time, we can efficiently find suitable candidates for t_1 and t_2. First, a valid RSA key pair $((e, N), d)$ is chosen, with $N = p \cdot q$. Then the two small moduli t_1 and t_2 are generated. We would like to emphasize the fact that neither the algorithm nor the small moduli impose any restrictions on the choice of the main RSA key, no special structure or generation process is required. Hence, any valid RSA key pair can be used for $((e, N), d)$.

We suggest choosing both t_i as different strong primes, i.e., $(t_i - 1)/2$ are also primes. As $t_i \ll p, q$ this obviously satisfies Conditions 1, 3, and 4. Condition 2 is not satisfied for a fixed t_i in a fraction of about $1/t_i$ cases. Therefore given $l(t_i)$ large enough, this probability is negligible. For randomly chosen t_i, the values pt_1 and qt_2 are independent, therefore $(pt_1)^{-1} \bmod qt_2$ is almost uniformly distributed in $\mathbb{Z}_{qt_2}$. As $t_2 \nmid pt_1$, this means that the probability that a chosen t_2 does not satisfy Condition 5, i.e., $t_2 \nmid X = pt_1 \cdot ((pt_1)^{-1} \bmod qt_2)$, is at most $1/t_2$. Hence, we expect very few strong prime choices. Since the density of strong primes is conjectured to be asymptotically $D \cdot x/\log^2(x)$ in [HL22], the task of finding suitable t_i is easy. Here, $D \approx 0.6601618$ is the twin prime constant.

Moreover, it is also possible to use a modified CRT combination that can handle the case $t_1 = t_2$ [FS02]. Hence, it suffices to choose a single t. However, as explained above, we choose two in order to use the standard CRT.

3.4. Security Analysis of the Proposed Countermeasure

All parts of our analysis can be rigorously proved. Only the analysis for one single variable relies on the following heuristically justified assumption.

Assumption 3.9 (Distribution of Certain Div Operations).
For an RSA modulus $N = p \cdot q$, d a secret key, $m \in \mathbb{Z}_N$ a given message, $t \ll p$ a prime, $e(\cdot)$ a random error as defined above, the value

$$\alpha := \Big(m^d \text{ div } (pt + e(pt))\Big) \bmod t$$

can be seen as a random variable uniformly distributed in $\mathbb{F}_t$.

Clearly, α only has a chance to fulfill the assumption correctly if $2^{l(pt)}$ is a multiple of t. However, the assumption is justified for other cases as well, because the distance to the uniform distribution is negligibly small as $t \ll 2^{l(pt)}$.

3.4.1. Undetectable Errors.

For the security analysis, we need to investigate the probability of any induced error to circumvent our countermeasure and result in an undetectable error. We are only concerned with errors that cause the final signature to be correct modulo p but false modulo q (or vice versa), in which case the classic Bellcore attack can be applied. Otherwise, no exploits of specific errors in a faulty CRT-RSA signature are known yet. In our analysis, we do not look at combinations with other side channel attacks like timing or power attacks.

As a consequence, we do not offer a general security proof or security reduction. We do not claim that the proposed algorithm is secure against any type of fault attack. Research in fault attacks has not yet been able to establish a mathematical framework to offer complete security proofs. Therefore, we concentrate on proving that for a given fault model, certain exploits are no longer possible. We have discussed the current situation concerning the search for a mathematical framework for fault security proofs in the introduction.

The checking mechanism in Lines 4 and 5 of Algorithm 3.8 is done via a small modulus, hence, undetectable errors are introduced into the system. However, we will show that the number of these errors is negligibly small. Therefore, they pose no threat to the security of the system.

An error will slip by Lines 4 and 5 undetected if it is eliminated by the modular reduction. If $\tilde{S}$ is a faulty value for S, then if $\tilde{S} = S + k \cdot t_1 \cdot t_2$, $k \in \mathbb{Z}$, both modular reductions in Lines 4 and 5 will fail to detect this error and set $c_1 = c_2 = 1$. For other values of $\tilde{S}$, it is $S' \not\equiv S \bmod t_i$ for $i = 1$ or $i = 2$. Hence $\tilde{S}^{e_{t_i}} \not\equiv m \bmod t_i$, which in turn forces $m - \tilde{S}^{e_{t_i}} > 0 \bmod t_i$. Therefore, it is $c_i \neq 1$. This observation is independent of the type of error.

We will first determine the probability that an error induced into a specific variable used in Algorithm 3.8 yields an undetectable fault (Section 3.4.1). This probability depends on the fault model used, which we will describe in the following. However, this is not enough to provide security. In Section 3.4.3, we will show that the values c_1 and c_2 must be secret values as well

to guarantee security. This implies that an adversary must not be able to guess these values efficiently. For our fault model, we will show that this condition holds.

As the fault model, we assume the Random Fault Model 1.9. Here, the strongest possible adversary capable of using the bit set or reset fault type stands against the best protected card. It is therefore the most realistic scenario. We will show the security against Bellcore attacks using this fault model in the following. For stronger fault models, given by less protected cards, we can also achieve security. However, such a fault model requires a small modification of Algorithm 3.8 (see Section 3.6). The Random Fault Model 1.9 assumes that an affected value x is replaced by a random bit string $\tilde{x} = x + e(x)$ as defined in Section 1.4.3. For simplicity, we only assume a single fault. However, as long as multiple errors are uncorrelated, the results are the same. It is virtually impossible to correlate the effects of faults on smartcards that use hardware countermeasures.

The two special inputs $m = 0$ and $m = 1$ will prove dangerous to the algorithm in the presence of some faults. Later, we will describe several enhancements to our scheme such that also these two messages cannot be exploited in a fault attack.

In the following analysis of attacks targeting each variable, the term *success probability* always refers to the success probability for a random fault induced into the affected variable to result in an undetectable error such that the Bellcore attack can be applied.

Faults Induced into Line 1 And 2

- **Attack targeting the stored variable d_p**
  The success probability for a fault induced into d_p is at most $3/t_1$ for messages $m \not\equiv \pm 1 \bmod t_1$. This probability is taken over the errors. A fraction of at most $3/t_1$ of all messages satisfies $m \equiv \pm 1 \bmod t_1$. The message $m = 1$ is secure. For other messages $m \equiv \pm 1 \bmod t_1$, $m \neq 1$, the success probability is at least $1/2$.

  Comment. Although this analysis seems to prove the algorithm insecure, this is not the case. Any adversary capable of constructing malicious instances of messages, i.e., an $m \equiv \pm 1 \bmod t_1$, needs to know t_1. But this parameter is secret. Hence, the adversary has no information on how to construct m. Therefore, the best he can do is to randomize the inputs. And since the number of malicious messages is less than $3/t_1$, this is a secure situation.

  Proof:

  Let $m \not\equiv \pm 1 \bmod t_1$. Given m, an error $e(d_p)$ leads to an undetectable error if

  $$m^{e(d_p)} \equiv 1 \bmod t_1. \tag{3.4}$$

  We need to analyze how many $e(d_p)$ exist at most with (3.4). Consider $\gcd(e(d_p), t_1 - 1)$. Since t_1 is a strong prime, we get $\gcd(e(d_p), t_1 - 1) \in \{1, 2, (t_1 - 1)/2, t_1 - 1\}$. Any $e(d_p)$ with $\gcd(e(d_p), t_1 - 1) \in \{1, 2\}$ can be written as $e(d_p) = 2^l \cdot b$, b odd and $\gcd(b, t_1 - 1) = 1$. For these $e(d_p)$, (3.4) implies $m^{2^l} \equiv 1 \bmod t_1$. Next, since t_1 is a strong prime, the equation $x^{2^l} \equiv 1 \bmod t_1$ has only the solutions $x \equiv \pm 1 \bmod t_1$. We conclude that for $m \not\equiv \pm 1$ and $\gcd(e(d_p), t_1 - 1) \in \{1, 2\}$, no error will be undetectable. Hence it remains to bound the number of $e(d_p)$ with $\gcd(e(d_p), t_1 - 1) \in \{(t_1 - 1)/2, t_1 - 1\}$ and $m^{e(d_p)} \equiv 1 \bmod t_1$. The worst case for m is if $m^{(t_1-1)/2} \equiv 1 \bmod t_1$,

in which case any $e(d_p)$ with the property that $e(d_p)$ is a multiple of $(t_1-1)/2$ leads to an undetectable fault. Since $e(d_p) \in \{-d_p, \ldots, 2^{l(d_p)+1}-1\}$, the number of $e(d_p)$ with $\gcd(e(d_p), t_1-1) \in \{(t_1-1)/2, t_1-1\}$ is a fraction of at most $2/(t_1-1) < 3/t_1$ of all possible $e(d_p)$.

Let us now determine the number of messages $m \equiv \pm 1 \bmod t_1$. As the messages are in $\mathbb{Z}_N$, there are at most $2 \cdot \lfloor N/t_1 \rfloor + 2$ messages satisfying the condition $m \equiv \pm 1 \bmod t_1$. This is a fraction of less than $3/t_1$ of all possible messages.

Now let $m \equiv 1 \bmod t_1$. If $m = 1$, then $m^{e(d_p)} = 1$ and $S_p = 1$. Hence, the error has no effect. Otherwise, every fault will cause an undetectable error, because $m^{e(d_p)} \equiv 1 \bmod t_1$ independent of the error $e(d_p)$.

Let $m \equiv -1 \bmod t_1$. The probability that a random fault causes an undetectable error is at least $1/2$, since every even $e(d_p)$ yields $m^{e(d_p)} \equiv 1 \bmod t_1$. In addition, if $e(d_p)$ is invertible modulo (t_1-1), the same considerations as above apply. This increases the success probability further. □

- **Attack targeting the stored variable pt_1**
  The success probability for a fault induced into pt_1 is at most $2/t_1$. This result is based on Assumption 3.9. The probability is taken over random choices of the error.

Proof:

If the modulus is randomly changed to $pt_1' = pt_1 + e(pt_1)$, write $m^d = \alpha_0 \cdot (pt_1 + e(pt_1)) + \alpha_1$ with $\alpha_1 < pt_1 + e(pt_1)$. The correct result S_p is now $S_p = \alpha_0 \cdot e(pt_1) + \alpha_1 \bmod pt_1$, while the faulty result S_p' is α_1. An undetectable error happens, if $S_p \equiv S_q \bmod t_1$, hence if $\alpha_0 \cdot e(pt_1) + \alpha_1 \equiv \alpha_1 \bmod t_1$. This is equivalent to $\alpha_0 \cdot e(pt_1) \equiv 0 \bmod t_1$.

As t_1 is a prime, t_1 has to divide at least one of the two factors. Hence, we need to compute the probability of $0 \equiv e(pt_1) \bmod t_1$ and of $0 \equiv \alpha_0 = m^d \operatorname{div} (pt_1 + e(pt_1)) \bmod t_1$. As $e(pt_1)$ is a uniformly distributed integer in a contiguous interval and α_0 is uniformly distributed by Assumption 3.9, the success probability is at most $1/t_1$ for each factor, and altogether at most $2/t_1$. This probability is taken over random choices of the error. □

- **Attack targeting m or the exponentiation's intermediate variable**
  The success probability for a fault induced during the exponentiation is at most $2/t_1$ for messages $m \not\equiv 0 \bmod t_1$. This probability is taken over the errors. For messages $m \equiv 0 \bmod t_1$, all faults yield an undetectable error. A fraction of at most $1/t_1$ of all messages m satisfies $m \equiv 0 \bmod t_1$.

Comment. There are many possible ways to compute $m^{d_p} \bmod pt_1$. Algorithm 3.10 presents a timing and simple power attack secure version of the well-known square-and-multiply algorithm (cf. [Cor99], [CKN00]). The result holds for other exponentiation algorithms as well.

Algorithm 3.10: Left-to-Right Square-And-Multiply-Always, Variant 2

Input: A message $m \in \mathbb{Z}_N$, a secret key $3 \leq d$, where $l(d)$ denotes the binary length of d, i.e., the number of bits of d, and a modulus pt_1
Output: $m^d \bmod pt_1$
\# **init**
1 Set $y := m^2 \bmod pt_1$
\# **main**
2 For i from $l(d) - 2$ downto 1 do
3 Set $y_0 := y$
4 Set $y_1 := y \cdot m \bmod pt_1$
5 Set $y := y_{d_i}^2 \bmod pt_1$
6 Set $y_0 := y$
7 Set $y_1 := y \cdot m \bmod pt_1$
8 Set $y := y_{d_0} \bmod pt_1$
9 **Output** y

Again, some messages are malicious, but similar to the reasoning before, the adversary can gain no advantage from this fact as he cannot choose m accordingly. Faults induced into y_0 and y_1 resemble the same situation — if they get incorporated into the computation at all. If the modulus pt_1 is affected by a fault during the exponentiation, the resulting scenario is equivalent to that analyzed above for a fault targeting pt_1 directly.

Proof:

If Algorithm 3.10 suffers a fault at the time when $i = l$, and the intermediate value y is altered, we have $S'_p = (y + e(y))^{2^{l-1}} \cdot m^w$ for $w = \sum_{i=1}^{l-1} d_i \cdot 2^i$. Hence messages $m \equiv 0 \bmod t_1$ lead to $S'_p \equiv S_p \bmod t_1$ and to an undetectable error in Line 4 of Algorithm 3.8. There are at most $1/t_1$ of all possible messages satisfying this condition.

For messages $m \not\equiv 0 \bmod t_1$, we analyze the probability of $(y+e(y))^{2^{l-1}} \equiv (y)^{2^{l-1}} \bmod t_1$. First consider the case $(y + e(y)) \equiv 0 \bmod t_1$. This implies $(y)^{l-1} \equiv 0 \bmod t_1$ as well. Since y is of the form m^x for some x, this in turn implies $m \equiv 0 \bmod t_1$, which is impossible.

From now on we assume $y + e(y) \not\equiv 0 \bmod t_1$. Then $(y + e(y))^{2^{l-1}} \equiv (y)^{2^{l-1}} \bmod t_1$ implies $1 = (y/(y + e(y)))^{2^{l-1}} \bmod t_1$. Since $t_1 \equiv 3 \bmod 4$, this is equivalent to $1 = (y/(y + e(y)))^2$, which in turn implies $\pm 1 = (y/(y + e(y)))$. For any fixed y, there are exactly two choices of $e(y)$ that satisfy this equality. Hence, in case $m \not\equiv 0 \bmod t_1$ we can bound the success probability by $2/t_1$. □

- **Attack targeting the result S_p**
  The success probability for a fault induced into S_p is at most $1/t_2$ for messages $m \neq 0, 1$. Again, the probability is taken over the error.

  This case will be analyzed while considering attacks on the CRT combination in line 3.

Faults Induced into Line 3

Line 3, the CRT combination, may also be successfully targeted in an attack. We assume that $S = S_p + X \cdot (S_q - S_p) \bmod N \cdot t_1 \cdot t_2$ with $X = pt_1 \cdot ((pt_1)^{-1} \bmod qt_2)$. Here X is a precomputed value stored on the smartcard.

- **Attack targeting the result S, or the two addends S_p and $X \cdot (S_q - S_p)$**
  The success probability for a fault induced into S, S_p, or $X \cdot (S_q - S_p)$ is at most $1/(t_1 \cdot t_2)$. The probability is taken over random errors only, it is independent from the chosen message.

  Proof:

  If the result S is faulty, then $S' = S + e(S)$. This would circumvent the countermeasure iff $e(S) \equiv 0 \bmod t_i$ for both i. Because both t_i are different primes, this means that $e(S) \equiv 0 \bmod t_1 \cdot t_2$ must hold. As the error $e(S)$ comes from a contiguous interval, this probability is at most $1/(t_1 \cdot t_2)$. The same result holds for faults induced into the two summands S_p and $X \cdot (S_q - S_p)$. □

- **Attack targeting X**
  The success probability for a fault induced into X is at most $1/(t_1 \cdot t_2)$ for all but a fraction of $2/\min(t_1, t_2)$ messages. This probability is taken over random choices of the error.

  Proof:

  The proof is very similar to Proposition 3.3. Assume that an adversary induced a fault into the value X in the CRT combination $S = S_p + X \cdot (S_q - S_p) \bmod N t_1 t_2$ with $X = pt_1 \cdot ((pt_1)^{-1} \bmod qt_2)$. X is assumed to be precomputed and stored on the card. A random fault induced into X will result in a faulty value $\tilde{S}$ instead of S:

$$\begin{aligned} \tilde{S} &= S_p + X \cdot (S_q - S_p) + e(X) \cdot (S_q - S_p) \bmod N t_1 t_2 \\ &= S + e(X) \cdot (S_q - S_p) \bmod N t_1 t_2, \end{aligned}$$

  with $e(X) \in \{-X, \ldots, 2^{l(X)} - 1 - X\}$. The additional term is the induced error. The countermeasure of Algorithm 3.8 will fail to detect this fault iff the addend is a multiple of both t_1 and t_2, i.e. if $t_1 \cdot t_2 | e(X) \cdot (S_q - S_p) \bmod N t_1 t_2$ because both t_1 and t_2 are different primes. The latter property also implies that at least one of the factors must be a multiple of t_1 and one (possibly the same) a multiple of t_2.

  As we consider the security independent from the adversary's choices for m, we first assume that neither t_1 nor t_2 divides $(S_q - S_p)$. As $e(X)$ is an equally distributed value from a consecutive interval, and t_1 and t_2 may be seen as independent values, the probability for $t_1 | e(X)$ and $t_2 | e(X)$ is at most $1/(t_1 \cdot t_2)$.

  For the message dependent question whether any of the primes t_i divides $(S_q - S_p)$, let $S_q := c$ be fixed first (with $0 \leq c < qt_2$). In this case, there are pt_1 integers in $[c - pt_1 + 1, c]$. Of these numbers, only multiples of t_1 are counted. Hence, there are at most $\lfloor (pt_1)/t_1 \rfloor = \lfloor p \rfloor$ many such integers. Therefore, the probability of getting such an integer is less or equal to $p \cdot 1/(pt_1) = 1/t_1$. If we now count the overall number of possible integers for all choices of c, we determine

$$\begin{aligned} &\Pr[(S_q - S_p) = k \cdot t_1] \\ &= \sum_{c=0}^{qt_2 - 1} \Pr[(S_q - S_p) = k \cdot t_1 | S_q = c] \cdot \Pr[S_q = c] \\ &= \sum_{c=0}^{qt_2 - 1} \Pr[(c - S_p) = k \cdot t_1 \text{ for some } k] \cdot \frac{1}{qt_2} \end{aligned}$$

$$\leq \frac{1}{qt_2} \cdot qt_2 \cdot \frac{1}{t_1} = \frac{1}{t_1}.$$

As the same consideration holds for t_2, we have a maximum of $2/\min(t_1, t_2)$ messages where the probability that a random error is not detected is significantly higher than $1/(t_1 \cdot t_2)$. Moreover, the error $e(X)$ also needs to be a multiple of either p or q in order to apply the Bellcore attack. □

Comment. As with attacks on d_p, the adversary has no information on how to construct a message that yields $S_q - S_p$ to be a multiple of t_1, t_2 or both. His best choice is to choose random messages, which only gives him a negligible success probability. Hence, this attack is not promising to an adversary.

- **Attack targeting S_p or S_q or $(S_q - S_p)$**
  The success probability for a fault induced into S_p or S_q is less than $1/t_2$. This probability is taken over random choices of the error only.

Proof:

If an adversary targets either S_p or S_q in the second summand, the output of the CRT combination is $\tilde{S} = S + e(S_p) \cdot X$ (or $e(S_q)$ respectively). This results in an undetectable error if $t_1 \cdot t_2 | e(S_p) \cdot X$. As t_1 and t_2 are primes, this means that both t_i have to divide at least one factor. As t_1 always divides $X = pt_1 \cdot ((pt_1)^{-1} \bmod qt_2)$ and t_2 never divides X by Condition 5, this may only happen if $t_2 | e(S_p)$. Because $e(S_p)$ is uniformly distributed over an interval of consecutive numbers, the success probability is at most $1/t_2$. The same reasoning holds for $(S_q - S_p)$. □

Comment. If t_2 is not chosen carefully to prevent $t_2 | X$, the success probability is increased with the probability to meet $t_2 | X$. This is independent from the error, therefore any error $e(S_p) \neq 0$ would be harmful. This explains Condition 5 of the condition list for selecting t_1 and t_2.

Faults Induced into Lines 4 – 6

We also need to investigate the possibility to induce faults into the detection mechanism, Lines 4 – 6. But faults occurring during the computation of c_i are in vain unless another fault has been induced already. If a random fault is induced into a correct c_i, we have $c_i \neq 1$ and the final signature will look like a random value, i.e., be unpredictable. The same consideration applies to Line 6.

3.4.2. Excluding the Two Messages $m = 0$ and $m = 1$.

The analysis shows that choosing $m \in \{0, 1\}$ leads to a malicious message as $m \equiv 0, 1 \bmod t_i$ in these cases as well. The choice $m = 0$ or $m = 1$ is useful for an adversary in attacks targeting m, the values S_p or S_q, or the intermediate results of the exponentiation. Therefore, these two messages must be treated separately. For all other cases, the adversary's ability to create malicious messages implies knowledge about t_i. As we assume these parameters to be secret, the adversary has no better choice than to choose m at random. This leaves him with a success

probability of at most $1/t_i$. Hence, t_i is a security parameter and can be chosen large enough to effectively prevent efficient attacks.

Now let us explain several methods dealing with the case $m \in \{0, 1\}$. The first method is to use padding schemes. In fact, almost any padding scheme, deterministic or randomized, will ensure that $m = 0$ and $m = 1$ will either not be signed at all or will only be signed with negligible probability. The same effect can be achieved if hashing of the original message is done on the card and not prior to submitting the hash value to be signed to the card. However, as explained in the introduction to Chapter 2, most smartcard certification authorities require that a smartcard implements a pure RSA signature algorithm that is secure without using OAEP or similar padding schemes.

To avoid padding schemes, one can modify the message used in Lines 1 and 2 in the following way: In Line 1 one uses the message $m_p := m + r_1 \cdot p$ and in Line 2 the message $m_q := m + r_2 \cdot q$. Here, $1 < r_i < t_i$, $i = 1, 2$ are fixed numbers. Obviously, it should hold that $r_1 \cdot p \bmod t_1 \notin \{-2, -1, \ldots, 2\}$, and for Line 2 equivalently. In this way, the algorithm actually computes $m^d \bmod N$. This blinding technique is also useful against other side channel attacks.

3.4.3. Further Security Considerations

Disclosure of most intermediate variables can be used to break the system. Attacks on most intermediate variables, e.g., d_p or m, have a negligible success probability for almost all messages. However, there are messages, where an adversary can mount a successful Bellcore-like attack with extremely high success probability. These messages depend on t_1 or t_2, e.g., $m \equiv \pm 1 \bmod t_1$ for an attack targeting d_p, or $m \equiv 0 \bmod t_1$ for an attack targeting m or the intermediate values of the exponentiation. Therefore, it is crucial to the security of Algorithm 3.8 that no intermediate variables are disclosed. This does not only hold for the secret randomization parameters t_1 and t_2. As an example, assume that our countermeasure prevents a Bellcore attack on a faulty S_p using $c_1 \neq 1$ and $c_2 = 1$. If c_1 is revealed, we have

$$\gcd\left(m^{c_1} - Sig^e, N\right) = p.$$

This also implies that an adversary must not be able to guess the value $c_1 \cdot c_2$ efficiently. This requires two conditions. First, the bitlength of the parameters t_i must be large enough to defend against a brute force search on c_i from Lines 4 and 5 of Algorithm 3.8. Second, the number of possible values for c_i must be large enough. Since Algorithm 3.8 is deterministic, every possible error value for a specific affected variable will yield exactly one value for c_1 and (possibly) c_2. Since we assume the Random Fault Model 1.9, we assume random errors. Since no variable has less than $l := \min(l(t_1), l(t_2))$ bits, this guarantees that at least 2^l different error values occur. Since no operation maps large quantities of different values to single values, the number of possible values c_i is also about 2^l or more. Hence, an adversary cannot guess any value $c_1 \cdot c_2$ with a non-negligible probability. However, we will show in Section 3.6 that this no longer holds for bit or byte errors.

The length of the two parameters t_i should be as small as possible to ease the cost of computation, but it must be large enough to guarantee security. Section 3.4.1 shows that the most promising attacks succeed with a probability of at most $3/\min(t_1, t_2)$. Hence, both t_1 and t_2 must be large enough to ensure that attacking the scheme succeeds only with negligible probability. The definition of "small" and "negligible" will have to be adapted to the actual implementation of a system using our algorithm. If we assume a very high level of security, we will demand a security of 2^{80}, i.e., $l(t_i) > 80$. Less conservative security considerations may allow to reduce

this bound. Practical applications may only need to guarantee the security of the signature key for a small time like 2 years — today's credit cards incorporate the same security feature. In cases like these, $l(t_i) = 60$ seems to be secure (the SETI@home project as one of the largest open attacks achieved about 2^{61} operations until 2002 [SET02]). If less powerful attacks are assumed, this level might be lowered even further.

3.4.4. Summarizing the Results.

Fault Attack on	Probability of the Attack
Line 1	$3/\min(t_1, t_2)$
Line 2	$3/\min(t_1, t_2)$
Line 3	$2/\min(t_1, t_2)$
Line 4	0 in our fault model
Line 5	0 in our fault model
Line 6	0 in our fault model

Table 3.6.: Summarizing the success probabilities of a fault attack adversary

Table 3.6 shows the most successful attack scenarios on each line of Algorithm 3.8. Summarizing the results of this section, the probability to induce an error that can fool our countermeasure and still break the system by the Bellcore attack is negligibly small if the bitlength of t_1 and t_2 is large enough. Additionally, in the real world various randomization strategies are applied on the card to counteract other side-channel attacks. These measures show that malicious messages, which have been shown to exist for some attacks, are virtually impossible to create.

3.5. Adapting to Other Fault Models

In this section, we will investigate the security of Algorithm 3.8 against stronger fault models, namely the Single Bit Fault Model 1.7 and the Byte Fault Model 1.8. First, we will analyze the probability of inducing undetectable faults into any intermediate variable, similar to Section 3.4.1. We will show that our algorithm actually detects almost all bit and byte faults. After this, we will investigate a major weakness of the original Algorithm 3.8 in the presence of bit or byte faults. We will fix this problem, and show that Algorithm 3.8 can also be used against these stronger fault models with minor modifications in Section 3.6.

3.5.1. Undetectable Faults in the Single Bit Fault Model

Similar to the analysis in Section 3.4.1, we present in this section the results for induced bit errors according to the Single Bit Fault Model 1.7. In this model, the effect of a fault induced into a variable x is modeled as $\tilde{x} = x \pm 2^k$ with $0 \leq k < l(x)$. All probabilities stated in the following will be over random choices of k and the sign of k. The term *success probability* always refers to the success probability for a bit fault induced into the targeted variable to result in an undetectable error such that the Bellcore attack can be applied.

Faults Induced into Line 1:

- **Attack targeting the stored variable d_p**
  The success probability for a fault induced into d_p is 0 for messages $m \not\equiv \pm 1 \bmod t_1$. This probability is taken over the errors. A fraction of at most $3/t_1$ of all messages satisfies $m \equiv \pm 1 \bmod t_1$. The message $m = 1$ is secure. For other messages $m \equiv \pm 1 \bmod t_1$, $m \neq 1$, the success probability is at least 1/2.

  Comment. Once again, the adversary has no better choice to choose his inputs than to use random messages. Otherwise he would have to know t_1, which we assume he does not.

  Proof:

  The analysis of this error scenario is completely analogous to the random error fault presented in Section 3.4.1, but the case $\gcd(e(d_p), \varphi(t_1)) = \gcd(\pm 2^k, t_1 - 1) \in \{(t_1 - 1)/2, t_1 - 1\}$ is impossible since $(t_1 - 1)/2$ is assumed to be a prime larger than 2. The case $\gcd(e(d_p), \varphi(t_1)) \in \{1, 2\}$ yields a non-negligible success probability for at most $3/t_1$ of all possible messages m. □

- **Attack targeting the stored variable pt_1**
  If a random bit fault is induced into pt_1, such that pt_1 is changed to $pt_1 \pm 2^k$, an undetectable error requires that $m^d \text{ div } (pt_1 \pm 2^k) \equiv 0 \bmod t_1$.

  Comment. Although this case cannot be reduced to Assumption 3.9, the adversary has no prospective chance to construct any message that might be malicious as he neither knows p, t or d nor has he any control on the location of the induced bit. Therefore, his best chance is to use random messages. This leaves him with a negligible success probability.

  Proof:

  If the modulus is randomly changed to $pt_1' = pt_1 \pm 2^k$, $0 \leq k < l(pt_1)$, consider writing $m^d = \alpha_0 \cdot (pt_1 \pm 2^k) + \alpha_1$ with $\alpha_1 < (pt_1 \pm 2^k)$. The correct result S_p is now $S_p = \alpha_0 \cdot (\pm 2^k) + \alpha_1 \bmod pt_1$, while the faulty result S_p' is α_1. An undetectable error happens, if $S_p \equiv S_q \bmod t_1$. This is the case if $\alpha_0 \cdot (\pm 2^k) + \alpha_1 \equiv \alpha_1 \bmod t_1$, i.e. if $\alpha_0 \cdot (\pm 2^k) \equiv 0 \bmod t_1$. As $\gcd(t_1, \pm 2^k) = 1$, this happens only if $\alpha_0 = m^d \text{ div } (pt_1 \pm 2^k) \equiv 0 \bmod t_1$. □

- **Attack targeting m or the exponentiation's intermediate variable**
  Any random bit fault induced during the exponentiation that causes an intermediate value y of Algorithm 3.10 to be changed to $y \pm 2^k$ must satisfy the equation $\pm 2^k \equiv -2y \bmod t_1$ to induce an undetectable error. For messages $m \equiv 0 \bmod t_1$, all faults yield an undetectable error.

  Comment. The probability that a given message creates an intermediate value y such that the above equation has a solution for k is extremely small. The fraction of messages of the form $m \equiv 0 \bmod t_1$ is at most $1/t_1$ of all possible messages. In addition, the adversary cannot construct malicious messages, because he has no information about t_1. His best choice is to choose random m for input. Therefore, his probability to induce an undetectable error is negligible.

Proof:

If Algorithm 3.10 is attacked at the time when $i = l$, and the intermediate value y is altered, we have $\tilde{S}_p = (y \pm 2^k)^{2^{l-1}} \cdot m^w$ for $w = \sum_{i=1}^{l-1} d_i \cdot 2^i$ and $0 \leq k < l(y)$. Hence messages $m \equiv 0 \bmod t_1$ lead to $\tilde{S}_p \equiv S_p \bmod t_1$ and to an undetectable error. There are at most $1/t_1$ of all possible messages satisfying this condition.

For messages $m \not\equiv 0 \bmod t_1$, the condition $(y \pm 2^k) \equiv 0 \bmod t_1$ is impossible (see the analysis of random faults).

From now on we assume $y \pm 2^k \not\equiv 0 \bmod t_1$. Then $(y \pm 2^k)^{2^{l-1}} \equiv (y)^{2^{l-1}} \bmod t_1$ implies $1 = \left(y/(y \pm 2^k)\right)^{2^{l-1}} \bmod t_1$. Since $t_1 \equiv 3 \bmod 4$, this is equivalent to $1 = \left(y/(y \pm 2^k)\right)^2$. This implies $\pm 1 = \left(y/(y \pm 2^k)\right)$. For any fixed y, $\pm 2^k \equiv 0 \bmod t_1$ implies that $2^k = 0$, which is impossible. Therefore, an undetectable error only happens if $\pm 2^k \equiv -2y \bmod t_1$. □

- **Attack targeting the result S_p**
  The success probability for a fault induced into S_p is 0 for messages $m \neq 0, 1$.

  This case will be analyzed while considering attacks on the CRT combination in Line 3.

Faults Induced into Line 3:

- **Attack targeting the result S, or the two addends S_p and $X \cdot (S_q - S_p)$**
  The success probability for a fault induced into S, S_p or $X \cdot (S_q - S_p)$ is 0.

Proof:

If the result S is faulty, then $S' = S \pm 2^k$, $0 \leq k < l(S)$. This would pass by the detection iff $\pm 2^k \equiv 0 \bmod t_i$ for both i. The analysis of a fault induced into m in Line 1 has shown that this is impossible. With the same argument, faults induced into the two summands S_p and $X \cdot (S_q - S_p)$ will always be detected. □

- **Attack targeting X**
  The success probability for a fault induced into X is 0 for all but a fraction of at most $1/(t_1 \cdot t_2)$ messages. This probability is taken over random choices of the error.

Comment. An attack on X will result in $\tilde{S} = S \pm 2^k \cdot (S_q - S_p)$, $0 \leq k < l(X)$. This may only be an undetectable error if $t_1 \cdot t_2 | (S_q - S_p)$. The number of possible messages satisfying this requirement is smaller than $1/(t_1 \cdot t_2)$. A more detailed analysis has been presented in Proposition 3.3. For this case both t_1 *and* t_2 have to divide $(S_q - S_p)$. In addition, the added error also needs to be a multiple of either p or q in order to allow a Bellcore attack.

As with faults induced into d_p, the adversary has no information on how to construct a message that yields $S_q - S_p$ to be a multiple of t_1, t_2 or both. His best choice is to choose random messages, which only gives him a negligible success probability. Hence, this attack is not promising to an adversary.

- **Attack targeting S_p or S_q or $(S_q - S_p)$**
  The success probability for a fault induced into S_p or S_q is 0.

Proof:

If an adversary hits either S_p or S_q in the second summand, we have $\tilde{S} = S \pm 2^k \cdot X$. This causes an undetectable error iff $t_2|X$, since t_1 always divides $X = pt_1 \cdot ((pt_1)^{-1} \bmod qt_2)$. The case $t_2|X$ is prevented by Condition 5 on the choice of t_2. □

Faults Induced into Lines 4 – 6:

We also need to investigate the possibility to induce faults into the detection mechanism, Lines 4 – 6. But attacks targeting the computation of c_i are in vain unless another fault has been induced already. If an error into a correct c_i is induced, it is $c_i \neq 1$ and the final signature will look like a random value. The same consideration applies to Line 6.

The two special messages $m = 0$ and $m = 1$ need to be excluded from the set of possible inputs. Here, the same considerations as in Section 3.4.2 apply.

Summary.

The analysis of bit errors is completely analogous to the analysis in Section 3.4. The results are summarized in Table 3.7. They show that an adversary's chance of successfully choosing a random m that satisfies any of these conditions is negligible. The overall success probability is less than the success probability for random faults. Since faults have to satisfy very special conditions, bit faults are not the best choice for an adversary, although the Single Bit Fault Model is a stronger model than the Random Fault Model. In this sense, our countermeasure protects better against the stronger fault models. However, as we will discuss in detail in Section 3.6, being undetectable is not equivalent to being no threat to Algorithm 3.8.

Fault Attack on	Probability of the Attack
Line 1	$3/t_1$
Line 2	$3/t_2$
Line 3	$1/(t_1 \cdot t_2)$
Lines 4 – 6	0 in the chosen fault model

Table 3.7.: Summarizing the success probabilities of a fault attack adversary for bit faults

3.5.2. Undetectable Faults in the Byte Fault Model

If an adversary is able to induce faults according to the Byte Fault Model 1.8, he can target any variable x and change it to a faulty value $\tilde{x} = x + b \cdot 2^k$, where $|b| \in \mathbb{Z}_{2^8}$, $0 \leq k < l(x) - 7$.

Fault Attack on	Probability of the Attack
Line 1	$3/t_1$
Line 2	$3/t_2$
Line 3	$1/(t_1 \cdot t_2)$
Lines 4 – 6	0 in the chosen fault model

Table 3.8.: Summarizing the success probabilities of a fault attack adversary for byte faults

The success probabilities for such attacks are completely the same as for bit faults. All proofs presented in Section 3.5.1 hold for byte faults as well, just replace every $e(x) = \pm 2^k$ by $e(x) = b \cdot 2^k$. Therefore, we do not repeat the analysis and simply state the final result in Table 3.8.

3.6. Attacks Against our Proposed Algorithm

Section 3.4 showed that the proposed countermeasure secures the CRT-RSA algorithm against faults based on the Random Fault Model 1.9. Unfortunately, it does not provide sufficient security against faults based on the stronger fault models, namely the Single Bit Fault Model 1.7 or the Byte Fault Model 1.8. This has been described in detail by Wagner in [Wag04]. Moreover, another attack has been described in [Wag04]. Although the latter is flawed and uses a controversial fault model, it offers some insight into the security of our scheme.

3.6.1. Bit and Byte Faults

We have shown in Section 3.4.3 that if $c_1 \neq 1$ or $c_2 \neq 1$ and the infective value is disclosed, an attacker can mount a Bellcore-like attack by computing $\gcd(m^{c_i} - Sig^e, N)$. This yields a factor of N. For bit faults and byte faults, the number of possible errors is rather small, hence, an adversary can guess a possible error value $e(x)$, resulting from a fault induced into some variable x and try to verify his assumption.

As a concrete example, consider a transient fault induced into m according to the Single Bit Fault Model in Line 1 of Algorithm 3.8. In this case, S_p is faulty and S_q is correct, hence, the variable S is faulty. As usual, we denote a faulty S by $\tilde{S}$. Since $\tilde{S} \equiv S \bmod qt_2$, we have $c_2 = 1$. However, a faulty m yields the value $\tilde{S}_p = \tilde{m}^{d_p} \bmod pt_1$, hence, we have $c_1 = (m - \tilde{m} + 1) \bmod t_1 = 1 - e(m) \bmod t_1$. This is not equal to $1 \bmod t_1$, since $t_1 \nmid e(m) = \pm 2^k$ for any choice of $k > 0$.

In order to recover c_1, consider that for a large fraction of possible bit faults or byte faults, we have $-t_1 + 1 < e(m) < 0$. For bit faults, we may have $e(m) = -2^k$ for $0 \leq k \leq l(t_1) - 1$, for byte faults, we have $e(m) = b \cdot 2^k$ for $-2^8 + 1 \leq b \leq -1$ and $0 \leq k \leq l(t_1) - 8$. In this case, c_1 can be recovered by testing all possible values for $e(m)$ according to the chosen fault model. The probability of inducing such a *usable* fault is approximately $1/2 \cdot l(t_1)/l(N)$ according to [Wag04], assuming a variant of the Byte Fault Model. The crucial fact exploited by this attack is that the set of usable errors is efficiently sampleable and highly probable. Hence, an adversary can perform computations for all possible guesses in polynomial time and he can hope to induce a usable error after polynomially many attempts. This is fundamentally different from random faults, where the number of possible faults is too large to be sampled in polynomial time.

Infective computations aim at randomizing the output, but they do not use an own source of randomness. Instead, they use the random source provided by the error. Consequently, their effectiveness depends on the quality of the random error source. If the error is truly random, i.e., uniformly distributed in a large set, a randomization of the final output is possible. Hence, an adversary using the Random Fault Model, as assumed in Section 3.4, faces a randomized faulty output, where the infective randomization strategy provides a good randomization. For all other attacks, which assume errors, that do not represent a good source of randomness, infective computations will always be insecure. This is what happens for the Bit Fault Model and the Byte Fault Model, where most of the values $e(m) \in \mathbb{Z}_{pt_1}$ occur with probability 0 and

only a small fraction occurs with non-zero probability. Any distribution of the error, which is strongly biased or asymmetric, can be susceptible to the attack described above.

Simple Solution for Attacks with Efficiently Sampleable Errors. A simple solution to this problem is straightforward. If the adversary can always recover $\tilde{S}$ if a strong fault model is used, the algorithm must not output a result. Hence, we present with Algorithm 3.11 a simple countermeasure suitable for bit and byte faults.

Algorithm 3.11: Secure CRT-RSA Algorithm with Explicit Checking Procedures

Input: A message $\mathsf{m} \in \mathbb{Z}_\mathsf{N}$
Output: $\mathsf{Sig} := \mathsf{m}^\mathsf{d} \bmod \mathsf{N}$ or a random number in $\mathbb{Z}_\mathsf{N}$
In Memory: $\mathsf{p} \cdot \mathsf{t}_1, \mathsf{q} \cdot \mathsf{t}_2, \mathsf{N}, \mathsf{N} \cdot \mathsf{t}_1 \cdot \mathsf{t}_2, \mathsf{d}_\mathsf{p}, \mathsf{d}_\mathsf{q}, \mathsf{t}_1, \mathsf{t}_2, \mathsf{e}_{\mathsf{t}_1}$, and $\mathsf{e}_{\mathsf{t}_2}$
1 Let $\mathsf{S}_\mathsf{p} := \mathsf{m}^{\mathsf{d}_\mathsf{p}} \bmod \mathsf{p} \cdot \mathsf{t}_1$
2 Let $\mathsf{S}_\mathsf{q} := \mathsf{m}^{\mathsf{d}_\mathsf{q}} \bmod \mathsf{q} \cdot \mathsf{t}_2$
3 Let $\mathsf{S} := \mathsf{CRT}(\mathsf{S}_\mathsf{p}, \mathsf{S}_\mathsf{q}) \bmod \mathsf{N} \cdot \mathsf{t}_1 \cdot \mathsf{t}_2$
4 Let $\mathsf{c}_1 := (\mathsf{m} - \mathsf{S}^{\mathsf{e}_{\mathsf{t}_1}} + 1) \bmod \mathsf{t}_1$
5 Let $\mathsf{c}_2 := (\mathsf{m} - \mathsf{S}^{\mathsf{e}_{\mathsf{t}_2}} + 1) \bmod \mathsf{t}_2$
6 If $(\mathsf{c}_1 \neq 1)$ or $(\mathsf{c}_2 \neq 1)$ then **output** FAILURE
7 **Output** S mod N

Infective computations have the advantage of replacing explicit checking procedures, which always pose a single point of failure. However, a weak source of randomness suggests to dispose of infective computations and return to the explicit checks. Explicit checking procedures are dangerous only if an adversary can induce two faults during the same run of the algorithm. Moreover, as explained in Section 1.4, modern high-end smartcards are equipped with a variety of countermeasures, which allow users to be confident that the strong power of an adversary is reduced to the Random Fault Model. Algorithm 3.8 is secure in this model.

More Sophisticated Solutions for Attacks with Efficiently Sampleable Errors. However, with a little additional work, it might also be possible to protect Algorithm 3.8 against bit or byte faults while still using infective computations. This is a new direction of research, hence, we will only briefly sketch the ideas here. Since the error does not provide enough randomness to sufficiently randomize the two infective values c_1 and c_2, the algorithm needs to acquire another source of randomness. To show how this can rise the level of security, assume that we have two random values R_1 and R_2.

The attack described above, using the Single Bit Fault Model 1.7 or the Byte Fault Model 1.8 to break Algorithm 3.8, exploits the fact that it can enumerate all possible values of the term $S^{e_{t_1}} \bmod t_1$ efficiently, if m has been attacked in Line 1. This helps if the resulting value for c_1 is smaller than t_1, thus, a reduction does not take place. The crucial fact is that an adversary has a good chance to induce faults, where no modular reduction modulo t_1 occurs. In this case, c_1 can be recovered by testing all possible error values. The same considerations apply for a fault induced into m in Line 2, however, due to symmetry, we will only describe attacks targeting the first line.

If the resulting value $c_1 > 1$ is randomized by a sufficiently large random integer R_1, the value c_1 will be a random value as well. In this case, the advantage of the strong fault models 1.7 and 1.8, which provide an efficiently sampleable set of error values, can no longer be used. We present the modified algorithm as Algorithm 3.12.

Algorithm 3.12: Secure CRT-RSA Algorithm with Additional Randomization

Input: A message $\mathsf{m} \in \mathbb{Z}_\mathsf{N}$
Output: $\mathsf{Sig} := \mathsf{m}^\mathsf{d} \bmod \mathsf{N}$ or a random number in $\mathbb{Z}_\mathsf{N}$
In Memory: $\mathsf{p} \cdot \mathsf{t}_1, \mathsf{q} \cdot \mathsf{t}_2, \mathsf{N}, \mathsf{N} \cdot \mathsf{t}_1 \cdot \mathsf{t}_2, \mathsf{d}_\mathsf{p}, \mathsf{d}_\mathsf{q}, \mathsf{t}_1, \mathsf{t}_2, \mathsf{e}_{\mathsf{t}_1}, \mathsf{e}_{\mathsf{t}_2}$, and two random variables R_1 and R_2
1 Let $\mathsf{S}_\mathsf{p} := \mathsf{m}^{\mathsf{d}_\mathsf{p}} \bmod \mathsf{p} \cdot \mathsf{t}_1$
2 Let $\mathsf{S}_\mathsf{q} := \mathsf{m}^{\mathsf{d}_\mathsf{q}} \bmod \mathsf{q} \cdot \mathsf{t}_2$
3 Let $\mathsf{S} := \mathsf{CRT}(\mathsf{S}_\mathsf{p}, \mathsf{S}_\mathsf{q}) \bmod \mathsf{N} \cdot \mathsf{t}_1 \cdot \mathsf{t}_2$
4 Let $\mathsf{c}_1 := (\mathsf{m} - \mathsf{S}^{\mathsf{e}_{\mathsf{t}_1}}) \cdot \mathsf{R}_1 + 1 \bmod \mathsf{t}_1$
5 Let $\mathsf{c}_2 := (\mathsf{m} - \mathsf{S}^{\mathsf{e}_{\mathsf{t}_2}}) \cdot \mathsf{R}_2 + 1 \bmod \mathsf{t}_2$
6 Let $\mathsf{Sig} := \mathsf{S}^{\mathsf{c}_1 \cdot \mathsf{c}_2} \bmod \mathsf{N}$
7 **Output** S

It is obvious from Algorithm 3.12, that $c_1 = c_2 = 1$ if no error occurs. In this case, we have $m - S^{e_{t_i}} \equiv 0 \bmod t_i$ for $i \in \{1, 2\}$. If both R_1 and R_2 are random integers with $l(t_1) = l(t_2)$ bits, then there are $2^{(l(t_1)-1)}$ many possible values for R_1 and R_2. Consequently, there are $2^{(l(t_1)-1)}$ many values for c_1 and c_2 unless $m - S^{e_{t_i}} \equiv 0 \bmod t$. Hence, the values c_1 and c_2 cannot be recovered using only information about the induced error. In this case, c_1 and c_2 will not be disclosed unless an adversary has some knowledge about R_1 and R_2. Therefore, the attack described in [Wag04] cannot be applied any longer.

However, randomness is extremely expensive on smartcards. Hence, it is preferable to protect the CRT-RSA algorithm without requiring random values R_1 and R_2, generated freshly for each run of the algorithm. Consequently, a new idea is to replace the notion of randomness by the notion of unpredictability. The attack described above requires than an adversary is able to enumerate all possible values for c_1. However, if the two factors R_1 and R_2 are large unknown values, the adversary loses the information about the set of possible values for faulty c_1 and c_2. This might already be enough to defend against the attack described above. In this case, there are several possible choices for R_1 and R_2, e.g.,

- two different fixed random values computed and stored on the card at production time,
- $R_1 = R_2 = d$,
- $R_1 = p$, $R_2 = q$ (or any combination of p and q),
- $R_1 = S_p$, $R_2 = S_q$ (or any combination of S_p and S_q), or
- $R_1 = H(r_1)$, $R_2 = H(r_2)$, where H is a cryptographically strong hash function and r_1 and r_2 are any of the values d, S_p, or S_q.

Since the adversary has no better choice to unveil c_1 and c_2 than to check all possible values from $\mathbb{Z}_{t_1}$ or $\mathbb{Z}_{t_2}$ (depending on which line has been attacked), it should be sufficient to choose $R_1 = R_2$. Of the above suggested choices for R_1 and R_2, it is preferable to have R_1 and R_2 depend on S, S_p, and/or S_q rather than on other values, because these ensure that the random factors depend on the chosen message and on the induced fault, for a lot of possible fault locations. However, the security of Algorithm 3.12 has not yet been proved. It is still an open problem.

3.6.2. Wagner's Attack

Another attack has been described in [Wag04]. Although the attack is flawed, we describe the idea, because it is an interesting approach. This attack relies on a controversial fault model,

however, if this attack was successful, it would render the use of Algorithm 3.8 useless as a countermeasure. The fault model proposed in [Wag04] uses an extremely asymmetric error distribution, where all faults only affect the lower $l(x) - 160$ bits of an affected variable x, whereas the highest 160 bits are unchanged. Therefore, it is "somewhere between" the Byte Fault Model and the Random Fault Model. We will refer to this fault model as *Wagner's Fault Model*.

Wagner's Fault Model is controversial, because it has never been considered before, and it has not been justified by practical considerations either. It cannot be achieved using the Random Fault Model 1.9, since an adversary can only hope to induce a random fault with an effect of the required kind with probability $1/2^{160}$, which is negligible. For bit faults and byte faults, however, this is possible with high probability. Moreover, if a smartcard only uses block-wise encryption of the data in memory, an attack can easily target the $l(x) - 160$ least significant bits of a variable x. Modern smartcards, however, implement a variety of countermeasures, which allows to put great confidence in the assumption that Wagner's Fault Model is unrealistic. Moreover, Wagner's attack requires several faulty results.

In the attack described in [Wag04], the modulus N in Line 6 of Algorithm 3.8 is targeted with Wagner's Fault Model. This attack aims at disclosing t_1 and t_2. Such a disclosure allows to break the system as explained in Section 3.4.3. A correct "large" signature S as computed in Line 3 can be written as $S = (S \bmod N) + k \cdot N$, for some $0 \leq k < 2^{161}$ (since $l(t_1 \cdot t_2) \leq 2^{160}$). A fault induced into N according to this new fault model yields $\tilde{N}$. Given the correct result $Sig = S \bmod N$ and a faulty final result $\widetilde{Sig} = S \bmod \tilde{N}$, we have

$$\widetilde{Sig} - Sig \equiv S - Sig \equiv (S \bmod N) + k \cdot N - (S \bmod N) \equiv k \cdot (N - \tilde{N}) \bmod \tilde{N}.$$

Since we know that $|N - \tilde{N}| \leq 2^{l(N)-161}$, we have $|k \cdot (N - \tilde{N}| < 2^{l(N)}$. With probability at least $1/2$, we have $l(N) = l(\tilde{N})$, hence, when computing $\widetilde{Sig} - Sig \bmod \tilde{N}$, there will be no overflow or modular reduction with sufficiently high probability. In this case, an integer multiple of k is known to the adversary and several attacks on N will allow him to compute k. This can be done by computing many pair-wise gcd's and taking the majority vote. Once an adversary has k, he can compute $S = Sig + k \cdot N$. Wagner claims that this allows to derive $t_1 \cdot t_2 = (S - Sig)/N$. However, we have $(S - Sig)/N = ((S \bmod N) + k \cdot N - (S \bmod N))/N = k$. Hence, the adversary only knows a lower bound $k < t_1 \cdot t_2$. If he can conclude the value of $t_1 \cdot t_2$, he is done, since factoring a 160 bit integer $t_1 \cdot t_2$ is clearly within the reach of modern factoring algorithms. However, he needs a lot of attacks to bound $k < t_1 \cdot t_2$ close enough to sample all possible values. Hence, this is not a feasible attack and no threat to our proposed algorithm.

3.7. Concluding Remarks and Open Problems

The Bellcore attack presented in this chapter is an extremely dangerous attack, since only a single faulty result breaks the system. Compared to attacks on plain RSA in Chapter 2, it shows that a gain in speed may often be paid with a loss in security. Hence, all countermeasures proposed add additional operations, thus slowing down the original CRT-RSA algorithm.

We have proposed a new algorithm, Algorithm 3.8, which protects against the Bellcore attack in the most realistic scenario, the Random Fault Model. For stronger fault models, Algorithm 3.11 has been presented, which no longer uses the benefits of infective computations, but protects the CRT-RSA scheme even in the Single Bit Fault Model and in the Byte Fault Model. A possible improvement to Algorithm 3.11 has been proposed with Algorithm 3.12, although, the security

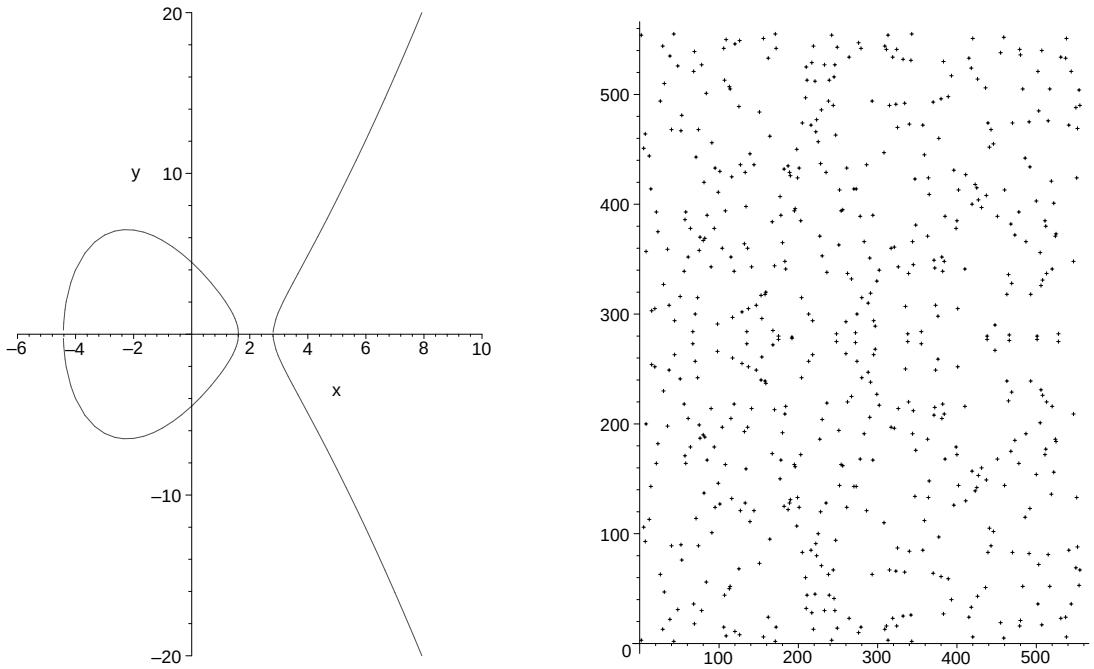
of the latter has not yet been established. Our algorithms develop known ideas into a form such that the algorithm can be proved to be secure within the presented framework with respect to the Bellcore attack. The only problem still unsolved by software mechanisms poses the Chosen Bit Fault Model 1.6 (attack on an unprotected smartcard). If the power of the adversary is not reduced by hardware or software means, the adversary may perform a successful oracle attack circumventing the proposed countermeasure, as described in Section 1.6.2. All currently proposed CRT-RSA implementations are broken by this oracle attack. To prevent an adversary to be able to use the Chosen Bit Fault Model, smartcards must fight both the cause of error and the effect on computation to reduce the power of the adversary significantly. Luckily, various but not all hardware manufacturers of cryptographic devices such as smartcard ICs have been aware of the importance of protecting their chips against intrusion (cf. Section 1.1.1). Another approach is to develop software countermeasures that use random bits to alter the parameters even for the same inputs. As oracle attacks need to test several identical runs of an algorithm, this will effectively reduce the power of the adversary. We leave this direction of research as an open problem.

We would also like to mention a possible generalization of our new countermeasure. The basic idea of this countermeasure is that the computation of $m^d \bmod N$ is performed as $m^d \bmod N \cdot t$ first, tested modulo t and then reduced modulo N for the final output, with some infective computations added. This basic principle may also yield a good approach for security of plain RSA schemes, i.e., standard repeated doubling. The analysis of such an approach is still an open problem.

4. Fault Attacks on Elliptic Curve Cryptosystems

In the previous chapters, we investigated fault attacks on the RSA cryptosystem. However, the other major family of public key cryptosystems, elliptic curve cryptosystems, is also susceptible to fault attacks. In this chapter, we will briefly introduce elliptic curves and elliptic curve cryptography in Section 4.1, and investigate how elliptic curves can be attacked by fault attacks in Sections 4.2, 4.3, and 4.4. As a consequence, we will develop a new fault type in Section 4.5, Sign Change Faults. This new fault type will be used to attack elliptic curve scalar multiplication in Chapter 5. The new attacks establish the need for a new protection mechanism to secure scalar multiplication against Sign Change Faults. Such a countermeasure will be presented in Chapter 6.

4.1. Elliptic Curve Cryptography



(a) Defined over $\mathbb{R}$. (b) Defined over $\mathbb{F}_{557}$.

Figure 4.1.: The elliptic curve $y^2 = x^3 - 15 \cdot x + 20$.

Elliptic curves have first been proposed for use in cryptography by Koblitz [Kob87] and independently by Miller [Mil85] in the mid 1980s. They can be used as a basis for any group based cryptosystem, e.g., the ElGamal cryptosystem [ElG85]. The security of elliptic curve cryptosys-

tems is generally assumed to be based on the discrete logarithm problem, which in turn is assumed to be hard. Actually, for practical systems, the security reduction is less straightforward. For the ElGamal cryptosystem [ElG85], the recovery of the secret key can be reduced to the discrete logarithm problem (ECDLP), whereas recovery of an encrypted message relies on the computational Diffie-Hellman problem (ECDHP). Semantic security of the ElGamal cryptosystem (in a semantically secure variant) is based on the decisional Diffie-Hellman problem (ECDDH). It holds that $ECDDH \leq_p ECDHP \leq_p ECDLP$. The three problems are not proved to be equivalent. While $ECDHP \equiv_p ECDLP$ in some groups, $ECDDH$ is probably easier than $ECDHP$ in some (other) groups. For details on these topics, we refer the reader to [HMV04], [Sma04], or [MvOV96]. For attacking the ECDLP, the best algorithms known today are generic algorithms, e.g., combining the Pohlig-Hellman algorithm with Pollard's rho method, with a running time not significantly faster than $O(\sqrt{p})$, where p is the largest prime divisor of the order of the attacked group. This implies exponential running time. These days, elliptic curves are becoming a popular alternative to RSA for public key cryptography, because they require much shorter key sizes than RSA, i.e., a private key with 160 bits for elliptic curves offers approximately the same security as a private RSA key with 2048 bits (see Table 4.2 in Section 4.1.4).

Elliptic curves are generally defined as the set of points satisfying a certain cubic equation called the *Weierstrass Equation* in some field $\mathbb{K}$, i.e., as the graph of the Weierstrass Equation. Mathematically, they are a special class of planar curves. Elliptic curves and their points can be expressed in a variety of representations. The most popular representations are affine coordinates, projective coordinates, and Montgomery form. However, there are other representations and flavors, e.g., Jacobian coordinates (original as defined in [IEE99] , Chudnovsky-Jacobian [CC86], or modified [CMO98]), Hessian coordinates [JQ01], and López-Dahab coordinates [LD99]. In this thesis, we will only work with affine and projective coordinates. However, most results can be easily extended to other representations.

4.1.1. Affine Coordinates

The general Weierstrass equation in affine representation defined over a field $\mathbb{K}$ is

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6. \tag{4.1}$$

However, in most cases, like this thesis, the field $\mathbb{K}$ has characteristic not equal to either 2 or 3, which allows to transform Equation (4.1) into the more compact form

$$y^2 = x^3 + Ax + B. \tag{4.2}$$

An example of an elliptic curve with parameters $A = -15$, $B = 20$ and $\mathbb{K} = \mathbb{R}$ is shown as Figure 4.1(a). Elliptic curves used in cryptography are usually defined over a prime field $\mathbb{F}_p$ with p prime or over a binary field $\mathbb{F}_{2^t}$ for some large $t \in \mathbb{N}$. In this thesis, we will concentrate on elliptic curves defined over prime fields only. Figure 4.1(b) shows the curve $E : y^2 \equiv x^3 - 15x + 20 \bmod 557$.

The set of all points (x, y) on an elliptic curve together with a designated *point at infinity*, denoted by $\mathcal{O}$, form an abelian group (cf. [BSS99], [Was03]). We will write $(x, y) \in E$ to denote that a point (x, y) is on the curve E. The point at infinity can be thought of as a point that is located on top of the y-axis. It intersects with every vertical line. While this point at infinity might appear as an artificial construct in affine coordinates, its existence becomes clear when the elliptic curve is considered in projective coordinates. This coordinate system will be introduced

later. $\mathcal{O}$ is the neutral element of the group operation, which is written in additive notation and referred to as point addition. Addition of two points on an elliptic curve is best defined geometrically by the *chord-and-tangent rule* (see Figure 4.2): Given two points P_1 and P_2 on an elliptic curve E, the sum $P_3 := P_1 + P_2$ first requires to compute $-P_3$ as the third intersection of E with the line defined by P_1 and P_2. P_3 is then computed by reflecting $-P_3$ across the x-axis. Note that if $P_1 = P_2$, the tangent line at P_1 is used to determine the third intersection. If $P_1 = -P_2$, the third intersection is the point $\mathcal{O}$.

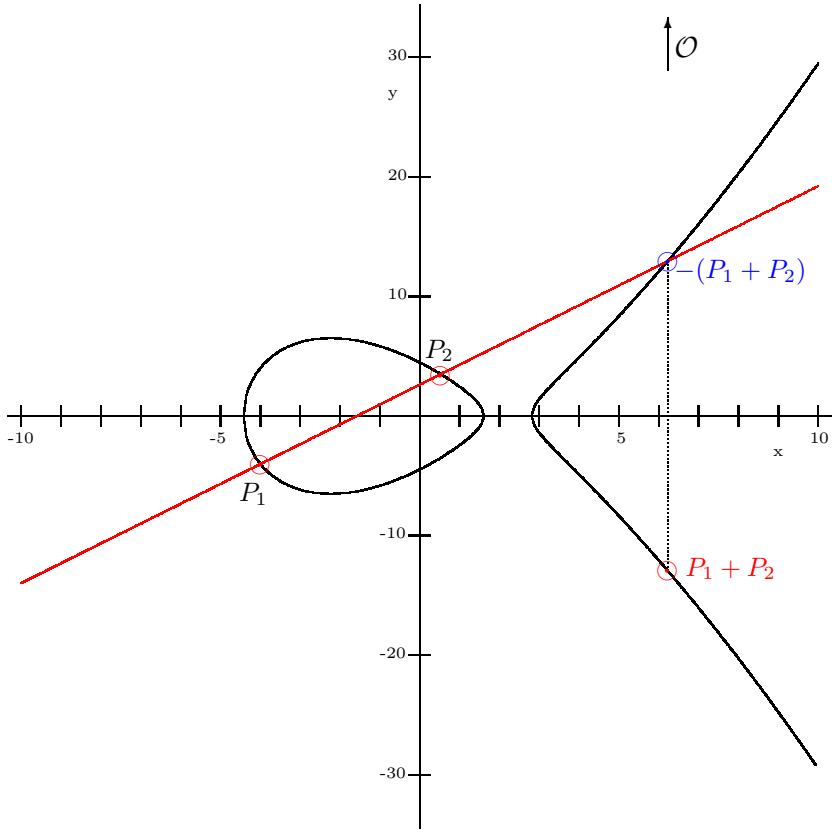


Figure 4.2.: Point addition on an elliptic curve.

Naturally, the geometric addition rule can also be expressed in a formula:

Definition 4.1 (Elliptic Curve Point Addition, Affine Coordinates).
(following the IEEE P1363 standard [IEE99])
Let $E : y^2 = x^3 + Ax + B$ be an elliptic curve over a prime field with characteristic not equal to 2 and 3. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on E. Then $P_3 := P_1 + P_2$ is computed as follows:

1. *If $P_1 = \mathcal{O}$, then $P_3 := P_2$, if $P_2 = \mathcal{O}$, then $P_3 := P_1$.*
2. *If $P_1 = -P_2$, i.e., if $(x_1, y_1) = (x_2, -y_2)$, we have $P_3 := \mathcal{O}$.*
3. *If $P_1 \neq -P_2$, we have $P_3 = (x_3, y_3)$ with*

$$x_3 := \lambda^2 - x_1 - x_2 \tag{4.3}$$
$$y_3 = -y_1 + \lambda \cdot (x_1 - x_3) \tag{4.4}$$

$$\text{where } \lambda := \begin{cases} \frac{y_1 - y_2}{x_1 - x_2} & \text{if } P_1 \neq P_2 \\ \frac{3x_1^2 + A}{2y_1} & \text{if } P_1 = P_2. \end{cases}$$

Note that A is the parameter from the curve equation $y^2 = x^3 + Ax + B$.

4.1.2. Projective Coordinates

Elliptic curves in projective coordinates are given by the homogeneous Weierstrass equation

$$E : y^2 z = x^3 + Axz^2 + Bz^3, \tag{4.5}$$

assuming that the underlying field $\mathbb{K}$ has characteristic different from 2 and 3. Points on E are represented as triplets $P = (x : y : z)$. The notation of points on E arises from the fact that $(x : y : z) = (\lambda x : \lambda y : \lambda z)$ for any $\lambda \in \mathbb{K} \setminus \{0\}$, i.e., points are characterized by the ratio between their coordinates. Here, the point at infinity is $\mathcal{O} = (0 : 1 : 0)$. Note that the point $(0 : 0 : 0)$ is not a valid point in the projective space.

Projective coordinates have a computational advantage over affine coordinates when being defined over a finite field. The standard addition formula requires no multiplicative inverses of field elements, which are usually expensive to compute. As the point at infinity is a valid point in the projective space, it requires less attention than in affine coordinates. A special treatment may otherwise be a source for attacks, as shown in [IT03].

Jacobian and Hessian coordinates, as mentioned above, are also projective coordinates with a different weighing of the three coordinate directions.

The addition of two points P_1 and P_2 on E in projective coordinates is defined algorithmically in the following definition. A nice overview of several addition formulas for different projective coordinates can be found in [CMO98].

Definition 4.2 (Elliptic Curve Point Addition, Projective Coordinates).
(following the IEEE P1363 standard [IEE99])
Let $E : y^2 z = x^3 + Axz^2 + Bz^3$ be an elliptic curve over a prime field with characteristic not equal to 2 and 3. Let $P_1 = (x_1 : y_1 : z_1)$ and $P_2 = (x_2 : y_2 : z_2)$ be points on E. Then $P_3 := P_1 + P_2$ is defined as follows:

1. *If $P_1 = \mathcal{O}$, then $P_3 = P_2$, if $P_2 = \mathcal{O}$, then $P_3 = P_1$.*
2. *If $P_1 = -P_2$, i.e. if $(x_1 : y_1 : z_1) = (x_2 : -y_2 : z_2)$ then we have $P_1 + P_2 = \mathcal{O} = (0 : 1 : 0)$.*
3. *If $P_1 \neq \pm P_2$, then we have $P_1 + P_2 = (x_3 : y_3 : z_3)$ as*

$$\begin{aligned} x_3 &:= r^2 - tw^2 \\ 2y_3 &:= vr - mw^3 \\ z_3 &:= z_1 z_2 w, \end{aligned}$$

$$\begin{aligned} \text{where} \quad u_1 &:= x_1 z_2^2, & s_1 &:= y_1 z_2^3, & w &:= u_1 - u_2, & r &:= s_1 - s_2, \\ u_2 &:= x_2 z_1^2, & s_2 &:= y_2 z_1^3, & t &:= u_1 + u_2, & m &:= s_1 + s_2, \\ \text{and} \quad v &:= tw^2 - 2x_3 \end{aligned}$$

4. If $P_1 = P_2$, then we have a point doubling, $2P_1 = (x_3 : y_3 : z_3)$ as

$$\begin{aligned} & x_3 := m^2 - 2s \\ & y_3 := m \cdot (s - x_3) - t \\ & z_3 := 2y_1 z_1, \\ \textit{where} \quad & m := 3x_1^2 + Az_1^4, \qquad s := 4x_1 y_1^2, \qquad \textit{and} \quad t := 8y_1^4. \end{aligned}$$

Note that A is the parameter from the projective Weierstrass Equation (4.5).

In practice, any point $(x : y : z)$ with $z = 0$ is interpreted as the point $\mathcal{O}$. In this case, Case 2 in Definition 4.2 can be handled by Case 3, which returns $z_3 = 0$ if $P1 = -P2$. Moreover, the IEEE P1363 standard [IEE99, A.10.5] suggests the following procedure: First, use the addition formula, i.e., Case 3, for all inputs. This case should return the triplet $(0 : 0 : 0)$ if the doubling formula, i.e., Case 4, has to be applied. Then the doubling formula would be applied if necessary.

4.1.3. Elliptic Curve Cryptosystems

As elliptic curves form an abelian group, any group based cryptosystem can be based on elliptic curves. Here, the most prominent example is the ElGamal cryptosystem. We give an example for EC-ElGamal in the following. It has been taken from [Cor99].

Protocol 4.3 (Elliptic Curve ElGamal Cryptosystem).

System parameters:
- *Choose an elliptic curve E defined over a prime field $\mathbb{F}_p$ or over a binary field $\mathbb{F}_{2^t}$ such that the order of E is divisible by a large prime q.*
- *Choose a base point P on E of order q.*

Key generation:
- *Choose the secret key $k \in_R \{1, 2, \ldots, q-1\}$.*
- *Compute the public key $Q = k \cdot P$ on E.*

Encryption:
- *Choose a random session key $r \in_R \{1, 2, \ldots, q-1\}$.*
- *Compute the two points $R = r \cdot P = (x_1, y_1)$ and $r \cdot Q = (x_2, y_2)$ $(= r \cdot k \cdot P)$.*
- *Compute $c = x_2 + m$, where $m \in \mathbb{Z}_p$ is the message to be encrypted.*
- *Send out $(R, c) = (x_1, y_1, c)$.*

Decryption
- *Compute $(x_2', y_2') = k \cdot R$.*
- *Recover $m := c - x_2'$.*

Obviously, the choice of E and P is crucial for the security of the system. The order of the base point P must be a large prime. Otherwise, discrete logarithm algorithms like the algorithm of Pohlig and Hellman [PH78] will be able to break the system. This implies that $\#E$ must have a large prime factor as well. In practice, several standardization bodies recommend curves for use in cryptosystems. These curves usually have prime order or an order with a small cofactor of 2 or 4, see Section 4.1.4 for details. Other security aspects, which are inherent to the ElGamal cryptosystem, must be regarded as well, e.g., no repeated use of the session key r. As they are not motivated by the use of elliptic curves, we refer the reader to standard literature for details, e.g., [MvOV96].

We would also like to stress the point that there are many ways to embed the message m into the system. As an alternative for the method chosen above, which simply computes an integer

c, it is also possible to break m into a tuple $m = (m_1, m_2)$ and transmit

$$(R, x_2 \cdot m_1, y_2 \cdot m_2). \tag{4.6}$$

as proposed in [GG99, §20.6]. This allows to transmit messages more efficiently. The crucial observation is that in the case of Protocol 4.3, the y-coordinate of kR is not needed in the decryption step, while Equation (4.6) requires the computation of the complete point kR. There are scalar multiplication algorithms, such as a variant of Montgomery's binary method, which can compute the x-coordinate only. This in turn is more efficient than computing both coordinates. However, there are verification schemes, which need the y-coordinate according to [IT02]. Note that it is not possible to use the point $M = (m, y)$ for some $y \in \mathbb{F}_p$ in order to embed a message m, because it is not guaranteed that there exists a valid y-coordinate such that M is a point on E.

4.1.4. Elliptic Curve Parameters in Practice

Several standardization bodies recommend elliptic curve domain parameters. These elliptic curves can be used in practice and they are considered to be cryptographically secure. In Table 4.1, we provide an overview over the properties of these recommended curves. We consider the recommendations of IEEE P1363-2000 [IEE98], Certicom's Standards for Efficient Cryptography Group (SECG) [SEC00], ANSI [ANS99], Federal Information Processing Standard FIPS 186-2 issued by the NIST [FIP00], the Wireless Transport Layer Security Specification by the WAP forum [WAP01], the specification for the Financial Services Markup Language (FSML) by the eCheck initiative [FSM99], the final recommendation of the NESSIE consortium [NES03], and from IPSec [RFC98a], [RFC98b], [IPS01].

Table 4.1 states properties of three different types of elliptic curves, curves defined over prime fields $\mathbb{F}_p$, curves defined over binary fields $\mathbb{F}_{2^t}$ and Koblitz curves. The latter refers to a special subset of curves which can be computed very efficiently. Koblitz curves may be defined over both prime fields and binary fields. For binary fields, Koblitz curves usually denote special curves with $A, B \in \{0, 1\}$, i.e., defined over the base field $\mathbb{F}_2$. A and B are the curve parameters from the Weierstrass equation for binary curves, which is

$$y^2 + xy = x^3 + Ax + B. \tag{4.7}$$

For prime fields, Koblitz curves refer to curves which possess an efficiently computable endomorphism. As Koblitz curves are not considered in detail in this thesis, we refer the reader to [Kob92] and [Gal99] for these details.

Elliptic curve cryptosystems offer the highest security per bit of any public key cryptosystem used today. Table 4.2 (taken from [Cer00]) presents a direct comparison of the system parameters of the same strength for RSA and elliptic curve cryptosystems.

4.2. Existing Fault Attacks on Elliptic Curve Cryptosystems

Fault attacks have first been described for the RSA cryptosystem in [BDL01]. Soon thereafter, these ideas have also been applied to several variants of elliptic curve cryptosystems. The first successful attacks aimed at special elliptic curve cryptosystems such as the KMOV cryptosystem or the Demytko cryptosystem (cf. [JQ96], [JQ97b], [JQBD97], [JLQ99]). Other attacks applied

Body	**Prime Curves**	**Binary Curves**	**Koblitz Curves**
IEEE	size: > 161 bits any "small" cofactor avoid known weak curves gives no specific curves	size: > 161 bits any "small" cofactor avoid known weak curves gives no specific curves	size: > 161 bits any "small" cofactor avoid known weak curves gives no specific curves
SECG	size: 112 – 512 (various) prime order or cofactor 4	size: 113 – 571 (various) cofactor 2	size: 160 – 571 (various) prime order over $\mathbb{F}_p$ cofactor 2 or 4 over $\mathbb{F}_{2^t}$
ANSI X9.63	size: > 160 any "small" cofactor[a] gives no specific curves	size: > 160 any "small" cofactor[a] gives no specific curves	-
NIST	size: 192,224,256,384,521 prime order parameter $A = -3$ fixed	size: 163,232,283,409,571 cofactor 2	size: 163,232,283,409,571 cofactor 2 or 4 only binary fields used
WAP	size: 112, 160, 224 prime order (some equal to SECG)	size: 113, 163, 233 cofactor 2 (some equal to SECG)	size: 163, 233 cofactor 2 or 4 over $\mathbb{F}_{2^t}$ (some equal to SECG)
FSML	-	size: 163, 283 cofactor 2 (subset of SECG)	size: 163, 283 cofactor 2 or 4 over $\mathbb{F}_{2^t}$ (subset of SECG)
NESSIE	size: > 160 prime fields preferred no explicit restrictions	size: > 160	-
IPSec	-	size: 155 – 571 cofactor 2, 4, or 12 (most equal to SECG)	-

[a] Actually, given a base point P on E of order n, the size of the cofactor $h := \#E/n$ is bounded by $n > 4\sqrt{q}$ where $\mathbb{F}_q$ is the finite field. ANSI admits that for prime fields $n \approx q$ will hold in practice.

Table 4.1.: Recommended Elliptic Curve Parameters

	System parameters (bits)	**Public key (bits)**	**Private key (bits)**
RSA	n/a	1088	2048
DSA	2208	1024	160
ECC	481	161	160

Table 4.2.: Size of system parameters and key pairs (approx.) from [Cer00]

general ideas for fault attacks which worked without specific regard to elliptic curve cryptosystems, e.g., by attacking the pseudo random number generator used on a smartcard ([ZM96], [ZM97]), by attacking the secret scalar/exponent used in the ElGamal cryptosystem ([BDH+98], [Dot02], and [GK04]), or by using memory-safe or computational-safe errors in implementations secure against timing or power attacks (see Section 1.6.2 and [YJ00], [YKLM01a]).

More general fault attacks exploiting specific properties of elliptic curves have been presented in [BMM00] and [CJ03].

The attacks presented in [BMM00] target the scalar multiplication on an elliptic curve, where the scalar multiple $Q = kP$ of a base point P is computed. The authors investigate what happens if the base point P is replaced by an arbitrary point $\tilde{P}$ at the beginning of the multiplication. If $\tilde{P}$ is not a point on the original curve E, the operation is called *pseudo-multiplication*, build up by *pseudo-additions*. In this case, the computation of $k\tilde{P}$ takes place on a different curve $\tilde{E}$. This curve is known to an adversary if $\tilde{P}$ is known, which is due to the fact that the B-parameter of a curve is used neither in the addition nor in the doubling formula (see Definition 4.1). Therefore, the curve $\tilde{E}$ can be specified by A and a new parameter $\tilde{B}$, which can easily be computed from the Weierstrass equation (4.2) given A and any point $\tilde{P}$. If the resulting curve $\tilde{E}$ is a cryptographically weak curve, the discrete logarithm problem might be tractable, yielding important information about the secret scalar k. Usually, several attacks are necessary in order to recover k by combining several solutions for k modulo small primes via the Chinese Remainder Theorem.

The attack becomes feasible if an adversary can choose the base point $\tilde{P}$. This allows him to choose a cryptographically weak curve with a subgroup of small order, and a base point $\tilde{P}$ from such a subgroup. If the base point cannot be chosen, but it is changed by a fault attack according to a known fault model with a limited number of possible faulty results, e.g., the Byte Fault Model 1.8, the attack is feasible as well. In this case, all possibilities for the faulty base point are tested until the curve $\tilde{E}$ is found. If that curve allows to solve the discrete logarithm problem, the secret scalar k can be recovered modulo some small order. The success probability heavily depends on the chosen fault model. The authors also extend their attack to fault attacks on intermediate variables of repeated doubling, which results in a sequence of normal additions followed by pseudo-additions.

In [CJ03], the results from [BMM00] are generalized. The authors show that the attack is also feasible if random errors in either the x- or the y-coordinate are assumed, thus relaxing the fault model to the more realistic scenario of the Random Fault Model 1.9 or the Arbitrary Fault Model 1.10. With additional (strong) assumptions, even random faults in both coordinates yield information about the secret scalar k. The attacks are also shown to be feasible for faults induced into the field representation, i.e., into the prime p if a prime field $\mathbb{F}_p$ is used or into the irreducible polynomial used in the case of binary fields $\mathbb{F}_{2^t}$. For such attacks, random faults require additional knowledge to allow successful attacks, while bit or byte faults yield better results. For binary fields, the attack is only feasible in affine coordinates, as projective doubling uses all curve parameters. As a last generalization, Ciet and Joye show that faults induced into any of the parameters a_1, a_2, a_3, a_4 used in the general Weierstrass Equation (4.1) can be used for successful attacks.

The authors of [BMM00] and [CJ03] claim that all their attacks can easily be fended off. If the smartcard checks whether the final result is a valid point on the original curve, it is claimed that the check captures faulty results with an overwhelming probability. Note that this final check requires that all curve parameters are checked for integrity, e.g., by using CRC sums to detect permanent errors. This result assumes that faults are induced in a way, such that the final result

$\tilde{Q}$ cannot be efficiently predicted and lies on the original curve only with negligible probability. In this case, the attack is not a real threat to modern systems, the proposed countermeasure is easy and has been deployed in all modern systems today. We will refer to the countermeasure of checking if a final point is a valid point on a curve E as the *standard countermeasure* in the following. The results from [BMM00] and [CJ03] show that elliptic curve cryptosystems are much more resistant against the classic fault models defined in Section 1.4 than, e.g., RSA. The reason for this is quite simple: in RSA, any change of a value from $\mathbb{Z}_N$ yields a valid value in $\mathbb{Z}_N$, while in elliptic curve cryptosystems, the tuple (x, y) is characterized by a special relation between x and y, namely the Weierstrass Equation. Hence, faults induced into elliptic curve points must ensure that this condition is satisfied by the faulty values as well. There are p^2 many possible point coordinates $(x, y) \in \mathbb{F}_p^2$. However, a curve E over $\mathbb{F}_p$ has at most $p+1+2\sqrt{p}$ many points, according to Hasse's Theorem [Sil00]. Therefore, only a very small fraction of roughly $1/p$ of all possible point coordinates represent valid points on a specific curve. Hence, if a valid point (x, y) is changed randomly, the chances of generating a valid point on a given curve are extremely low. Therefore, elliptic curve points are inherently more secure against fault attacks than single finite field elements.

The crucial point in the attacks presented in [BMM00] and [CJ03] is that the authors assume that a faulty result is not on the original elliptic curve with overwhelming probability. Such faulty results can be detected easily. Any attack that yields faulty results, which are valid points on the original curve, would be undetectable by the aforementioned standard countermeasure. Such points can be used for new attacks. In the remainder of this chapter, we will carefully derive necessary conditions for errors yielding such undetectable faulty results. We will show that contrary to the belief in [BMM00] and [CJ03], it is possible to create undetectable errors with a high probability, thus rendering the proposed standard countermeasure useless. This will yield a new type of fault attacks, Sign Change Faults.

4.3. Undetectable Faulty Points in Point Addition

The attacks described in Section 4.2 all worked on a "high" level of the attacked algorithms, i.e., either the scalar multiplication itself or the point or curve parameters were targeted, rather than the addition or doubling formulas for elliptic curve points. However, faults induced into the addition formulas might be useful to recover secret data as well. This idea is not entirely new. In [IT03], Izu and Takagi concentrate on the addition formulas by exploiting the fact that a projective point with zero z-coordinate cannot be transformed into affine coordinates. Any such point represents the point at infinity $\mathcal{O}$. This raises an exception, which can yield valuable information if detected. However, the setting in [IT03] is rather special and it is not applicable to cryptographically secure elliptic curves. Moreover, the attack described in [IT03] does not use faults.

Therefore, we present in the following an analysis investigating the possibility of attacks which induce faults into the variables used in the addition formula. We will state our results for the affine addition formula only, because our aim is to show that undetectable faulty points can be created. We specify what errors must be induced in order to bypass the standard countermeasure described in Section 4.2. We omit an analysis of the affine doubling formula and of addition formulas for other point representations, as analyzing the affine addition already yields the desired results. Analyses for other formulas are rather lengthy and a task best done using a computer algebra system, e.g., Maple.

For our analysis, we investigate each variable used in the affine addition (see Definition 4.1(3)) separately. We do not choose a specific fault model yet, as we only want to derive conditions the inflicted error has to meet in order to yield an undetectable faulty point $\tilde{P}_3$. However, we may assume that upon attack, the error induced into a variable x can be written as a summand, i.e., $x \overset{\not}{\mapsto} x + e$ for $e \in \mathbb{F}_p$. We consider e to be a random variable. The distribution of e depends on the fault model. The possibility to satisfy the conditions derived for e by any of the known fault models will be analyzed later. However, since some intermediate variables are used more than once, we will consider both transient and permanent faults. To clear notation, we introduce the following definition.

Definition 4.4 (Valid Points and Valid Faulty Points).
Given a point $P = (x, y)$, we call P a valid point *of the elliptic curve E if (x, y) satisfies the Weierstrass equation of E. If we investigate a certain operation, i.e., addition, doubling, or scalar multiplication, we call P a* valid faulty point*, if the operation has been attacked and the resulting point P is different from the correct result, yet it still satisfies the Weierstrass equation of E.*

Let P_1, P_2 be two points on an elliptic curve E defined by its Weierstrass equation $y^2 \equiv x^3 + Ax + B \bmod p$. For the point addition, we assume that $P_1 \neq \pm P_2$. According to Definition 4.1, this implies that $P_3 = (x_3, y_3) := P_1 + P_2$ is computed as

$$x_3 := \lambda^2 - x_1 - x_2 \tag{4.8}$$

$$y_3 = -y_1 + \lambda \cdot (x_1 - x_3)\,, \tag{4.9}$$

$$\text{where } \lambda := \frac{y_1 - y_2}{x_1 - x_2}. \tag{4.10}$$

The variables used are the point coordinates x_1, y_1, x_2, y_2, the intermediate variable λ and the output coordinates x_3 and y_3. All of them can be targeted by fault attacks.

In the following, we present a small selection of results for faults induced into individual variables as examples. The complete analysis is rather technical and has been included in this thesis as Appendix A. It describes in detail the conditions required for an error to be undetectable. These details are given for each variable used in Equations (4.8), (4.9), and (4.10).

4.3.1. Examples for a Detailed Analysis of Undetectable Errors

In this section, we present the analyses of faults induced into two intermediate variables in detail. These results will be referred to again later in this chapter.

4.3.1.1. Faults Induced into y_1 During the Computation of $\lambda = (y_1 - y_2)/(x_1 - x_2)$

For a fault induced into the variable $y_1 \overset{\not}{\mapsto} y_1 + e$, $e \in \mathbb{F}_p$ in Equation (4.10), we need to investigate both transient and permanent faults, because y_1 is used again in the computation of y_3, i.e., Equation (4.9).

Transient Faults. The induced fault enforces the usage of the faulty values $\tilde{\lambda}$, $\tilde{x}_3$, and $\tilde{y}_3$ in the computation. Given a transient fault, they take the values

$$\tilde{\lambda} = \frac{y_1 + e - y_2}{x_1 - x_2} = \frac{y_1 - y_2}{x_1 - x_2} + \hat{e} = \lambda + \hat{e} \qquad \text{where } \hat{e} := \frac{e}{x_1 - x_2}$$

$$\tilde{x}_3 = \tilde{\lambda}^2 - x_1 - x_2 = \lambda^2 + 2\lambda\hat{e} + \hat{e}^2 - x_1 - x_2 = x_3 + 2\lambda\hat{e} + \hat{e}^2$$

$$\begin{aligned}\tilde{y}_3 &= -y_1 + \tilde{\lambda}\,(x_1 - \tilde{x}_3) = -y_1 + \lambda\left(x_1 - x_3 - 2\lambda\hat{e} - \hat{e}^2\right) + \hat{e}\left(x_1 - x_3 - 2\lambda\hat{e} - \hat{e}^2\right) \\ &= y_3 + \hat{e}\left(x_1 - x_3 - 2\lambda\hat{e} - \hat{e}^2 - 2\lambda^2 - \lambda\hat{e}\right) = y_3 + \hat{e}\,(x_1 - x_3 - (\hat{e} + \lambda)\,(2\lambda + \hat{e}))\,.\end{aligned}$$

The resulting faulty point $\tilde{P}_3 = (\tilde{x}_3, \tilde{y}_3)$ is a valid faulty point only if it satisfies the Weierstrass Equation (4.2), i.e., if $\tilde{y}_3^2 - \tilde{x}_3^3 - A\tilde{x}_3 - B \equiv 0 \bmod p$. This can be used to compute $\hat{e}$. We have

$$\begin{aligned}&\tilde{y}_3^2 - \tilde{x}_3^3 - A\tilde{x}_3 - B \\ &= \hat{e}\cdot\left(\hat{e} - \frac{2y_2}{x_1 - x_2}\right)\cdot\left(\hat{e}^2\cdot(x_2 - x_1) + 2\hat{e}\cdot(y_2 - y_1) + \right. \\ &\qquad \left.\frac{x_1^3 - 3x_1^2x_2 - Ax_1 + Ax_2 - 2y_2^2 + 2x_2^3 + 2y_1y_2}{x_1 - x_2}\right) + R, \\ &\quad \text{where } R = y_3^2 - x_3^3 - Ax_3 - B + \hat{e}\cdot\frac{2y_1(y_1^2 - y_2^2 - x_1^3 + x_2^3 + Ax_2 - Ax_1)}{(x_1 - x_2)^2}, \\ &= (x_2 - x_1)\cdot\hat{e}\cdot\left(\hat{e} - \frac{2y_2}{x_1 - x_2}\right)\cdot\left(\hat{e} + \lambda + \sqrt{2x_1 + x_2}\right)\cdot\left(\hat{e} + \lambda - \sqrt{2x_1 + x_2}\right), \qquad (4.11)\end{aligned}$$

by standard arithmetic (using MAPLE). Here, we have $R \equiv 0$. This becomes evident by applying the Weierstrass equation, using $y_i^2 = x_i^3 + Ax_i + B$ for all $i \in \{1, 2, 3\}$. It can be applied since we know that P_1, P_2, and the correct result P_3 are valid points on the elliptic curve. Now, Equation (4.11) shows that we have between 2 and 4 solutions for $\hat{e}$, since we assume that $x_1 \not\equiv x_2$:

$$\hat{e} \in \left\{0,\ \frac{2y_2}{x_1 - x_2},\ -\lambda \pm \sqrt{2x_1 + x_2}\right\} \qquad (4.12)$$

$$\Rightarrow\ e \in \left\{0,\ 2y_2,\ (x_1 - x_2)\cdot\left(-\lambda \pm \sqrt{2x_1 + x_2}\right)\right\}. \qquad (4.13)$$

These results for e show that one transient fault which produces a valid faulty point depends on the value of y_2, even if it is induced into y_1. This is caused by the fact that y_1 is used later for the computation of y_3 in Equation (4.9). One might expect that negating y_1 with the error $e = -2y_1$ would yield a valid point, since the valid input point $(x_1, -y_1)$ is used. However, this does not yield a valid faulty point $(\tilde{x}_3, \tilde{y}_3)$. For this to happen, the value y_1 used in Equation (4.9) would have to be faulty as well. This requires a permanent fault, which we investigate now.

Permanent Faults. Given a permanent fault in y_1, the computations of $\tilde{\lambda}$ and $\tilde{x}_3$ are the same as shown above for a transient fault, but the computation of $\tilde{y}_3$ is different. We get the faulty output $\tilde{P}_3 = (\tilde{x}_3, \tilde{y}_3)$ with the values

$$\tilde{x}_3 = x_3 + 2\lambda\hat{e} + \hat{e}^2$$

$$\begin{aligned}\tilde{y}_3 &= -y_1 - e + \tilde{\lambda}\,(x_1 - \tilde{x}_3) = y_3 - e + \hat{e}\,(x_1 - x_3 - \hat{e}\,(1 + \lambda)\,(2\lambda + \hat{e})) \\ &= \quad y_3 + \hat{e}\,(x_1 - x_3 - \hat{e}\,(1 + \lambda)\,(2\lambda + \hat{e})) - \hat{e}\,(x_1 - x_2)\,.\end{aligned}$$

As shown before, we get solutions for $\hat{e}$ by applying the Weierstrass equation (4.2), and solving the resulting equation for $\hat{e}$. The computation is completely analogous to the analysis shown above, where we describe the computation in greater detail.

We have

$$\tilde{y}_3^2 - \tilde{x}_3^3 - A\tilde{x}_3 - B = (x_1 - x_2) \cdot \hat{e} \cdot \left(\hat{e} + \frac{2y_1}{x_1 - x_2}\right) \cdot \left(\hat{e} + \lambda + \sqrt{x_1 + 2x_2}\right) \cdot \left(\hat{e} + \lambda - \sqrt{x_1 + 2x_2}\right),$$

with standard arithmetic (using Maple). The possible solutions for e are

$$e \in \left\{0,\ -2y_1,\ (x_1 - x_2) \cdot \left(-\lambda \pm \sqrt{x_1 + 2x_2}\right)\right\}. \tag{4.14}$$

As one can see in Section A.1 of Appendix A, these solutions are completely symmetric to the solutions for an attack on y_2.

4.3.1.2. Faults Induced into λ During the Computation of $x_3 = \lambda^2 - x_1 - x_2$

For faults induced into λ, we investigate three different cases. First, the variable λ could have been affected prior to its first use in Equation (4.8). The effect of the fault could be transient or permanent. But then, it could also be that the result λ^2 could have been affected. Here, we will only analyze the case when λ is faulty, the analysis of faults affecting λ^2 can be found in Section A.1 in Appendix A. In case of a fault affecting λ, the faulty value $\lambda + e$ is used and we get the faulty results

$$\begin{aligned}
& \tilde{x}_3 = (\lambda + e) - x_1 - x_2 = x_3 + 2\lambda e + e^2 \\
\text{and}\quad & \tilde{y}_3 = -y_1 + \lambda \cdot \left(x_1 - (x_3 + 2\lambda e + e^2)\right) \\
& \quad = y_3 - 2\lambda^2 e - \lambda e^2 && \text{in case of a transient fault,} \\
\text{or}\quad & \tilde{y}_3 = -y_1 + (\lambda + e) \cdot \left(x_1 - (x_3 + 2\lambda e + e^2)\right) \\
& \quad = y_3 + e\left(x_1 - x_3 - 2\lambda e - e^2 - 2\lambda^2 - \lambda e\right) \\
& \quad = y_3 + e\left(x_1 - x_3 - (2\lambda + e) \cdot (e + \lambda)\right) && \text{in case of a permanent fault.}
\end{aligned}$$

Transient Faults. Let $\hat{e} := -2\lambda e - e^2$. In this case, the resulting faulty point $\tilde{P}_3$ is equal to $(\tilde{x}_3, \tilde{y}_3) = (x_3 - \hat{e}, y_3 + \lambda\hat{e})$. In order for $(\tilde{x}_3, \tilde{y}_3)$ to be a valid point on the curve, it has to satisfy the Weierstrass Equation (4.2), i.e., it must hold that

$$\begin{aligned}
& \tilde{y}^2 \equiv \tilde{x}^3 + A\tilde{x} + B \bmod p. \\
\Rightarrow\quad & \hat{e}^3 + \hat{e}^2\left(\lambda^2 - 3x_3\right) + \hat{e}\left(3x_3^2 + A + 2y_3\lambda\right) \equiv 0 \bmod p \\
\Rightarrow\quad & \hat{e} \in \left\{0, \frac{3x_3 - \lambda^2}{2} \pm \frac{1}{2}\sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}\right\} \bmod p.
\end{aligned} \tag{4.15}$$

As $\hat{e}$ is a quadratic equation in e, this yields 6 possible solutions for e, namely

$$e \in \left\{0, -2\lambda, -\lambda \pm \sqrt{\frac{3}{2}\left(\lambda^2 - x_3\right) \pm \sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}}\right\} \bmod p. \tag{4.16}$$

Permanent Faults. For permanent faults affecting λ, the situation is the same as for transient faults in y_2 in the computation of $\lambda = (y_2 - y_1)/(x_2 - x_1)$. It has been analyzed above yielding the conditions stated in Equation (4.12):

$$e \in \left\{0,\ \frac{2y_2}{x_1 - x_2},\ -\lambda \pm \sqrt{2x_1 + x_2}\right\}$$

4.4. Attacks Inducing Undetectable Faults

In Appendix A, and with two examples in the last section, we analyze the conditions for a fault on every variable used in the affine addition formula to result in a faulty output, which cannot be detected by the standard countermeasure proposed in [BMM00] and [CJ03]. In both publications, [BMM00] and [CJ03], the authors suggest that their countermeasure, i.e., checking whether the final result is a valid point on the curve E or not, is sufficient as a countermeasure against fault attacks on elliptic curves. In this section, we will show that this claim is wrong. By inducing faults into coordinate values during a point addition, it is possible to create undetectable errors yielding valid faulty points. We will show that these attacks have a very high success probability. Therefore, the countermeasure proposed in [BMM00] and [CJ03] is rendered useless by these attacks. Moreover, this countermeasure actually supports our new attacks by eliminating all other faulty results, which do not represent valid faulty points. This allows attacks to be less precise. Once again, a countermeasure against one attack actually benefits another, a situation first described in [YKLM01a]. As noted before, we refer to the countermeasure of checking if a final point is a valid point on a curve E as the standard countermeasure.

Let x be the faulty variable changed by the adversary using some specific fault model. The effect of the fault can be expressed by the mapping

$$x \overset{\lightning}{\mapsto} \tilde{x} = x + e(x), \tag{4.17}$$

where $e(x)$ represents the error compared to the correct value x. The possible values for $e(x)$ are determined by the fault model used. First, we show that using random faults, the standard countermeasure is indeed sufficient. Afterwards, we will investigate bit faults and byte faults and show that these faults lead to a new fault type, Sign Change Faults, which is not thwarted by the standard countermeasure.

4.4.1. Faults Induced into Point Addition Using Random Faults

First, we consider the Random Fault Model defined in Definition 1.9. Here, we interpret the error term $e(x)$ from Equation (4.17) as a random variable, which is uniformly distributed in the interval $\{-x, -x+1, \ldots, 2^{l(p)}-1-x\}$. This interval is derived from the fact that $\tilde{x} \in \{0, 2^{l(p)}-1\}$. All computations are performed within the finite field $\mathbb{F}_p$, hence, we may treat $e(x)$ as a random variable from the interval $\{-x, -x+1, \ldots, 2^{l(p)}-1-x\} \bmod p = \{0, 1, \ldots, p-1\} = \mathbb{F}_p$. The error $e(x)$ is not a uniformly distributed variable in $\mathbb{F}_p$ because $2^{l(p)}$ is not a multiple of p. Values of $e(x)$ smaller than 0 or larger than $p-1$ are wrapped around by the reduction. However, as $p < 2^{l(p)} \leq 2p$, each value in $\mathbb{F}_p$ has at most two preimages in $\{-x, -x+1, \ldots, 2^{l(x)}-1-x\}$. Therefore, we have $\text{Prob}(e(x) \equiv c \bmod p) \in \{1/p, 2/p\}$. There is a bias in the distribution towards the boundary values, however, given the fact that p is a large prime, the difference between $1/p$ and $2/p$ is negligible.

For all variables analyzed in the last section, an error has to satisfy a certain condition. Given fixed points P_1 and P_2 on a fixed curve E, an error has to take at least one out of at most 6 different values in $\mathbb{F}_p$ in order to allow an undetectable fault. For example, a transient fault induced into the intermediate variable λ during the computation of x_3, results in a valid faulty point iff

$$e \in \left\{ 0, -2\lambda, -\lambda \pm \sqrt{\frac{3}{2}\left(\lambda^2 - x_3\right) \pm \sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}} \right\} \bmod p,$$

according to Equation (4.16). Here, the condition $e = 0$ is not an option for an adversary, as it returns the correct result. Therefore, for any targeted variable, the success probability to induce an undetectable fault is at most $5/p$, which is negligible. Attacks using the random fault model in the above described manner are useless for an adversary.

4.4.2. Faults Induced into Point Addition Using Bit Faults

Now, we consider the Single Bit Fault Model as defined in Definition 1.7. A fault induced into a variable x is assumed to result in an error $e(x) = \pm 2^b \bmod p$ for some $0 \leq b \leq l(p) - 1$. Therefore, such bit faults represent a very small number of possible errors. The Single Bit Fault Model generally neglects the fact that bits can only be flipped to the complementary value, i.e., only one out of the the two error values $e = 2^b$ and $e = -2^b$ is actually possible. However, this detail is usually not important in practice. Thus the total number of different faulty values for x is at most $2l(p)$. As $l(p) \approx \log(p)$, an adversary can check all possible errors by exhaustive search. This is the main advantage of bit errors compared to random errors as mentioned before in Section 1.4.2.

In the case of an error occurring in the addition routine, the task is to show that some of the conditions for the induced error e can be met using bit flips.

Judged by the complexity of the conditions for the error e, some of the conditions look easier to accomplish. The important cases are the following error conditions:

- $e = \ \ 2y_2$ (transient error in y_1 in the computation of λ, see Equation (4.13)), or
- $e = -2y_1$ (permanent error in y_1 in the computation of λ, see Equation (4.14)), or
- $e = -2\lambda$ (transient error in λ in the computation of x_3, see Equation (4.16)),

or as shown in Appendix A,

- $e = -2y_2$ (error in y_2 in the computation of λ, see Equation (A.3)),
- $e = -2y_3$ (error in y_1 in the computation of y_3, see Equation (A.10)), or
- $e = -2y_3$ (error in y_3 in the computation of y_3, see Equation (A.11)).

In the situation where the corresponding coordinate or value is a power of 2, these conditions can be met by a bit flip. For example, if $P_1 = (x_1, y_2)$ and $P_2 = (x_2, y_2)$ should be added to $P_1 + P_2$ where $y_2 = 2^b$, and the $2^{b+1}st$ bit of y_1 is 0, a bit flip of that bit in y_1 during the computation of λ yields a valid faulty point according to Equation (4.13) (see first condition above). If an adversary is able to choose the base point P in a scalar multiplication, he might be able to compute a point P such that at some step $kP + k'P$ is computed where the above conditions are met. For a concrete example, consider the standard left-to-right repeated doubling algorithm for scalar multiplication on elliptic curves, presented as Algorithm 4.5 on the next page.

If an adversary can choose a base point $P = (x_2, y_2)$ such that $y_2 = 2^b$ for some $0 \leq b < l(p)-1$, a bit flip fault can be induced in each execution of Line 4 in Algorithm 4.5. If the error $e = 2^{b+1}$ is induced into the y-coordinate of Q during the computation of the slope λ, Algorithm 4.5 computes $Q := Q - P$, which is a valid faulty result. For bit flips, we usually assume that any bit of a variable is hit with the same uniform probability, hence, after a small number of attempts, an adversary can hope to induce the desired fault. Moreover, the standard countermeasure actually helps in this attack. It will sort out all those faulty values, which do not satisfy any

of the error conditions derived in Section 4.4. Therefore, an adversary can induce faults until a faulty value is returned by the device. In this case, the adversary knows exactly how this faulty result has been created. This enables him to successfully recover the secret scalar k, given enough faulty values. This attack will be described in the next chapter. Note that the other fault types described in Section 1.3 can also be used for this attack. Additionally, it is obvious that the Chosen Bit Fault Model, Definition 1.6 can be used for this attack as well, with an even greater success probability.

Algorithm 4.5: Left-to-right Repeated Doubling on an Elliptic Curve E

Input: A point P on E, and a secret key $1 < k < \text{ord}(P)$, where n denotes the binary length of k, i.e. the number of bits of k
Output: kP on E
\# **init**
1 Set Q := $\mathcal{O}$
\# **main**
2 For i from n − 1 down to 0 do
3 Set Q := 2Q
4 If $d_i = 1$ then set Q := Q + P
5 **Output** Q

The fault type which is described here follows a certain pattern. It changes the sign of a variable. If the value y_2 is changed to $y_2 - 2y_2$, the point used is $\tilde{P}_2 = -P_2 = (x_2, -y_2)$. It is obviously a valid point, yet, it does not yield the correct result. This type of error always circumvents the standard countermeasure. We will therefore introduce it as a new fault type, Sign Change Faults. These are described in the following. We have shown that this fault type can be realized using bit faults if the base point P has a special y-coordinate. We will show in the following that Sign Change Faults can also be realized in several other ways, which shows that this fault type is realistic. In the following chapters, we will use this fault type to successfully attack elliptic curve scalar multiplication.

4.5. Sign Change Faults

In this section, we describe a new fault type, Sign Change Faults. We show how Sign Change Faults can be induced into physical devices, thus showing that the new model is practical. Attacks using this new fault model will be presented in the next chapter. This new fault type has been published as [BOS04].

Definition 4.6 (Fault Type: Sign Change Fault Type).
Let X be a variable affected by a Sign Change Fault (SCF). Then the variable is changed to the fixed value

$$X \not\mapsto \tilde{X} = -X.$$

Neither the value of X nor of $\tilde{X}$ is known to the adversary. The effect of the fault may be transient or permanent.

Sign changes can be applied to any variable in any cryptosystem. However, due to the different nature of sign changes, such attacks show very different success probabilities in practice. For a point $P = (x, y)$ on an elliptic curve $E(\mathbb{F}_p)$, where $p > 3$ is prime, a sign change is just a

change of the sign of the y-coordinate, i.e., $P \overset{\not}{\mapsto} (x, -y)$. However, if the point P is a point on an elliptic curve $E(\mathbb{F}_{2^t})$ for some t, a sign change can be expressed as $P \overset{\not}{\mapsto} (x, x - y)$. This obviously requires a much more aggressive change of the bits of the y-coordinate. For values x in $\mathbb{Z}_N$, where $N = p \cdot q$ with p, q prime as in the RSA cryptosystem, sign changes have a very similar effect as for points P over prime fields, $p \overset{\not}{\mapsto} -p = N - p \bmod N$.

At first sight, a Sign Change Fault does not seem to be easily performed in a general setting. A random change of the y-coordinate cannot hope to yield $-y$ with non-negligible probability as shown in Section 4.4.1. However, there exist several special yet common settings, where Sign Change Faults can be realized. From the above considerations, it seems that changing the y-coordinate of an elliptic curve point defined over a prime field is easier than changing a point defined over a binary field. This is indeed true, most examples for a successful induction of Sign Change Faults only apply to elliptic curves defined over prime fields. In the following, we will introduce possible attacks, which yield faults of the Sign Change Fault type. These results have been achieved in cooperation with Johannes Blömer and Jean-Pierre Seifert [BOS04].

Symmetric Representation. One possible scenario assumes that an implementation of finite field arithmetic works with positive and negative field elements. This implementation could store the sign of such a field element as a single bit together with the absolute value. In this case, attacks on that single bit are possible with a variety of fault models. As the minimal amount of memory which can be allocated by a smartcard is one byte, all fault models assuming faults in a single bit, a byte or in all bits using any fault type as defined in Section 1.3 can be used to change the sign bit. However, there are some slight drawbacks: First, attacking a byte in order to flip the least significant bit, the sign bit, is not always successful in every fault model, e.g., random faults have a success probability of 1/2. Second, a smartcard could easily check if the value of the sign byte is greater than 1 and trigger a security reset if this is the case. This would lower the success probability to 1/256, if all byte values are equally probable. Therefore, byte faults and random faults have a lower success probability, although still acceptably high.

Usually, finite field elements are not stored with an explicit sign bit, since they can be easily represented by the canonical set $\{0, 1, \ldots, p-1\}$, given a modulus p. However, it is possible to use a symmetric representation with the set $\{-(p-1)/2, \ldots, -1, 0, 1, \ldots, (p-1)/2\}$. Here numbers can be represented using one bit less, which might give a small advantage in speed as negation of elliptic curve points is very cheap. In this case, the sign bit is used and Sign Change Faults are possible. Unfortunately, a symmetric number system is hardly used in practice.

NAF-based Elliptic Curve Scalar Multiplication. The performance of most variants of the classical repeated doubling algorithm will improve if the scalar k is recoded into non-adjacent form (NAF). The 2-NAF uses digits from the set $\{-1, 0, 1\}$ and ensures that no two adjacent digits are non-zero. It achieves a higher ratio of zeros to non-zeros, which reduces the number of additions/subtractions in the resulting double-and-add-or-subtract method. For details on the NAF, see [Boo51], [Rei60], [Mac61], [Hwa79], [Kor93], [JY00], [Ott01], or [OT04]. Using the NAF, subtractions are introduced. Since negating a point on an elliptic curve simply means to change the sign of the y-coordinate, subtractions are extremely cheap operations on elliptic curves. They require at most an additional subtraction, i.e., $-x = p - x$, where p is the modulus. The savings using repeated doubling based on the NAF are 11.11% on average (see [MO90]), because the NAF has an expected 33% lower Hamming-weight than the corresponding binary expansion and it is at most one bit longer than the binary expansion. It can be created from the binary expansion or any other signed digit representation with the same digit set

in linear time (cf. [Ott01]). Other variants using a signed m-ary digit set $\{-(m-1), -(m-2), \ldots, -1, 0, 1, \ldots, m-1\}$ exist, which also yield significant savings [OT04], [Ott01]. Because of these improvements, NAF based variants of repeated doubling are often used in modern smartcards. Any of these NAF based variants of the repeated doubling algorithm is susceptible to Sign Change Faults.

The most direct attack on the NAF is to induce a SCF in one of the signed digits of the recoded scalar k. This means to change a digit from -1 to $+1$ or vice versa, whereas a 0 is not changed at all. Although there exist compact representations of signed-digit representations, cf. [JT01a], the most popular encoding described in [JY00] admits a straightforward bit-flip attack. Note that an attacker being able to induce faults according to the Chosen Bit Fault Model 1.6 has no difficulties to induce the formerly sketched SCF on the recoded k.

Another conceivable way to attack the NAF is offered by the fact that any NAF-based algorithm has to incorporate a conditional branch, where for any secret key bit $k_i = 1$ an addition is performed and for any secret key bit $k_i = -1$, a subtraction is performed. Currently, a whole zoo of physical attacks is available to either target the bit k_i (see Section 1.1) or to target the conditional decision itself (e.g., power spikes or clock glitches as described in Section 1.1). These attacks aim at forcing the conditional statement to choose the wrong branch. In case of a NAF-based version of Algorithm 4.5 (which will be described later as Algorithm 5.2), choosing the wrong branch means to add $-P$ instead of P or vice versa. This results in a transient Sign Change Fault induced into the point P. This attack is also valid for more sophisticated NAF-based repeated doubling variants, which aim to secure the basic scheme against power and timing attacks, e.g., by using dummy operations (cf. [Cor99]).

Attacks On The Crypto Co-Processor. Besides attacking the scalar multiplication algorithm on a high level, Sign Change Faults can also be realized by attacking the addition and multiplication procedures, which are used to compute point coordinates. One practical scenario for this attack relies on the internal functionality of many real-world embedded crypto co-processors supporting modular arithmetic, cf. [HP98] and [BOPV03]. Namely, in order to speed up the time-critical modular *multiplication* operation, they most often also rely on the non-adjacent form, cf. [Boo51], [Mac61], and [Kor93]. For practicability issues most often only the classical 2-NAF recoding is used. However, this time, it is not the secret scalar k, which is transformed into NAF, but factors used during the computation of $P_1 + P_2$ for two elliptic curve points P_1 and P_2 (see below). Therefore, this attack also applies to variants of the repeated doubling algorithm, which are not based on the NAF of the secret scalar multiplier k.

The crucial point that we are exploiting here is that any efficient implementation of the NAF must provide special hardware to handle negative numbers, i.e., being able to compute the 2^{th} complement of any register used as a multiplicand without time delay. Actually, considering the prototypical architecture of such an embedded crypto co-processor, cf. [BOPV03] and [Sed87], this task is trivial to solve. Namely, simply invert every bit send to the long integer arithmetic unit (ALU) and additionally add $+1$. Note that providing this functionality is a must for the modulus register in order to implement the underlying modular multiplication algorithm efficiently, cf. [Mon85], [Sed87], [WQ90], [Bar96] or [BOPV03]. For concreteness we refer to [Sed87] and its corresponding patent [Sed89]. Other prominent embedded real-world crypto co-processors with similar characteristics are described in [HP98] and [BOPV03]. Given this functionality, it can be used for an attack.

We consider such a crypto co-processor, cf. [Sed87], adding simultaneously at least three different operands with a possible sign change in one single instruction. Changing the value of

an operand to its negative, i.e., to its 2^{th} complement, one usually needs to change only one single bit among the control signals of the corresponding ALU. This is due to the fact that the ALU of most crypto co-processors is, as already explained above, designed to handle automatically the 2^{th} complement of any operand. Here, an attack can be mounted that results in an SCF.

Let us now examine the addition formulas in greater detail to see which Sign Change Faults yield valid faulty points.

Affine Addition. Let us recall the affine addition formula, Definition 4.1, for two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, both not equal to $\mathcal{O}$:

$$\begin{aligned} x_3 &:= \lambda^2 - x_1 - x_2 \\ y_3 &= -y_1 + \lambda \cdot (x_1 - x_3)\,, \\ \text{where } \lambda &:= \begin{cases} \frac{y_1 - y_2}{x_1 - x_2}. & \text{if } P1 \neq \pm P2 \\ \frac{3x_1^2 + A}{2y_1} & \text{if } P1 = P2. \end{cases} \end{aligned}$$

These formulas show that lots of load/store and exchange operations are needed in the ALU in order to compute the desired result (x_3, y_3). Therefore, attacks on the crypto co-processor during these operations may change the control signals, thus negating the loaded value. If, for example, the value y_1 is changed to $-y_1$ in the computation of λ, a valid faulty point $\tilde{P}_3$ is created according to Equation (4.14). Any variable used in the formulas must be loaded into the ALU at some time before the computation can be performed. During this preparing load/store or exchange instruction, the corresponding value must go through the ALU. While executing this operation, the handled value is susceptible to an SCF as only a single bit among the control signals must be changed to load/store or exchange the value in its target register to its negative. This yields an SCF by changing one single control signal. Note that [BCN+04] actually describes how to implement such physical attacks in practice.

A sign change in a variable x can be described as $x \not\mapsto \tilde{x} = x + e$, where $e = -2x$. We have seen four examples in Section 4.4.2, where sign changes in variables used in the affine addition formula yield valid faulty points. Two simple possibilities are described in Section A.1 in Appendix A: First, we could change the sign of y_2 in the computation of $y_1 - y_2$ while computing λ, according to Equation (A.3). Second, we could change the sign of y_3 after the correct result has been computed according to Equation (A.11). The two other possibilities depend on the implementation. According to Equation (4.14), a permanent sign change of the value y_1 prior to its first use also yields a valid faulty point. However, a permanent fault can only be achieved by an attack on the ALU if the value y_1 is only loaded/stored once, i.e., if the faulty value is used for all occurrences in the addition formula. The last possibility is a fault induced into the value λ used to compute x_3. According to Equation (4.16), a transient fault induced into λ will yield a valid faulty point $\tilde{P}_3$. However, in order to achieve a transient fault in λ, this value would have to be computed or loaded twice, such that the value used in the computation of y_3 is correct again. This is highly unlikely in an efficient implementation.

For the affine doubling, similar considerations apply. Obviously, any sign change in variables used in the computation of x_3 and y_3 yield the same result as for the affine addition. However, the computation of λ is different. Here, it is clear that any permanent change of y_1 yields a valid faulty point $\tilde{P}_3$. Whether a permanent fault can be induced depends on the implementation.

Projective Addition. Sign Change Faults via the ALU can also be induced into the projective addition formula, which has been presented in Definition 4.2. This is easy to see. Let $P_1 = (x_1 : y_1 : z_1)$, $P_2 = (x_2 : y_2 : z_2)$ be elliptic curve points in projective coordinates. For a point addition, we have

$$x_3 := r^2 - tw^2 \qquad 2y_3 := vr - mw^3 \qquad z_3 := z_1 z_2 w,$$

$$\begin{array}{lllll} \textit{where} & u_1 := x_1 z_2^2, & s_1 := y_1 z_2^3, & w := u_1 - u_2, & r := s_1 - s_2, \\ & u_2 := x_2 z_1^2, & s_2 := y_2 z_1^3, & t := u_1 + u_2, & m := s_1 + s_2, \\ \text{and} & v := tw^2 - 2x_3 & & & \end{array}$$

and for doubling the point $P_1 = (x_1 : y_1 : z_1)$:

$$\begin{array}{lllll} & x_3 := m^2 - 2s & y_3 := m \cdot (s - x_3) - t & & z_3 := 2y_1 z_1, \\ \textit{where} & m := 3x_1^2 + Az_1^4, & s := 4x_1 y_1^2, & \text{and} & t := 8y_1^4. \end{array}$$

Again, it becomes clear that lots of load/store or exchange instructions are needed in order to realize this formulas involving the coordinates of the original points P_1 and P_2. For example, an implementation could use y_1 or y_2 via previous load/store or exchange instructions as a multiplicand in the modular multiplications to compute s_0, s_1, z_3, or t.

Concluding Remarks. All attack settings introduced above can be further relaxed, taking into account that the device will check whether the final output is a valid point on the curve or not to defend against the attacks described in Section 4.2. In this case, faults which change single bits with an acceptably high probability, e.g., byte faults according to Fault Model 1.8, can be used, because unsuccessful physical attacks are winnowed by this final check.

Some of the attacks described above can be applied to curves defined over both, prime fields and binary fields. This is true for attacks on a symmetric representation and on the NAF, as attacks here do not initiate a finite field negation, but indicate that a point negation must be performed. For an attack on the crypto co-processor, however, the faulty control signal triggers a field negation, which yields a negated point only in the case of prime field based elliptic curves. For binary field based elliptic curves, a point is negated by replacing the y-coordinate by $x - y$. Hence, this is different from $-y$ unless we are in the special case where $x = 0$.

The above considerations show that Sign Change Faults can be induced in a large variety of settings, most of which represent today's modern smartcard environments. Therefore, Sign Change Faults are practical. They fulfill the condition stated in Chapter 1, demanding that fault models and fault types must be justified in practice in order to be a valuable subject of analysis. In the next chapter, we will show how Sign Change Faults can be used to successfully recover a secret scalar factor k.

5. Sign Change Faults — Attacking Elliptic Curve Cryptosystems

The results of Section 4.4.1 show that elliptic curve cryptosystems are easy to secure against most classic fault types, e.g., random faults. However, we have shown that a new fault type, Sign Change Faults, represents a practical and thus dangerous attack on elliptic curve cryptosystems. Here, faulty values, which are undetectable by the standard countermeasure described in Section 4.2, can be created with a very high chance of success. In this section, we will show how to use Sign Change Faults to break elliptic curve cryptosystems.

In all cryptosystems based on elliptic curves, e.g., the ElGamal cryptosystem and its variants, a crucial computation is the scalar multiplication of a public base point P with a secret scalar factor k. Attacks aim to recover the value k. In this chapter, we show how Sign Change Faults can be used to successfully attack a variety of implementations of elliptic curve scalar multiplication, namely the NAF-based left-to-right repeated doubling in Section 5.1, the NAF-based right-to-left repeated doubling in Section 5.2, the classic repeated doubling in Section 5.3, and Montgomery's binary method in Section 5.4. The attacks succeed to recover the secret scalar factor efficiently.

Sign Change Attacks will be the main topic of the remainder of this thesis. The next chapter, Chapter 6, is devoted to the presentation and analysis of a new countermeasure, which protects scalar multiplication algorithms against Sign Change Attacks. Chapter 7 will briefly show that Sign Change Faults can also be applied successfully to modular exponentiation, i.e., in the context of the RSA cryptosystem.

First, we need to define our fault model, which is based on the Sign Change Fault type:

Definition 5.1 (Fault Model: Sign Change Faults / (sc) fault model).
We have defined a Sign Change Fault *(SCF) as the flipping of the sign of a given variable in Definition 4.6. Sign Change Faults are defined both for elliptic curve points as well as for finite field elements:*

$$\begin{aligned} x &\overset{\not\downarrow}{\mapsto} -x \bmod p && \text{for } x \in \mathbb{F}_p \\ P &\overset{\not\downarrow}{\mapsto} -P && \text{for } P \text{ a point on some elliptic curve } E \end{aligned}$$

where

$$\begin{aligned} P = (x, y) &\overset{\not\downarrow}{\mapsto} -P = (x, -y) && P \text{ affine point on } E \text{ defined over a prime field} \\ P = (x, y, z) &\overset{\not\downarrow}{\mapsto} -P = (x, -y, z) && P \text{ projective point on } E \text{ defined over a prime field} \end{aligned}$$

We define the Sign Change Fault Model, *(sc) fault model, with the following parameters.*

location:	*complete control (can target the sign of a specific variable)*
timing:	*no control (cannot target a specific point of time, every operation and every iteration of a loop, where the targeted variable is involved, is hit with the same (uniform) probability)*
number of bits:	*random number of faulty bits (the sign is changed)*
fault type:	*Sign Change Faults (see Definition 4.6)*
probability:	*certain (every physical attack results in a fault of the desired kind)*
duration:	*permanent faults and transient faults (both variants are investigated if they differ, however, it is assumed that an adversary cannot use both kinds in one attack setting)*

5.1. Sign Change Attack on NAF-based Left-to-Right Repeated Doubling

We first present an attack on the NAF-based left-to-right repeated doubling algorithm, because this is the most important example of repeated doubling in practice. The left-to-right method is preferred over the right-to-left version, because it does not need an additional variable. As discussed in Section 2.2, there are attacks on repeated doubling algorithms, which show that it is not self-evident that an attack on one version carries over seamlessly to the other version, e.g., [FV03]. Hence, we will present both results in this thesis.

Since negating an elliptic curve point is a very cheap operation, practical applications usually use a NAF-based repeated doubling algorithm. Therefore, we put our main focus on NAF-based variants as well. We have described the NAF in some detail in Section 4.5.
Sign Changes do not depend on the underlying representation of the elliptic curve points, i.e., they apply to affine and projective coordinates as well. All of our attacks apply to both representations alike, hence, we will not assume a special representation in this chapter.

In Algorithm 5.2, we present a NAF-based left-to-right version of the well known repeated doubling algorithm. Algorithm 5.2 already implements the standard countermeasure against random fault attacks from Section 4.2 in Line 5.

Algorithm 5.2: NAF-based Repeated Doubling on an Elliptic Curve E (Left-to-Right Version)

Input: A point P on E, and a secret key $1 < k < \mathrm{ord}(P)$ in non-adjacent form, where n denotes the number of bits of k
Output: kP on E
\# **init**
1 Set Q := $\mathcal{O}$
\# **main**
2 For i from $n-1$ downto 0 do
3 Set Q := 2Q
4 If $k_i = 1$ then set Q := Q + P else if $k_i = -1$ then set Q := Q − P
5 If Q is not on E then set Q := $\mathcal{O}$
6 **Output** Q

Starting from Algorithm 5.2, we will use indices to distinguish between the variable values in the main loop. We will denote the correct final result by Q and a faulty final result by $\tilde{Q}$. To avoid ambiguity when referring to intermediate values, we rewrite Lines 3 and 4 as

3 Set $\mathsf{Q}'_\mathsf{i} := 2\mathsf{Q}_{\mathsf{i}+1}$
4 If $\mathsf{k}_\mathsf{i} = 1$ then set $\mathsf{Q}_\mathsf{i} := \mathsf{Q}'_\mathsf{i} + \mathsf{P}$
else if $\mathsf{k}_\mathsf{i} = -1$ then set $\mathsf{Q}_\mathsf{i} := \mathsf{Q}'_\mathsf{i} - \mathsf{P}$
else set $\mathsf{Q}_\mathsf{i} := \mathsf{Q}'_\mathsf{i}$

Naturally, we set $Q_n := \mathcal{O}$.

Most elliptic curves defined over prime fields, which are recommended by current standards such as ANSI, IEEE, and SECG (see Table 4.1), have prime order, i.e., they contain a prime number of points. Therefore, we will assume this property for our curves as well. This fact implies that any point $P \neq \mathcal{O}$ on E must have the same (large) prime order. We will use this assumption frequently.

We state our attack using an algorithm similar to the attack presented by Boneh, DeMillo and Lipton in [BDL01] on RSA. This attack has been analyzed and used extensively in Chapter 2. Similar to [BDL01], we need to be able to induce faults into $c = (n/m)\log(2n)$ runs on the same input (P, k, E) to recover k with probability at least $1/2$. Here, m is a parameter to be defined later. Differently from the attack in [BDL01], we need a correct result Q to verify our test cases — which is indeed a plausible assumption. We use the following result from [BDL01] to bound the number of necessary faulty outputs needed by our attack (see also Fact 2.4).

Fact 5.3 (Number of Necessary Attacks).
Let $x = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$ and let M be the set of all contiguous intervals of length $m < n$ in x. If $c = (n/m) \cdot \log(2n)$ bits of x are chosen uniformly independently at random, then the probability that each interval in M contains at least one chosen bit is at least $1/2$.

5.1.1. Sign Change Attack Targeting Q'_i in Line 4.

All of the variables in Lines 3 and 4 can be successfully targeted by a Sign Change Attack (SCA). In the following, we present a fault attack, which induces faults into the variable Q'_i in Line 4 during some loop iteration $n-1 \geq i \geq 0$. Afterwards, we will describe how to modify the attack for the other variables.

The basic idea of our attack algorithm is to recover the bits of k in pieces of $1 \leq r \leq m$ bits. Here, m is chosen to reflect a trade-off between the number of necessary faulty results derived from Fact 5.3 and the approximate amount 2^m of offline work. Throughout this chapter, we assume that $2^m \ll \#E$. To motivate the algorithm, we first assume that a faulty value $\tilde{Q}$ is given that resulted from an SCF in Q'_i. In order to describe the resulting pattern in a compact way, we introduce the following notation, similar to Definition 2.17.

Definition 5.4 (High and Low Parts of k and kP).
Let $(k_{n-1}, k_{n-2}, \ldots, k_0)$ be the binary expansion or NAF of a positive integer k and let P be a point on an elliptic curve. We define upper and lower parts of k and kP as

$$h_i(k) := \sum_{j=i}^{n-1} k_j 2^{j-i}, \qquad H_i(k) := h_i(k) \cdot P, \tag{5.1}$$

$$l_i(k) := \sum_{j=0}^{i} k_j 2^j, \qquad L_i(k) := l_i(k) \cdot P. \tag{5.2}$$

In the context of scalar multiplication, k denotes the secret scalar factor. If the number k is clear from the context, we will write h_i for $h_i(k)$ etc.

Corollary 5.5 (Transformations Between L_i And H_i).
Let k and P be defined as in Definition 5.4. With $Q := \sum_{j=0}^{n-1} k_j 2^j P$, we have the following equalities:

$$L_i(k) = Q - 2^{i+1} H_{i+1}(k) \tag{5.3}$$

$$H_i(k) = \frac{Q - L_{i-1}(k)}{2^i} \tag{5.4}$$

Proof:

We have

$$Q - 2^{i+1} H_{i+1}(k) = \sum_{j=0}^{n-1} k_j 2^j P - 2^{i+1} \sum_{j=i+1}^{n-1} k_j 2^{j-i-1} P = \sum_{j=0}^{i} k_j 2^j P = L_i(k).$$

This proves Equation (5.3), which directly yields $L_{i-1}(k) = Q - 2^i H_i(k)$. This in turn implies Equation (5.4). □

Equation (5.4) uses division, which is not explicitly defined for elliptic curves. However, we assume elliptic curves defined over fields with prime order, therefore, since $2^i \not\equiv 0 \bmod \#E$, multiplicative inverses are guaranteed to exist. Hence, division is just multiplication with the multiplicative inverse as usual. This justifies the above notation. Moreover, the right hand side of Equation (5.4) adds up the summands $k_j 2^j$ with $j \geq i$. Therefore, every summand is divisible by 2^i, hence, a multiplicative inverse is not needed if instead the summands $k_j 2^{j-i}$ are used.

In a correct run of Algorithm 5.2, the final value $Q = Q_0 = kP$ is computed as

$$\begin{aligned} Q &= 2 \cdot (2 \cdot (\ldots 2 \cdot (2 \cdot (2 \cdot (k_{n-1}P) + k_{n-2}P) + k_{n-3}P) \ldots) + k_1 P) + k_0 P \\ &= 2 \cdot \big(\ldots 2 \cdot \big(2 \cdot \big(Q'_i + k_i P\big) + k_{i-1} P\big) \ldots\big) + k_0 P \quad \text{with} \quad Q'_i = \sum_{j=i+1}^{n-1} k_j 2^{j-i} P \overset{(5.1)}{=} 2H_{i+1}(k) \\ &= 2^i Q'_i + \sum_{j=0}^{i} k_j 2^j P \quad \overset{(5.1),(5.2)}{=} \quad 2^{i+1} H_{i+1}(k) + L_i(k). \end{aligned}$$

Now it is easy to see that a faulty value $\tilde{Q}$, which results from a fault induced into Q'_i, can be written as:

$$\begin{aligned} \tilde{Q} &= -2^i Q'_i + \sum_{j=0}^{i} k_j \cdot 2^j \cdot P = -2^i \sum_{j=i+1}^{n-1} k_j 2^{j-i} P + \sum_{j=0}^{i} k_j \cdot 2^j \cdot P \\ &\overset{(5.2)}{=} -Q + 2L_i(k) \qquad (5.5) \\ &\overset{(5.3)}{=} Q - 2 \cdot 2^{i+1} H_{i+1}(k). \qquad (5.6) \end{aligned}$$

On the right hand side of Equation (5.5), the only unknown part is $L_i(k)$, which defines a multiple of P. If only a small number of the signed bits $k_0, k_1, \ldots, k_i$ used in that sum is unknown, these bits can be guessed and verified using Equation (5.5). This allows to recover the signed bits of k starting from the LSBs. Moreover, based on Equation (5.6) it is also possible to recover the signed bits of k starting from the MSBs (see Section 5.1.3). As we assume that faults

are induced uniformly at random, an adversary may choose freely between these two recovery strategies. In the following, we will use the LSB version based on Equation (5.5). We assume that Q is known. The complete attack is stated as the following algorithm.

Algorithm 5.6: The Sign Change Attack on Q'_i

Input: Access to Algorithm 5.2, n the length of the secret key $k > 0$ in non-adjacent form, $Q = kP$ the correct result, m a parameter for acceptable amount of offline work.
Output: k with probability at least $1/2$.

```
# Phase 1: Collect Faulty Outputs
 1 Set c := (n/m) · log(2n)
 2 Create c faulty outputs of Algorithm 5.2 by inducing a SCF in Q'_i for random values of i.
 3 Collect the set S = {Q̃ | Q̃ ≠ Q is a faulty output of Algorithm 5.2 on input P}.
# Phase 2: Inductive Retrieval of Secret Key Bits
 4 Set s := −1 indicating the number s + 1 of known bits of k.
 5 While (s < n − 1) do
# Compute the known LSB part 2L_s(k) of the exponent k.
 6     Set L := 2 Σ_{j=0}^{s} k_j 2^j P
# Try all possible bit patterns with length r ≤ m.
 7     For all lengths r = 1, 2, ... m do
 8         For all valid NAF-patterns x = (x_{s+1}, x_{s+2}, ..., x_{s+r}) with x_{s+r} ≠ 0 do
# Compute the test candidate T_x
 9             Set T_x := L + 2 Σ_{j=s+1}^{s+r} x_j 2^j P
# Verification Step: Verify the test candidate using Equation (5.5)
10             for all Q̃ ∈ S do
11                 if (T_x − Q̃) = Q then
12                     conclude that k_{s+1} = x_{s+1}, k_{s+2} = x_{s+2}, ..., k_{s+r} = x_{s+r},
13                     set s := s + r, and continue at Line 5
# Handle a Zero Block Failure
14     If no test candidate satisfies the verification step, then
15         assume that k_{s+1} = 0 and set s := s + 1.
16 Verify Q = kP. If this fails then output FAILURE.
17 Output k
```

Comment. Algorithm 5.6 has been abbreviated for clarity in two minor details. On the one hand, the highest iteration that suffered a SCF in Line 2 of Algorithm 5.6 needs not be the last iteration $n-1$. Let j be the maximal i such that the value Q'_i has been attacked. Algorithm 5.6 only recovers bits of k up to iteration j. Following Fact 5.3, we assume a "lucky case", which means that we assume that every interval of length m was targeted by at least one SCF. This only guarantees that $j \geq n - m$. Hence, the $m-1$ most significant bits may not be recovered. However, up to $m-1$ missing bits can easily be recovered by exhaustive search if m is small enough. If this fails, the adversary comes to the same conclusion as suggested in Line 16: the guessed bits must be wrong and the attack has failed.

Furthermore, it is clear that given n as the length of the NAF of k, Algorithm 5.6 does not need to test patterns whenever $s + r \geq n$. Here, s indicates that the $s+1$ least significant bits $k_0, k_1, \ldots, k_s$ are known. In fact, we may assume that the most significant bit of k is $k_{n-1} = 1$, otherwise n cannot be uniquely defined. Therefore, we may assume w.l.o.g. that $s + r < n - 1$.

Note that we always have $k > 0$. We also rely on the fact that $(k_0, k_1, \ldots, k_s, x_{s+1}, \ldots, x_{s+r})$ (constructed in Line 8) is always in valid NAF.

We will prove the success of Algorithm 5.6 in two lemmas. First, we will show that only a correct guess for the pattern of k can satisfy the verification step in Line 11. Then, we will show that Algorithm 5.6 will always correctly recover at least the next unknown bit of k. The second result is based on the assumption that each contiguous interval of m bits was targeted by at least one Sign Change Fault. This represents the "lucky case" of Fact 5.3. Before stating the results, we introduce Zero Block Failures. This definition is essentially the same as Definition 2.8, but here we put it in the context of Sign Change Faults on elliptic curve points.

Definition 5.7 (Zero Block Failure).
Assume that Algorithm 5.6 already recovered the $s+1$ least significant signed bits $k_0, k_1, \ldots, k_s$ of k. If the signed bits k_{s+1}, k_{s+2}, …, k_{s+r} are all zero and all Sign Change Faults that happened in iterations $s+1, \ldots, s+m$ are restricted to the first r iterations $s+1, s+2, \ldots, s+r$, the situation is called a Zero Block Failure*.*

We briefly recall the notion of Zero Block Failures, which was introduced before in the context of RSA in Section 2.1.2. A Zero Block Failure is named after the fact that errors in a block of zeros will not be detected as errors within that block. Equation (5.2) shows that for any s, $L_s(k) = L_{s+1}(k) = \ldots = L_{s+r}(k)$ for all sequences $k_{s+1} = 0, k_{s+2} = 0, \ldots, k_{s+r} = 0$. In this case, the values $\tilde{Q}_1 = -Q + 2L_s(k)$ and $\tilde{Q}_2 = -Q + 2L_{s+r}(k)$ are equal. Therefore, given $\tilde{Q} = -Q + 2L_s(k)$, Algorithm 5.6 cannot determine how many zero bits — if any — follow k_s. Hence, tailing zeros must be neglected, because their number cannot be determined correctly. This is the reason why Algorithm 5.6 only tests patterns x which end in ± 1 in Line 8. However, the fact that tailing zeros do not change the value of $\tilde{Q}$ allows us to write $\tilde{Q} = -Q + 2L_i(k)$ with $k_i \neq 0$ for some unknown i if $\tilde{Q} \neq -Q$. Moreover, if it is known that $\tilde{Q} = -Q + 2L_s(k)$, we may specify i in greater detail. In fact, we have $\tilde{Q} = -Q + 2L_i(k)$ with $i := \max\{j \mid k_j \neq 0 \wedge j \leq s\}$. The case where $\tilde{Q} = -Q$, i.e., when the maximum does not exist, can be represented by choosing $i = -1$. This simplification will be used in the following two lemmas.

First, we will investigate the case when a test pattern x satisfies the verification step. This allows Algorithm 5.6 to recover the next r bits of k.

Lemma 5.8 (No False Positives).
If Algorithm 5.6 computes a test bit pattern x such that T_x satisfies the verification step in Line 11 for some $\tilde{Q} \in S$, then x is the correct bit pattern.

Proof:

Assume that the verification step is satisfied for a given test bit pattern $x = (x_{s+1}, \ldots, x_{s+r})$ with $1 \leq r \leq m$ and $x_{s+r} \neq 0$, x in non-adjacent form. We assume that we have a false positive, i.e., the pattern x is different from the corresponding pattern of k, namely $k_{s+1}, k_{s+2}, \ldots, k_{s+r}$. Hence, there must be a faulty result $\tilde{Q} \in \mathcal{S}$ that satisfies the verification step in Line 11 together with this x. The verification step in Line 11 yields $\mathcal{O} = T_x - \tilde{Q} - Q$. We use Line 9 of Algorithm 5.6 to express T_x and Equation (5.5) to express $\tilde{Q}$ in detail. As explained above, we may assume that $\tilde{Q} = -Q + 2L_i(k)$ with $k_i \neq 0$ for some unknown $i \geq 0$. We have

$$\mathcal{O} = \left(2 \cdot \sum_{j=0}^{s} k_j 2^j P + 2 \cdot \sum_{j=s+1}^{s+r} x_j 2^j P\right) - \left(-Q + 2 \cdot \sum_{j=0}^{i} k_j 2^j P\right) - Q$$

$$= 2 \cdot \left(\underbrace{\sum_{j=0}^{s} k_j 2^j + \sum_{j=s+1}^{s+r} x_j 2^j}_{R_+} - \underbrace{\sum_{j=0}^{i} k_j 2^j}_{R_-} \right) \cdot P \quad = \quad R_x \cdot P \tag{5.7}$$

$$\text{where} \quad R_x = 2 \cdot \sum_{j=0}^{\max(i,s+r)} y_j 2^j \quad \text{and} \quad y_j = \begin{cases} 0 & \text{if } j \le \min(i,s) \\ k_j & \text{if } i < j \le s \\ (x_j - k_j) & \text{if } s < j \le \min(i, s+r) \\ x_j & \text{if } \max(i,s) < j \le s+r \\ -k_j & \text{if } s+r < j \le i. \end{cases}$$

Equation (5.7) implies that either $R_x = 0$ or R_x is a multiple of the order of P.

Case 1. Assume that $R_x = 0$. This is easily shown to be impossible. It implies that $R_+ = R_-$, i.e., both sums are valid NAF representations of the same number. Since the NAF is unique, this implies that both representations are equal. Hence, all digits are equal in contradiction to the assumption that there is as least one $x_j \neq k_j$ with $s+1 \le j \le s+r$.

Case 2. Assume that $R_x \neq 0$ is a multiple of the order of P on E. We know that $\text{ord}(P) = \#E \gg 2^m$ as $\#E$ is prime. Therefore, $\#E$ divides R_x. If $i = -1$, i.e., $\tilde{Q} = -Q$, we have $R_x \cdot P = T_x = \mathcal{O}$. As we may assume that $s + r < n - 1$ as explained above, we have $R_x < \#E$. This contradicts our assumption that $\#E$ divides R_x. If $0 \le i \le s$, we have $R_x = 2^{i+2} \cdot R'_x$ with $|R'_x| < 2^{s+r-i} < 2^{n-1-i} \le k < \#E$. Therefore, $\#E$ cannot divide R_x. If $s + 1 \le i \le s + r$, we have $R_x = 2^{s+2} \cdot R'_x$ with $|R'_x| < 2^{m+1}$. Again, $\#E$ cannot divide R_x. For $s + r < i < n - 1$, we have $R_x = 2^{s+2} \cdot R'_x$ with $|R'_x| < 2^{i-s+1} \le 2^{n-2-s+1} \le 2^{n-1} \le k < \#E$. Therefore, $\#E$ cannot divide R_x. The last case, $i = n - 1$, is impossible. It would imply that $Q'_i = \mathcal{O}$ has been attacked, where no Sign Change Fault can be induced, i.e., $\tilde{Q} = Q$. However, we explicitely prevent values $\tilde{Q} = Q$ from being members of the set $\mathcal{S}$ in Line 3 of Algorithm 5.6. Therefore, Case 2 is impossible. □

Lemma 5.7 has shown that if Algorithm 5.6 has found a test pattern x, it correctly represents the corresponding bit pattern of k. This result can be used to show that whether Algorithm 5.6 finds a test pattern x or not, it always recovers the correct next bits of k.

Lemma 5.9 (Correct Recovery).
We assume that the bits $k_0, k_1, \ldots, k_s$ of k have already been computed by Algorithm 5.6. Furthermore, we assume that a Sign Change Fault was induced into the intermediate value Q'_i in Line 4 of Algorithm 5.2 for some $i \in \{s+1, s+2, \ldots, s+m\}$.

Then, in order to recover the next signed bit k_{s+1}, Algorithm 5.6 will be in one of two cases: In the first case, it finds a test bit pattern $x = (x_{s+1}, x_{s+2}, \ldots, x_{s+r})$, $r \le m$, that satisfies the verification step in Line 11 and concludes that $k_j = x_j$ for all $s + 1 \le j \le s + r$ in Line 12. In the second case, it detects a Zero Block Failure and concludes that $k_{s+1} = 0$ in Line 15. In both cases, the conclusion is correct and between 1 and r bits of k are recovered correctly.

Proof:

We will investigate the two cases of the lemma separately.

Case 1. Assume that Algorithm 5.6 finds a test bit pattern $x = (x_{s+1}, x_{s+2}, \ldots, x_{s+r})$ that satisfies the verification step in Line 11. According to Lemma 5.8, there cannot be a false positive and x correctly represents the bit pattern of k. Therefore, the conclusion $k_j = x_j$ for all $s + 1 \leq j \leq s + r$ is correct.

Case 2. Assume that Algorithm 5.6 does not find a test bit pattern x that satisfies the verification step. In this case, a Zero Block Failure is conjectured by Algorithm 5.6 and it sets $k_{s+1} = 0$.

We assume that this conjecture is wrong. We know by the assumption in the lemma that at least one of the iterations $s + 1, s + 2, \ldots, s + m$ was targeted by a Sign Change Fault. Let $\tilde{Q} \in \mathcal{S}$ be the faulty output resulting from such an induced fault, i.e., $\tilde{Q} = -Q + 2L_i(k)$ with $s + 1 \leq i \leq s + m$ according to Equation (5.5). If the conjecture that we have a Zero Block Failure is wrong, we know by Definition 5.7 that we may choose $\tilde{Q}$ such that at least one of the bits $k_{s+1}, k_{s+2}, \ldots, k_i$ is not zero. This implies that we may write $\tilde{Q} = -Q + 2L_w(k)$ with $w := \max\{j \mid k_j \neq 0 \wedge s + 1 \leq j \leq i\}$ as explained above. Now it is easy to see that the test bit pattern $0 \neq x = (k_{s+1}, k_{s+2}, \ldots, k_{w-1}, k_w)$ of length $1 \leq r \leq m$ satisfies the verification step. This means that the value T_x defined in Line 9 of Algorithm 5.6 correctly represents $2L_w(k)$. Therefore, a valid test pattern x exists and Algorithm 5.6 will find a value for k_{s+1} in Line 12. Hence, the assumption that a Zero Block Failure is detected incorrectly must be wrong. □

The results of the previous lemmas are summarized in the following theorem. It is a straightforward result from the two previous lemmas, combined with a simple count of the operations performed by Algorithm 5.6.

Theorem 5.10 (Success of the Proposed Sign Change Attack).
Algorithm 5.6 succeeds to recover the secret scalar multiple k of bit length n in time $O(n \cdot 2^m \cdot c \cdot M)$ with probability at least $1/2$. Here, $c = (n/m) \cdot \log(2n)$ and M is the maximal cost of a full scalar multiplication or a scalar multiplication including the induction of a Sign Change Fault.

Proof:

The result of Lemma 5.9 relies on the assumption that every contiguous interval of length m was targeted by at least one Sign Change Fault in Line 2 of Algorithm 5.6. According to Fact 5.3, this assumption holds with probability $1/2$ if $c = (n/m) \cdot \log(2n)$ faulty results are collected. This requires c scalar multiplications with the ability to induce a SCF. According to Lemma 5.9, every iteration of the while loop of Algorithm 5.6 recovers either a zero bit or a test bit pattern x. It shows that this recovery is always correct if the "lucky case" of Fact 5.3 is assumed. Therefore, Algorithm 5.6 recovers between 1 and m bits in each while iteration. Therefore, at most n iterations are needed. The worst case occurs when $k = 2^{n-1}$ and no SCF was induced into Q'_{n-1} while Algorithm 5.6 created the set of faulty outputs in Line 2. As all bits but the most significant are zero, only Zero Block Failures would occur, allowing to recover only a single bit in each iteration of Algorithm 5.6.

In a single iteration, Algorithm 5.6 first computes the value L in Line 6, requiring a single scalar multiplication. Then, the algorithm tests at most all possible NAF patterns of

length m (assuming tailing zeros for shorter patterns). These are at most

$$\eta = \frac{1}{3} \cdot \left(2^{m+2} + (-1)^{m+1}\right) = O(2^m),$$

according to [EK90]. Every test bit pattern x yields a test candidate T_x using one scalar multiplication in Line 9. Obviously, some speedups could be applied, e.g., storing T_x allows to compute a new T_x using a single addition. Note that the precomputation of L in Line 6 already represents a speed up. For each test candidate T_x, at most c point additions and comparisons need to be done in Line 11. If all possible bits of k have been recovered, a last full scalar multiplication must be applied to differentiate between Zero Block Failures and a real failure. This yields total costs of

$$O(c \cdot \tilde{M} + n \cdot (M + 2^m \cdot c \cdot (A + C)) + M),$$

where M denotes the cost of a full scalar multiplication, $\tilde{M}$ the cost of a full scalar multiplication with the ability to induce a fault, A the cost of a point addition, and C the cost of a point comparison. To present a short result, we simply treat the addition and comparison cost of Line 11 together as one full scalar multiplication. We also assume that $M = \tilde{M}$.

In this case, the worst case running time of Algorithm 5.6 is $O((c + n2^m c + 1) \cdot M)$ where $c = (n/m) \cdot \log(2n)$ and M is the maximal cost of a full scalar multiplication or a scalar multiplication including the induction of a Sign Change Fault. □

5.1.2. Other Attacks

The basic idea of Algorithm 5.6 can also be used for Sign Change Attacks, which induce faults into any other variable used inside the loop in Algorithm 5.2. We have explicitly investigated an attack targeting the variable Q'_i in Line 4 of Algorithm 5.2 in the previous section. However, if the variable Q'_i in Line 3 is affected by an induced fault, it has the exact same effect. For the other variables, we will briefly present the results in the following.

Attacks targeting Q_{i+1} in Line 3. A fault induced into Q_{i+1} in Line 3 changes the sign of Q_{i+1}. This yields $Q'_i = 2 \cdot (-Q_{i+1}) = -Q'_i$, which is the same error as if Q'_i would be affected by a fault induced into Line 3 or 4. Hence, all results from the previous section apply.

A fault induced into the variable Q_{i+1} in Line 3 of Algorithm 5.2 can be used for another Sign Change Attack if it is possible to target just one of the two copies of Q_{i+1}. In this case, a Sign Change Fault induced into Q_{i+1} yields $Q'_i = \mathcal{O}$ and

$$\tilde{Q} = L_i(k).$$

Algorithm 5.6 can be used to recover the bits of k starting from the LSBs with the following modified verification step:

11(a) if $\left(2\tilde{\mathsf{Q}} - \mathsf{T_x}\right) = \mathcal{O}$ then

This attack does not need to know a correct result Q. We can prove that this verification step does not yield any false positives with Lemma 5.8. The proof is completely analogous with opposite signs in Equation (5.7). Lemma 5.9 and Theorem 5.10 hold unchanged. With Equation (5.3), we get $L_i(k) = Q - 2^{i+1}H_{i+1}(k)$. Hence, it is also possible to recover the bits starting

from the MSBs.

Attacks targeting the Base Point P in Line 4. Another attack can be mounted if it is possible to change the sign of the base point P in Line 4. Since the base point is used in every iteration, we need to differentiate between a transient fault, where only once a faulty value $-P$ is used, and a permanent fault, which changes all subsequent usages of P as well.

If the error induced into P is transient, we have

$$\tilde{Q} = Q - 2k_i 2^i P.$$

Algorithm 5.6 can be used if the verification step is modified to

11(b) if $\left(\tilde{\mathsf{Q}} + 2\mathsf{x}_{s+1}2^{s+1}\mathsf{P}\right) = \mathsf{Q}$ then

Obviously, changing only the modification step here means to break a butterfly on a wheel, as we may save a lot of operations since neither L nor T_x are needed. If $k_i = 0$, this fault has no effect. If $k_i = \pm 1$, we have a single bit changed, which is essentially the same as a sign change of the bit k_i of the secret scalar. A drawback of this attack is that we need a block size of $m = 1$, hence, an increased number of faulty outputs need to be collected. In addition, a lot of Zero Block Failures will occur, as every zero in the NAF of k yields a Zero Block Failure. The result of Lemma 5.8 about the impossibility of false positives holds for this special case as well, simply assume that all bits besides k_i and x_{s+r} are equal to zero in Equation (5.7) and change the signs of R_+ and R_-. Hence, Theorem 5.10 holds for this case as well, showing that the attack succeeds to recover k in polynomial time.

If the error induced into P is permanent, we have

$$\tilde{Q} = Q - 2L_i(k).$$

Algorithm 5.6 can also be used for this case with only a sign change in the verification step, i.e.,

11(c) if $\left(\tilde{\mathsf{Q}} + \mathsf{T}_\mathsf{x}\right) = \mathsf{Q}$ then

Obviously, the two lemmas and the theorem of Section 5.1 hold as well.

Attacks targeting P should be considered much less probable than attacks targeting any other variable. As P is the base point, it is stored in long term memory, where it is loaded from every time. Moreover, permanent errors could easily be detected by re-checking.

An Attack targeting Q_i in Line 4. A fault induced into Q_i in Line 4 yields $\tilde{Q} = 2L_{i-1}(k) - Q$. This yields the exact same attack as described in Algorithm 5.6.

5.1.3. Recovering The Bits Starting From The MSB

As mentioned above, the bits of k can also be recovered starting from the MSB, based on Equation (5.6), which states that

$$\tilde{Q} = Q - 2 \cdot 2^{i+1} H_{i+1}(k).$$

Algorithm 5.6 has to be modified accordingly, yet, the claims about the success probability hold for this attack as well, with some modifications. For completeness, we present the modified algorithm here. The modified parts have been highlighted. Note that we assume that $k_{n-1} = 1$.

Algorithm 5.11: The Sign Change Attack on Q'_i, MSB Version

Input: Access to Algorithm 5.2, n as the length of the secret key $k > 0$ in non-adjacent form, $Q = kP$ the correct result, m a parameter for acceptable amount of offline work.
Output: k with probability at least 1/2.

\# **Phase 1:** Collect Faulty Outputs
1 Set $c := (n/m) \cdot \log(2n)$
2 Create c faulty outputs of Algorithm 5.2 by inducing a SCF in Q'_i for random values of i.
3 Collect the set $\mathcal{S} = \{\tilde{Q} \mid \tilde{Q} \neq Q$ is a faulty output of Algorithm 5.2 on input $P\}$.
\# **Phase 2:** Inductive Retrieval of Secret Key Bits
4 Set $s := n - 1$ representing the index of the highest known bit of k.
5 While $(s > 0)$ do
\# Compute the known MSB part $2H_s(k)$ of the exponent k.
6 Set $H := \sum_{j=s}^{n-1} k_j 2^{j-s} \cdot P$
\# Try all possible bit patterns with length $r \leq m$.
7 For all lengths $r = 1, 2, \ldots m$ do
8 For all possible test bit patterns $x = (x_{s-r}, x_{s-r+1}, \ldots x_{s-1})$ with $x_{s-r} \neq 0$ do
\# Compute the test candidate T_x
9 Set $T_x := 2^r \cdot H + \sum_{j=s-r}^{s-1} x_j 2^{j-s+r} \cdot P$
\# Verification Step: Verify the test candidate using Equation (5.6)
10 for all $\tilde{Q} \in \mathcal{S}$ do
11 if $\left(\tilde{Q} + 2^{s-r+1} T_x\right) = Q$ then
12 conclude that $k_{s-1} = x_{s-1}, k_{s-2} = x_{s-2}, \ldots k_{s-r} = x_{s-r}$,
13 set $s := s - r$ and continue at Line 5
\# Handle a Zero Block Failure
14 If no test candidate satisfied the verification step, then
15 assume that $k_{s-1} = 0$ and set $s := s - 1$.
16 Verify $Q = kP$. If this fails output FAILURE.
17 **Output** k

The value T_x computed in Line 9 represents the value $H_{s-r}(k)$ if $x_j = k_j$ for all $s - r \leq j \leq s - 1$. Algorithm 5.11 has been abbreviated in the same minor details as stated in the comment following Algorithm 5.6. The proof of correctness, i.e., the equivalent of Lemma 5.8, Lemma 5.9, and Theorem 5.10, can be carried out in a pattern similar to the proof for the LSB version. In the following, we present an updated version of Lemma 5.8.

Lemma 5.12 (False Positives).
If Algorithm 5.11 computes a test bit pattern x such that T_x satisfies the verification step in Line 11 for some $\tilde{Q} \in \mathcal{S}$, then x represents the correct bit pattern of k.

Proof:

Assume that the verification step is satisfied for a given test bit pattern $x = (x_{s-1}, \ldots, x_{s-r})$ with $1 \leq r \leq m$ and $x_{s-r} \neq 0$, x in non-adjacent form. We assume that we have a false positive, i.e., the pattern x is different from the corresponding pattern of k, namely

$k_{s-1}, k_{s-2}, \ldots, k_{s-r}$. Hence, there must be a faulty result $\tilde{Q} \in \mathcal{S}$ that satisfies the verification step in Line 11 together with this x. The verification step in Line 11 yields $\mathcal{O} = 2^{s-r+1} \cdot T_x + \tilde{Q} - Q$. We use Line 9 of Algorithm 5.11 to express T_x and Equation (5.6) to express $\tilde{Q}$ in detail, i.e., $\tilde{Q} = Q - 2^{i+2}H_{i+1}(k)$ with $k_{n-1} = 1$ for some $i \geq 0$. We have

$$\begin{aligned}
\mathcal{O} = \;& 2^{s-r+1} \cdot T_x + \tilde{Q} - Q \\
= \;& \left(2^{s+1}H_s(k) + 2^{s-r+1} \sum_{j=s-r}^{s-1} x_j 2^{j-(s-r)} \cdot P \right) + \left(Q - 2^{i+2}H_{i+1}(k)\right) - Q \\
= \;& 2 \left(\underbrace{\sum_{j=s}^{n-1} k_j 2^j + \sum_{j=s-r}^{s-1} x_j 2^j}_{R_+} \right) \cdot P - 2 \cdot \left(\underbrace{\sum_{j=i+1}^{n-1} k_j 2^j}_{R_-} \right) \cdot P \;=\; 2 \cdot R_x \cdot P \qquad (5.8)
\end{aligned}$$

where $R_x = \sum_{j=\min(s-r,i+1)}^{n-1} y_j 2^j$ and $y_j = \begin{cases} 0 & \text{if } j \geq \max(s, i+1) \\ k_j & \text{if } s \leq j < i+1 \\ (x_j - k_j) & \text{if } \max(s-r, i+1) \leq j \leq s-1 \\ x_j & \text{if } s-r \leq j < \min(s, i+1) \\ -k_j & \text{if } i+1 \leq j < s-r. \end{cases}$

We know that $k_{n-1} = 1$. Hence, both R_+ and R_- represent positive numbers. Equation (5.8) is satisfied if $R_+ - R_- \equiv 0 \bmod \mathrm{ord}(P) = \#E$. We investigate the two cases $R_+ - R_- = 0$ over the integers and $R_+ - R_- \equiv 0 \bmod \#E$, $R_+ - R_- > 0$ separately.

Case 1. Assume that $R_+ - R_- = 0$ over the integers. As both R_+ and R_- are valid non-adjacent unique representations of two integers, say r_+ and r_-, we have equality iff $r_+ = r_-$. In this case, however, the two non-adjacent forms must be equal, since every number has a unique non-adjacent form representation. Therefore, we cannot have a false positive. This contradicts our assumption.

Case 2. Assume that $R_x \neq 0$ is a multiple of the order of P on E. Therefore, $\#E$ divides R_x. Algorithm 5.11 starts with $s = n-1$, since the most significant bit $k_{n-1} = 1$ is known. As we are working with the NAF of k, this implies that $k_{n-2} = 0$. We have $R_+ = 2^{n-1} + R'_+$ with $|R'_+| < 2^{n-2}$. If $i+1 \leq n-1$, the value R_- is not equal to 0, since it adds up at least k_{n-1}. Again, we know that $k_{n-1} = 1$ and $k_{n-2} = 0$, therefore, we can write $R_- = 2^{n-1} + R'_-$ with $|R'_-| < 2^{n-2}$. In this case, we know that $|R_+ - R_-| = |R'_+ - R'_-| < 2^{n-1} \leq k < \#E$. This implies that if a false positive is encountered, we must have $i+1 = n$. However, the latter implies that $R_- = 0$, i.e., $\tilde{Q} = Q$. Since such entries have been explicitly excluded in Line 3 of Algorithm 5.11, this case may never happen, either. Consequently, Case 2 is impossible as well, which proves the lemma. □

The results of Lemma 5.9 and Theorem 5.10 hold for the MSB recovery with only minor modifications concerning the indices. Therefore, we only state the final result without proof.

Theorem 5.13 (Success of the Proposed Sign Change Attack).
Algorithm 5.11 succeeds to recover the secret scalar multiple k of bit length n in time $O(n{\cdot}2^m{\cdot}c{\cdot}M)$ with probability at least $1/2$. Here, $c = (n/m){\cdot}\log(2n)$ and M is the maximal cost of a full scalar multiplication or a scalar multiplication including the induction of a Sign Change Fault.

5.2. Sign Change Attack on NAF-based Right-to-Left Repeated Doubling

As we have explained above, attacks on one version of repeated doubling are not guaranteed to apply to other variants of repeated doubling as well in general. Therefore, we will investigate in this section, how a Sign Change Attack can be mounted on the twin of the left-to-right repeated doubling algorithm (Algorithm 5.2): the right-to-left repeated doubling algorithm. We will see that for Sign Change Attacks, both versions are susceptible for these attacks in the same manner. We start with stating the right-to-left repeated doubling algorithm as Algorithm 5.14.

Algorithm 5.14: NAF-based Repeated Doubling on an Elliptic Curve E (Right-to-Left Version)

Input: A point P on E, and a secret key $1 < \mathsf{k} < \mathsf{ord(P)}$ in non-adjacent form, where n denotes the number of bits of k
Output: kP on E
\# **init**
1 Set Q := $\mathcal{O}$
2 Set R := P
\# **main**
3 For i from 0 to n − 1 do
4 If $\mathsf{k_i} = 1$ then set Q := Q + R else if $\mathsf{k_i} = -1$ then set Q := Q − R
5 Set R := 2R
6 If Q is not on E then set Q := $\mathcal{O}$
7 **Output** Q

We will investigate Sign Change Attacks against Algorithm 5.14 in the same manner as in Section 5.1.1. Again, we will use indices to differentiate between the variable values in the main loop. We denote the correct final result by Q and a faulty final result by $\tilde{Q}$. To refer to intermediate values clearly, we rewrite Lines 4 and 5 as

4 If $\mathsf{k_i} = 1$ then set $\mathsf{Q_i} := \mathsf{Q_{i-1}} + \mathsf{R_{i-1}}$
else if $\mathsf{k_i} = -1$ then set $\mathsf{Q_i} := \mathsf{Q_{i-1}} - \mathsf{R_{i-1}}$
else set $\mathsf{Q_i} := \mathsf{Q_{i-1}}$
5 Set $\mathsf{R_i} := 2\mathsf{R_{i-1}}$

5.2.1. Sign Change Attack Targeting Q_i in Line 4.

In a correct run of Algorithm 5.14, the final value $Q = Q_{n-1} = kP$ is computed as

$$\begin{aligned} Q &= k_0 2^0 \cdot P + k_1 2^1 \cdot P + \ldots + k_{n-1} 2^{n-1} \cdot P \\ &= \big(\underbrace{k_0 2^0 \cdot P + k_1 2^1 \cdot P + \ldots + k_i 2^i \cdot P}_{=Q_i = L_i(k)} \big) + k_{i+1} 2^{i+1} \cdot P + \ldots + k_{n-1} 2^{n-1} \cdot P \end{aligned}$$

$$= Q_i + k_{i+1}2^{i+1} \cdot P + \ldots + k_{n-1}2^{n-1} \cdot P \overset{(5.1),(5.2)}{=} L_i(k) + 2^{i+1}H_{i+1}(k).$$

Now it is easy to see that a faulty value $\tilde{Q}$, which results from an attack inducing a SCF in Q_i, can be written as:

$$\begin{aligned}\tilde{Q} &= -Q_i + 2^{i+1}H_{i+1}(k) = -L_i(k) + 2^{i+1}H_{i+1}(k) \\ &\overset{(5.2)}{=} Q - 2L_i(k) \qquad (5.9) \\ &\overset{(5.3)}{=} -Q + 2 \cdot 2^{i+1}H_{i+1}(k). \qquad (5.10)\end{aligned}$$

Equations (5.9) and (5.10) allow to recover the bits of k starting from the LSBs or starting from the MSBs. They are the same formulas as Equation (5.5) and (5.6) for the attack on the left-to-right version, only with the opposite sign. Let us first assume that we wish to recover the bits of k starting from the LSBs, i.e., using Equation (5.9). In this case, the Sign Change Attack targeting Q_i in the right-to-left repeated doubling algorithm is completely analogous to Algorithm 5.6. The only difference is a new verification step, that respects the opposite sign. The modified Line 11 simply replaces $T_x - \tilde{Q}$ be $T_x + \tilde{Q}$, i.e.,

11(d) if $\left(\mathsf{T_x} + \tilde{\mathsf{Q}}\right) = \mathsf{Q}$ then

Not surprisingly, the analysis of this modified attack is completely equivalent to the analysis of the original Algorithm 5.6. The results of Lemma 5.8 (no false positives), of Lemma 5.9 (correct recovery), and of Theorem 5.10 (success probability and runtime) carry over seamlessly, even the proofs change only negligibly. Therefore, we omit the lemmas and the proofs here and state the final result only.

Theorem 5.15 (Success of the Proposed Sign Change Attack).
Algorithm 5.6 with the modified verification step

11(d) if $\left(\mathsf{T_x} + \tilde{\mathsf{Q}}\right) = \mathsf{Q}$ then

succeeds to recover the secret scalar multiple k of bit length n in time $O(n \cdot 2^m \cdot c \cdot M)$ with probability at least $1/2$. Here, $c = (n/m) \cdot \log(2n)$ and M is the maximal cost of a full scalar multiplication or a scalar multiplication including the induction of a Sign Change Fault.

5.2.2. Other Attacks

The basic idea of Algorithm 5.6 can also be used for Sign Change Attacks inducing faults into any other variable used inside the loop in Algorithm 5.14. Once again, we state the results for all variables individually.

An Attack targeting Q_{i-1} in Line 4. A fault induced into Q_{i-1} in Line 4 yields $\tilde{Q} = Q - 2L_{i-1}(k)$, hence, Algorithm 5.6 can also be used for this case with the same modification step as in Theorem 5.15.

Attacks targeting R_{i-1} and R_i in Lines 4 and 5. The variable values of R_{i-1} and R_i are used twice, once for the computation of Q_i and once for the computation of R_i (or Q_{i+1} and R_{i+1}, respectively). Therefore, faults induced into R_{i-1} in Line 4 or R_i in Line 5 are different depending on whether the induced fault is a transient fault or a permanent fault. Since a fault

in R_i has the same effect as a fault in R_{i-1}, just in a different iteration of the loop, we do not need to consider both scenarios separately. We will only investigate attacks targeting R_{i-1}.

Transient Faults. In case of a transient fault induced into R_{i-1} in Line 4, the intermediate value Q_i is changed to $\tilde{Q}_i = Q_{i-1} - k_i R_{i-1} = Q_i - 2k_i R_{i-1}$. Since we assume a transient fault, the correct value for R_{i-1} is used when the value R_i is computed in Line 5. Therefore, the remaining computation is correct, and we get

$$\tilde{Q} = Q_i - 2k_i R_{i-1} + \sum_{j=i+1}^{n-1} k_j 2^j P = Q - 2k_i R_{i-1} = Q - 2^{i+1} k_i \cdot P, \qquad \text{since} \quad R_i = 2^{i+1} P.$$

Therefore, Algorithm 5.6 can be used with a modified verification step

11(e) $\qquad$ if $\left(\tilde{\mathsf{Q}} + \mathsf{x}_{s+1} 2^{s+2} \mathsf{P}\right) = \mathsf{Q}$ then

However, a larger number of faulty values must be collected as these Sign Change Faults require a block length of $m = 1$. For bit values $k_i = 0$, the attack has no effect, for other bit values it represents a bit flip of the secret scalar factor k. This case represents the same situation as an attack on the base point P in Line 4 of Algorithm 5.2, which has been described in greater detail in Section 5.1.2.

A transient fault induced into R_{i-1} in Line 5 has the same effect as a permanent fault induced into R_i in Line 5. This is due to the fact that R_{i-1} is no longer used after Line 5 and the value $R_i = 2R_{i-1}$ is computed with a wrong value, i.e., $R_i = 2 \cdot (-R_{i-1}) = -R_i$.

Permanent Faults. In Line 4 of Algorithm 5.14, the variable R_{i-1} takes the value $R_{i-1} = 2^i P$. If it is affected by an induced fault, the value changes to $\tilde{R}_{i-1} = -2^i P$, so subsequent values of R also have their sign changed if the change is permanent. We assume that the fault changes the sign of R_{i-1} before it is used to create Q_i. Ultimately, the faulty final result $\tilde{Q}$ will be

$$\tilde{Q} = Q_{i-1} - \sum_{j=i}^{n-1} k_j 2^j P = -Q + 2Q_{i-1} = -Q + 2L_{i-1}(k). \tag{5.11}$$

Equation (5.11) yields the same formula as a fault induced into Q'_i in the left-to-right version, it is essentially the same as Equation (5.5), upon which Algorithm 5.6 was based on. The only difference is that in Equation (5.11), the value $L_{i-1}(k)$ is used and in Equation (5.5) the value $L_i(k)$. However, this makes absolutely no difference for the Sign Change Attack, Algorithm 5.6. Moreover, due to Equation (5.3), we may write $\tilde{Q} = -Q + 2(Q - 2^i H_i(k)) = Q - 2^{i+1} H_i(k)$, which allows a recovery starting from the MSBs as well.

A permanent fault induced into R_{i-1} in Line 5 has the same effect as a permanent fault induced into R_i in Line 4 in the next iteration. Therefore, the final faulty output can be described as $\tilde{Q} = -Q + 2L_i(k)$. This is the equation Algorithm 5.6 was build upon.

Attacks targeting the base parameters R_{-1} and Q_{-1} Faults induced into Q_{-1} cannot yield faulty outputs. We assume that the point $\mathcal{O}$ is a special point which cannot be affected by a Sign Change Fault. The effect of a fault induced into $R_{-1} = P$ may be transient or permanent. If it is transient, it's effect has been investigated above, as it represents an attack on R_{i-1} in Line 4 for $i = 0$. The result is $\tilde{Q} = Q - 2k_0 P$. This can be used to recover the bit k_0. A successful permanent sign change of R_{-1} yields $\hat{Q} = -Q$. This can be computed by an adversary easily

even without access to the device. Therefore, it cannot yield any information the adversary did not already possess.

5.2.3. Recovering The Bits Starting From The MSB

From Equation (5.10), we know that an attack on the intermediate variable Q_i also works with an approach, that recovers the bits of k starting from the MSBs. We can use the equation

$$\tilde{Q} \overset{(5.10)}{=} -Q + 2 \cdot 2^{i+1} H_{i+1}(k).$$

As this equation is essentially the same as the one derived for an attack on Q'_i in the left-to-right repeated doubling algorithm, all results from Section 5.1.3 carry over to this case as well with a slightly modified verification step in Algorithm 5.11, i.e.,

11(f) $\quad$ if $\left(2^{s-r+1}\mathsf{T_x} - \tilde{\mathsf{Q}}\right) = \mathsf{Q}$ then

The same result holds analogously for the other intermediate variables.

5.3. Attacks on the Standard Repeated Doubling Algorithms

Sections 5.1 and 5.2 have shown that the NAF-based repeated doubling algorithms can easily be attacked using the Sign Change Fault Model. However, this also holds for the classic repeated doubling algorithms. We can use Algorithms 5.2 and 5.14 for repeated doubling based on the binary expansion as well, with the only effect that the "if $\mathsf{k_i} = -1$" statement is not used at all. Since we know that $k > 0$, both algorithms compute the correct result, no matter if k is given in NAF or in the classic binary expansion.

For the attack algorithm (Algorithm 5.6), the test bit pattern x has to be chosen differently, as x may now take any unsigned bit pattern with $x_{s+r} = 1$. Since none of the proofs in Section 5.1.1 relies on any special property of the NAF, they carry over seamlessly to the binary expansion as well. Therefore, the results presented for NAF-based repeated doubling hold for binary expansion based repeated doubling as well.

We will now investigate another method for repeated doubling, which is fundamentally different from the methods presented so far: Montgomery's binary method.

5.4. Sign Change Attack on Montgomery's Binary Method

Another multiplication method for elliptic curve scalar multiplication has been proposed by Montgomery in [Mon87]. The algorithm will be referred to as *Montgomery's Binary Method*. It has been developed in the context of integer factorization, and we present it as Algorithm 5.16. However, it is also an improvement of the repeated squaring algorithm, e.g., Algorithm 5.14, in the context of timing and power attacks. A plain implementation of Algorithm 5.14 is highly susceptible to both timing and power attacks, because the main loop incorporates an addition and a doubling iff the secret key bit is non-zero, and only a doubling if the secret key bit is zero. This difference may be detected by timing measurements or analysis of the power profile (cf. [Koc96b], [Sch00a], [KJJ99]). Montgomery's binary method always incorporates an addition and a doubling, independent of the value of the secret key bit. Therefore, neither timing nor power attacks can differentiate between the two cases. The addition is the same in both cases, i.e., the same values are added, it only differs in the target variable where the result is stored. This

information is not accessible by timing or power measurements. For the doubling, it is assumed that neither timing nor power measurements reflect the character of the point to be doubled unless it is the point at infinity. Therefore, Algorithm 5.16 is a good candidate for a secure algorithm. However, the same fact which immunizes it against timing and power attacks, i.e., the uniformity of timing and power consumption, is also a drawback. It uses $l(k) - 1$ additions and $l(k)$ doublings regardless of the pattern of the secret scalar k. This represents the worst case of the repeated doubling method. Since the NAF may require one more bit to represent a number compared to the binary expansion, Montgomery's binary method is always based on the standard binary expansion.

We will show that Sign Change Faults can also be used to break Montgomery's binary method, therefore, it is not secure against Sign Change Attacks. First, let us describe Montgomery's binary method. We also assume that $k > 0$, i.e., $k_{n-1} = 1$.

Algorithm 5.16: Montgomery's binary method

Input: P an elliptic curve point, k an integer with n := l(k).
Output: kP on E
1 Set P1 := P
2 Set P2 := 2P
3 for i from n − 2 downto 0 do
4 if ($d_i = 0$) then set P2 := P1 + P2 and P1 := 2P1
5 if ($d_i = 1$) then set P1 := P1 + P2 and P2 := 2P2
6 **Output** P1

5.4.1. Preliminaries

For an SCA on Algorithm 5.16 we need to be able to express a faulty result in terms of a correct result and either lower bits or higher bits only. In order to do so, we first need more information about intermediate values of the variables, which we want to target and change by a Sign Change Fault.

Similar to the attacks presented in the previous sections, we will use indices to distinguish between the variable values in the main loop. We will denote the correct final result by Q and a faulty final result by $\tilde{Q}$. To avoid ambiguity when referring to intermediate values, we rewrite Lines 4 and 5 as

4 if ($d_i = 0$) then set $P2_{(i)} := P1_{(i+1)} + P2_{(i+1)}$ and $P1_{(i)} := 2P1_{(i+1)}$
5 if ($d_i = 1$) then set $P1_{(i)} := P1_{(i+1)} + P2_{(i+1)}$ and $P2_{(i)} := 2P2_{(i+1)}$

We have stated Algorithm 5.16 in the way usually used in the literature, with the two intermediate variables $P1$ and $P2$. In order to avoid confusion between point numbers and indices, we write the indices in brackets. This notation slightly differs from the notation of the previous sections, although, we are confident that this will make the formulas easier to read. We extend this notation to denote by $P1_{(n-1)}$ and $P2_{(n-1)}$ the values of $P1$ and $P2$ after Line 2. The following corollary gives detailed information about how the intermediate values in Algorithm 5.16 are computed. These results will be the basis for the attack presented in the next section.

Corollary 5.17 (Intermediate values of $P1$ and $P2$ in Algorithm 5.16).
Let $P1$ and $P2$ be the variables in Algorithm 5.16, where $P1_{(i)}$ and $P2_{(i)}$ denote intermediate values of $P1$ and $P2$. Then the following claims hold for Algorithm 5.16.

1. *Let* $\Delta := P2_{(n-1)} - P1_{(n-1)} = P$. *Then the relation*

$$P2_{(i)} - P1_{(i)} = \Delta$$

is maintained by Algorithm 5.16 for all $n-1 \geq i \geq 0$. *The intermediate values* $P1_{(i)}$ *and* $P2_{(i)}$ *can be expressed as*

$$P1_{(i)} := 2^{n-1-i} \cdot P1_{(n-1)} + \sum_{j=i}^{n-2} k_j 2^{j-i} \Delta \tag{5.12}$$

$$P2_{(i)} := P1_{(i)} + \Delta. \tag{5.13}$$

In Algorithm 5.16, we have $\Delta = P1_{(n-1)} = P$. *In case we have* $k > 0$, *i.e.,* $k_{n-1} = 1$, *Equation* (5.12) *simplifies to*

$$P1_{(i)} = h_i(k) \cdot P = H_i(k). \tag{5.14}$$

2. *If the starting values* $P1_{(n-1)} = P$ *and* $P2_{(n-2)} = 2P$ *are replaced by arbitrary values* $P1_{(n-1)} = a \cdot P$ *and* $P2_{(n-2)} = b \cdot P$, *for some* $a, b \in \mathbb{N}$, *Equations* (5.12) *and* (5.13) *hold with* $\Delta := P2_{(n-1)} - P1_{(n-1)}$. *For all intermediate values, the relation* $P2_{(i)} - P1_{(i)} = \Delta$ *is satisfied.*

Proof:

We will show the claims by induction. We start with $\Delta := P2_{(n-1)} - P1_{(n-1)}$ for arbitrary values of $P1_{(n-1)}$ and $P2_{(n-1)}$. Therefore, the relation $\Delta := P2_{(i)} - P1_{(i)}$ is satisfied for $i = n-1$ by definition. For $i = n-1$, we have

$$P1_{(i)} = P1_{(n-1)} = 2^{n-1-i} P_{(n-1)} + \sum_{j=i}^{n-2} k_j 2^{j-i} \Delta$$

$$\text{and} \quad P2_{(i)} = P2_{(n-1)} = P1_{(n-1)} + P2_{(n-1)} - P1_{(n-1)} = P1_{(n-1)} + \Delta$$

by definition. This represents the start of the induction.

Prior to iteration $i < n-1$, we have $\Delta = P2_{(i+1)} - P1_{(i+1)}$ and $P1_{(i+1)} = 2^{n-1-(i+1)} \cdot P1_{(n-1)} + \sum_{j=i+1}^{n-2} k_j 2^{j-i-1} \Delta$ and $P2_{(i+1)} = P1_{(i+1)} + \Delta$ by the inductive assumption. The values $P1_{(i)}$ and $P2_{(i)}$ are then determined by the bit value of k_i.

Case 1. If $k_i = 0$, we have

$$P1_{(i)} = 2P1_{(i+1)}$$

$$\text{and} \quad P2_{(i)} = P1_{(i+1)} + P2_{(i+1)} = 2P1_{(i+1)} + \Delta = P1_{(i)} + \Delta.$$

Given that $P1_{(i+1)} = 2^{n-1-i-1} \cdot P1_{(n-1)} + \sum_{j=i+1}^{n-2} k_j 2^{j-i-1} \Delta$, we have

$$P1_{(i)} = 2^{n-1-i} \cdot P1_{(n-1)} + \sum_{j=i+1}^{n-2} k_j 2^{j-i} \Delta + k_i 2^{i-i} \Delta = 2^{n-1-i} \cdot P1_{(n-1)} + \sum_{j=i}^{n-2} k_j 2^{j-i} \Delta.$$

It also implies that $P2_{(i)} - P1_{(i)} = P1_{(1)} + \Delta - P1_{(i)} = \Delta$.

Case 2. Similarly, if $k_i = 1$, we have

$$\begin{aligned} P1_{(i)} &= P1_{(i+1)} + P2_{(i+1)} && = 2P1_{(i+1)} + \Delta \\ \text{and} \quad P2_{(i)} &= 2P2_{(i+1)} = 2\left(P1_{(i+1)} + \Delta\right) && = 2P1_{(i+1)} + 2\Delta = P1_{(i)} + \Delta. \end{aligned}$$

Given that $P1_{(i+1)} = 2^{n-1-i-1} \cdot P1_{(n-1)} + \sum_{j=i+1}^{n-2} k_j 2^{j-i-1}\Delta$, we have

$$\begin{aligned} P1_{(i)} = 2P1_{(i+1)} + \Delta &= 2^{n-1-i} \cdot P1_{(n-1)} + \sum_{j=i+1}^{n-2} k_j 2^{j-i}\Delta + k_i 2^{i-i}\Delta \\ &= 2^{n-1-i} \cdot P1_{(n-1)} + \sum_{j=i}^{n-2} k_j 2^{j-i}\Delta. \end{aligned}$$

Again, we also have $P2_{(i)} - P1_{(i)} = P1_{(i)} + \Delta - P1_{(i)} = \Delta$.

Altogether, Cases 1 and 2 prove the second claim of Corollary 5.17. For the first claim, we know that Algorithm 5.16 selects the initial values $P1_{(n-1)} = P$ and $P2_{(n-1)} = 2P$, i.e., $\Delta = P$. We also assume that $k_{n-1} = 1$. Hence, Equation (5.12) simplifies to $P1_{(i)} = h_i(k) \cdot \Delta = H_i(k)$ according to Definition 5.4. □

5.4.2. Sign Change Attack Targeting $P1_{(i+1)}$

Before stating the attack algorithm, we will first show how a sign change in a specific step i can be expressed if only all higher bits $k_{i+1}, k_{i+2}, \ldots, k_{n-1}$ or all lower bits $k_{i-1}, k_{i-2}, \ldots, k_0$ are known. This expression is the basis for an attack in the manner introduced by Boneh, DeMillo and Lipton in [BDL01].

For the fault model, we consider the Sign Change Fault Model introduced in Definition 5.1. We first assume that the intermediate variable $P1_{(i+1)}$ has been affected by a fault for some $0 \le i \le n-2$, i.e., in the loop. We assume that this is a permanent fault, i.e., all subsequent uses of the variable $P1_{(i+1)}$ are faulty as well. Algorithm 5.16 shows that $P1_{(i+1)}$ is used twice if $d_i = 0$, hence, permanent and transient faults are different. We will state results about a transient fault attack in Section 5.4.4. According to the fault model, the sign of $P1_{(i+1)}$ is flipped by a Sign Change Fault, i.e., $P1_{(i+1)} \overset{\not\downarrow}{\mapsto} -P1_{(i+1)}$. The value $P1_{(i+1)}$ is the value of $P1$, which is used on the right hand side of Lines 4 and 5 of Algorithm 5.16, i.e., the values $P1_{(i)}$ and $P2_{(i)}$ are computed using $P1_{(i+1)}$ and $P2_{(i+1)}$ (as defined in Corollary 5.17). Let $\tilde{Q}$ be an output of Algorithm 5.16 that resulted from a computation where $P1_{(i+1)} \overset{\not\downarrow}{\mapsto} -P1_{(i+1)}$. In this case, we call $\tilde{Q}$ an output $\tilde{Q}$ *with fault position* i. The naming is chosen in this way because the first iteration which works with a faulty value is iteration i.

Lemma 5.18 (Sign Change Faults in Algorithm 5.16).
If an adversary changes the sign of the intermediate variable $P1_{(i+1)}$ permanently in Algorithm 5.16, the algorithm works with a faulty value $P1_{(i+1)} = -H_{i+1}(k)$ and the correct value $P2_{(i+1)} = H_{i+1}(k) + P$. The final output $\tilde{Q}$ can be expressed as

$$\tilde{Q} = \left(2l_i(k) - 2^{i+1}\right) \cdot H_{i+1}(k) + L_i(k). \tag{5.15}$$

$$= \left(\frac{l_i(k)}{2^i} - 1\right) \cdot (Q - L_i(k)) + L_i(k) \qquad \textit{(LSB version)} \tag{5.16}$$

$$= (1 + 2h_{i+1}(k)) \cdot \left(Q - 2^{i+1}H_{i+1}(k)\right) - 2^{i+1}H_{i+1}(k) \qquad \textit{(MSB version).} \tag{5.17}$$

Proof:

We assume that the sign change takes place in the variable $P1_{(i+1)}$. According to Equations (5.13) and (5.14), this means that the algorithm computes $P1_{(i)}$ and $P2_{(i)}$ using the two values $\tilde{P}1_{(i+1)} = -H_{i+1}(k)$ and $P2_{(i+1)} = H_{i+1}(k) + P$. After that change, Algorithm 5.16 computes a result with the two start values $\tilde{P}1_{(i+1)}$ and $P2_{(i+1)}$ and the remaining bit pattern $k_i, k_{i-1}, \ldots, k_0$, i.e., the iteration $i+1$ takes the position of $n-1$ in Equation (5.12). Since the fault was assumed to be permanent, only one value for $\tilde{P}1_{(i+1)}$ is used. Hence, we can apply Corollary 5.17 to derive the formulas for $\tilde{Q}$. According to Corollary 5.17, we have $\Delta := P2_{(i+1)} - \tilde{P}1_{(i+1)} = 2H_{i+1}(k) + P$, and the final faulty result is

$$
\begin{aligned}
\tilde{Q} &\overset{(5.12)}{=} 2^{i+1} \cdot \tilde{P}1_{(i+1)} + \sum_{j=0}^{i} k_j 2^j \Delta \quad = \quad -2^{i+1} H_{i+1}(k) + l_i(k) \cdot (2H_{i+1}(k) + P) \\
&= \left(2l_i(k) - 2^{i+1}\right) \cdot H_{i+1}(k) + L_i(k) \qquad \text{which is Equation (5.15)} \\
&\overset{(5.4)}{=} \left(2l_i(k) - 2^{i+1}\right) \cdot \left(\frac{Q - L_i(k)}{2^{i+1}}\right) + L_i(k) \\
&= \left(\frac{l_i(k)}{2^i} - 1\right) \cdot (Q - L_i(k)) + L_i(k) \qquad \text{which is Equation (5.16)} \\
&\overset{(5.3)}{=} \left(2l_i(k) - 2^{i+1}\right) \cdot H_{i+1}(k) + \left(Q - 2^{i+1} H_{i+1}(k)\right) \\
&= 2l_i(k) H_{i+1}(k) + Q - 2^{i+2} H_{i+1} \quad = \quad 2L_i(k) h_{i+1}(k) + Q - 2^{i+2} H_{i+1} \\
&\overset{(5.3)}{=} 2h_{i+1} \left(Q - 2^{i+1} H_{i+1}\right) + Q - 2^{i+2} H_{i+1}(k) \\
&= \left(1 + 2h_{i+1}(k)\right) \cdot \left(Q - 2^{i+1} H_{i+1}(k)\right) - 2^{i+1} H_{i+1} \qquad \text{which is Equation (5.17).}
\end{aligned}
$$

□

Given the formulas of Lemma 5.18, we can now state the algorithm for an SCA on Montgomery's binary method. Note that we assume throughout this chapter, that $\text{ord}(P) = \#E$ is prime, hence, all multiplicative inverses of nonzero field elements exist modulo $\text{ord}(P)$. This allows to use Equation (5.16) without restrictions. The SCA Algorithm 5.19 is different from the Sign Change Attacks presented in the previous sections. This version allows the occurrence of false positives. Therefore, the algorithm works in blocks of m bits and tests all bit patterns, which are build from a known pattern and at most m random bits. This idea is similar to the idea employed in Algorithm 2.36, describing the LSB recovery strategy for an attack on the standard left-to-right repeated squaring algorithm. Similarly, there may be several conflicting patterns, which all satisfy the verification step. Among these patterns are the correct one and possibly several false positives. All these verified candidates are kept in a set K. In the next round of m bits, all possible bit patterns from that set K are used as the "known" LSB part of k. We will show in the remainder of this section, that the runtime of Algorithm 5.19 is polynomial in n. We present the LSB version of the algorithm. The MSB version follows in the same manner as it has been described in Section 5.2. By $y \circ x$, we denote concatenation of bit strings, i.e., we have $y \circ x = (y_0, y_1, \ldots, y_u, x_0, x_1, \ldots, x_t)$ for $y = (y_0, y_1, \ldots, y_u)$ and $x = (x_0, x_1, \ldots, x_t)$. In Algorithm 5.19, we denote by ϵ the empty bit pattern consisting of no bits.

Algorithm 5.19: Sign Change Attack on $\mathsf{P1}_{(\mathsf{i}+1)}$ in Algorithm 5.16

Input: Access to Algorithm 5.16, n as the binary length of the secret key k, $\mathsf{Q} = \mathsf{kP}$ the correct result, m a parameter for acceptable amount of offline work.

Output: k with probability at least 1/2.

\# **Phase 1:** Collect Faulty Outputs

Set $\mathsf{c} := (\mathsf{n}/\mathsf{m}) \cdot \log(2\mathsf{n})$

Create c faulty outputs of Algorithm 5.16 by inducing SCFs in $\mathsf{P1}_{(\mathsf{i}+1)}$ for random values of i.

Collect the set $\mathcal{S} = \{\tilde{\mathsf{Q}} |\ \tilde{\mathsf{Q}} \neq \mathsf{Q}$ is a faulty output of Algorithm 5.16 on input P$\}$.

\# **Phase 2:** Inductive Retrieval of Secret Key Bits

Set $\mathsf{s} := 0$ representing the current candidate bit length between $\mathsf{s} \cdot \mathsf{m} + 1$ and $(\mathsf{s}+1) \cdot \mathsf{m}$

Set $\mathsf{K} := \{\epsilon\}$ to contain candidates for verified/"known" LSB-parts of k

While $(\mathsf{s} \cdot \mathsf{m} < \mathsf{n} - 1)$ do

  for all patterns $\mathsf{y} \in \mathsf{K}$ do

    Set $\mathsf{u} := \mathsf{l}(\mathsf{y})$ \# the algorithm ensures that $(\mathsf{s}-1) \cdot \mathsf{m} < \mathsf{u} \leq \mathsf{s} \cdot \mathsf{m}$

    Set $\mathsf{K}' = \emptyset$ \# to contain candidates of length between $\mathsf{s} \cdot \mathsf{m} + 1$ and $(\mathsf{s}+1) \cdot \mathsf{m}$

    For all r with $\mathsf{s} \cdot \mathsf{m} < \mathsf{u} + \mathsf{r} \leq (\mathsf{s}+1) \cdot \mathsf{m}$ do

      For all test bit patterns $\mathsf{x} = (\mathsf{x}_{\mathsf{u}+1}, \mathsf{x}_{\mathsf{u}+2}, \ldots \mathsf{x}_{\mathsf{u}+\mathsf{r}})$ of length $\mathsf{r} < 2\mathsf{m}$ do

\# Compute the test candidate T_x using the pattern $\mathsf{y} \circ \mathsf{x}$ and Equation (5.16)

$$L := \sum_{j=0}^{u} y_j 2^j + \sum_{j=u+1}^{u+r} x_j 2^j$$

$$T_x := \left(L/2^{u+r} - 1\right) \cdot (Q - L \cdot P) + L \cdot P$$

\# Verification Step: Verify the test candidate using Equation (5.16)

For all $\tilde{\mathsf{Q}} \in \mathcal{S}$ do

If $\left(\mathsf{T}_\mathsf{x} = \tilde{\mathsf{Q}}\right)$ then

Add $\mathsf{y} \circ \mathsf{x} = (\mathsf{y}_0, \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{u}, \mathsf{x}_{\mathsf{u}+1}, \ldots, \mathsf{x}_{\mathsf{u}+\mathsf{r}})$ to the set K'

  Set $\mathsf{K} := \mathsf{K}'$ and $\mathsf{s} := \mathsf{s}+1$ and continue at Line 6

\# **Phase 3:** Compute k from all candidates of length $\mathsf{u} > \mathsf{n} - 1 - \mathsf{m}$

For all $\mathsf{y} \in \mathsf{K}$ do

  Set $\mathsf{u} := \mathsf{l}(\mathsf{y})$

  Set $\mathsf{r} := \mathsf{n} - 1 - \mathsf{u}$

  For all test bit patterns $\mathsf{x} = (\mathsf{x}_{\mathsf{u}+1}, \mathsf{x}_{\mathsf{u}+2}, \ldots \mathsf{x}_{\mathsf{u}+\mathsf{r}})$ of length $\mathsf{r} < \mathsf{m}$ do

    If $\left(\sum_{j=0}^{u} y_j 2^j + \sum_{j=u+1}^{n-1} x_j 2^j\right) \cdot P = Q$ then **Output** $(\mathsf{y}_0, \mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{u}, \mathsf{x}_{\mathsf{u}+1}, \ldots, \mathsf{x}_{\mathsf{n}-1})$

Output FAILURE

We have shown in Lemma 5.18, Equation (5.16), that a SCF induced into $P1_{(s+r)}$ is correctly emulated with T_x given the correct bit pattern $x_j = k_j$, for all $s+1 \leq j \leq s+r$. Algorithm 5.19 has been abbreviated in two minor details, similar to the Sign Change Attack algorithms in the previous sections. We explicitly handle the recovery of the most significant m bits. Hence, we do not ensure that $u + r < n - 1$. However, since $k_{n-1} = 1$, we only need to recover the bits k_0 to k_{n-2} in practice. As explained in the comment following Algorithm 5.6, we may therefore always assume $u + r < n - 1$ without imposing additional restrictions. This assumption will

become handy in some proofs later.

Another detail is connected to the results about false positives. We will show that the sum over all cardinalities of the round sets K has an upper bound for special values of y. This bound will allow us to conclude a polynomial running time. Additionally, we will show that if there should ever be more values present in the algorithm than that upper bound, the value k can be computed directly. Therefore, a full version of the attack algorithm would need to count and store all pairs $(\tilde{Q}, y)$ for the case that the upper bound is ever reached. Since this case is extremely improbable in practice, we omit this safety net in favor of clarity. This case will become clear in the following section.

When attacking Montgomery's Binary Method, we do not encounter Zero Block Failures. We use the same definition of $L_{u+r}(k)$ as in previous sections, therefore, a block of zeros, e.g., $k_{u+1} = k_{u+2} = \ldots = k_{u+r}$ still yields $L_{u+r}(k) = L_u(k)$ for any choice of $r \geq 0$. However, the faulty value $\tilde{Q}$ as defined in Equation (5.16) is computed using the multiplicative inverse of 2^{u+r}. This is independent of the bit values of any k_i, hence, a zero block has no special effect on the computation of $\tilde{Q}$. This is different from the effect observed in, e.g., Section 5.1.1, and leaves us without a Zero Block Problem.

5.4.3. False Positives

Differently from the attacks on the NAF-based or binary expansion based repeated squaring algorithms in the previous sections, false positives may occur in principle for Montgomery's Binary Method. The form of a faulty result described in Equation (5.16) shows that the computed intermediate scalar factors may become larger than the group order $\#E$. Hence, false positives may occur. In this section, we will show that this fact does not prevent the use of our attack algorithm. Although false positives slow down the attack algorithm, this loss in efficiency only adds a polynomial factor in the worst case. Hence, the attack is still an efficient algorithm. In practice, the algorithm behaves much better if k is chosen randomly. Numerous experiments never witnessed the occurrence of a false positive.

The possibility of false positives has triggered the strategy to collect possible bit patterns in the set K. Since we always assume that every contiguous interval of length m contains at least one fault position, every set K in every iteration will always contain the correct bit pattern. However, also false positives may be accepted by the verification step in Line 15. Therefore, they also serve as "known" lower parts of k and are used to create and test new candidates. If some of their candidates satisfy the verification step again, or new false positives arise, we might have false positives in every iteration of the main loop. This is different from, e.g., Algorithm 5.6, where the first positive ends the loop of choosing x, since only the correct pattern can satisfy the verification step. At the end, we need to test all possible candidates k' of $n-1$ bits to see if they satisfy $k'P = Q$. In the following, we will first investigate the implications of a false positive to be able to bound their number afterwards. We will show that the number of false positives cannot render our algorithm useless by imposing an exponential workload.

Definition 5.20 (Verification Function, Valid Guesses and False Positives).
Let $y = (y_0, y_1, \ldots, y_u)$, $u \geq 0$, be a bit pattern and let $\tilde{Q}$ be a faulty final result from the set $\mathcal{S}$ collected in Line 2 of Algorithm 5.19. The verification function *Ver is defined as*

$$Ver(\tilde{Q}, y) = \begin{cases} true & if \quad \tilde{Q} = \left(\frac{l_u(y)}{2^u} - 1\right) \cdot (Q - L_u(y)) + L_u(y) \\ false & otherwise. \end{cases} \tag{5.18}$$

We call any bit pattern $y = (y_0, y_1, \ldots, y_u)$, $0 \leq u$, with $Ver(\tilde{Q}, y) = true$ a valid guess *for the corresponding bit pattern of k. If the bit patterns of y and k are not equal, i.e., if $y \neq (k_0, k_1, \ldots, k_u)$, i.e., if we have $y_j \neq k_j$ for at least one $0 \leq j \leq u$, we call y a* false positive.

We have $l_u(y) = \sum_{j=0}^{u} y_j 2^j$ according to Definition 5.4. The verification function has been derived from the verification step of the Sign Change Attack, i.e., Line 15 of Algorithm 5.19. This in turn is based on Equation (5.16), which describes the faulty output $\tilde{Q}$ in terms of the lower bits of k. In Algorithm 5.19, a guessed pattern $y \circ x = (y_0, y_1, \ldots, y_u, x_{u+1}, x_{u+2}, \ldots, x_{u+r})$ is assumed to correctly represent the $u+r+1$ least significant bits of k if $\mathrm{Ver}(\tilde{Q}, y \circ x) = \text{true}$ for at least one value $\tilde{Q} \in \mathcal{S}$. Definition 5.20 allows to state the following results about properties of valid guesses.

Lemma 5.21 (Properties of Valid Guesses).
Let $x = (x_0, x_1, \ldots, x_t)$ and $y = (y_0, y_1, \ldots, y_u)$ be two bit patterns with $0 \leq t \leq u \leq n-2$. Let $\tilde{Q} \in \mathcal{S}$ be a faulty final result collected in Line 2 of Algorithm 5.19. If $Ver(\tilde{Q}, x) = true$ and $Ver(\tilde{Q}, y) = true$, then we have one of the following cases:

1. *The two patterns x and y are equal, i.e., $x = y$ and $t = u$, or*
2. *we have $u > t$ and*

$$\frac{l_t(x)}{2^t} - \frac{l_u(y)}{2^u} \equiv \frac{1}{2^u} \cdot \left(2^{u-t} l_t(x) - l_u(y)\right) \equiv 0 \bmod \#E,$$

   i.e., the pattern y is the same as the pattern x shifted $u-t$ bits to the left, and reduced modulo $\#E$, or
3. *we have $u \geq t$ and $2^{u-t} l_t(x) - l_u(y) \not\equiv 0 \bmod \#E$ and k can be computed directly as*

$$k = \frac{2^{u+1}\left(l_u(y) - l_t(x)\right) + 2^{u-t} l_t(x)^2 - l_u(y)^2}{2^{u-t} l_t(x) - l_u(y)} \bmod \#E. \tag{5.19}$$

Proof:

The point $\tilde{Q}$ is a faulty final result, hence, we know that $\tilde{Q} = \kappa \cdot P$ for some unknown $0 \leq \kappa < \#E$. We assume that Case 1 is not satisfied, i.e., that we have $x \neq y$. If $\mathrm{Ver}(\tilde{Q}, x) = \text{true}$ and $\mathrm{Ver}(\tilde{Q}, y) = \text{true}$, we know that

$$\begin{aligned}
\kappa &\equiv \left(\frac{l_t(x)}{2^t} - 1\right) \cdot (k - l_t(x)) + l_t(x) \bmod \#E \\
&\equiv \left(\frac{l_u(y)}{2^u} - 1\right) \cdot (k - l_u(y)) + l_u(y) \bmod \#E \\
\Rightarrow \quad \left(\frac{l_t(x)}{2^t} - \frac{l_u(y)}{2^u}\right) \cdot k &\equiv -\frac{l_u(y)^2}{2^u} + 2 l_u(y) + \frac{l_t(x)^2}{2^t} - 2 l_t(x) \\
\Rightarrow \quad \left(2^{u-t} l_t(x) - l_u(y)\right) \cdot k &\equiv 2^{u-t} l_t(x)^2 - l_u(y)^2 + 2^{u+1}\left(l_u(y) - l_t(x)\right). \qquad (5.20)
\end{aligned}$$

If $t = u$, we have $l_u(y) \neq l_t(x)$, because $x \neq y$ was assumed. In this case, we have $0 \leq 2^{u-t} l_t(x) = l_t(x) < 2^{n-1} < \#E$ and $0 \leq l_u(y) < 2^{n-1} < \#E$, which implies that $2^{u-t} l_t(x) - l_u(y) = l_t(x) - l_u(y) \not\equiv 0 \bmod \#E$. Hence, in this case, k can be computed

directly using Equation (5.19), since $\#E$ is prime and all multiplicative inverses of non-zero values exist.

If $t < u$, we need to ensure that the denominator in Equation (5.19) exists. If it is non-zero modulo $\#E$, k can be computed using Equation (5.19). This proves Case 3. If the denominator was $0 \bmod \#E$, we have $l_u(y) \equiv 2^{u-t} l_t(x) \bmod \#E$. As shown above, we even know that $l_u(y) = 2^{u-t} l_t(x)$ over the integers, however, our argument holds modulo $\#E$ as well, since the multiplication by 2^{u-t} is a bijection modulo $\#E$. Since $l_u(y)$ is equal to $l_t(x)$ multiplied with a power of two, we know that the pattern of y is a shift of the pattern of x by $u - t$ positions and reduced modulo $\#E$. □

Lemma 5.21 is the basis for determining an upper bound on the number of false positives. In principle, the occurrence of false positives may lead to an exponential blow-up of the attack algorithm. In every block of m bits, all bit patterns $y \circ x$ are collected if they yield $\mathrm{Ver}(\tilde{Q}, y \circ x) =$ true for some $\tilde{Q} \in \mathcal{S}$. Every valid guess is used as a guess for the least significant bits of k in the next block of m bits, where up to $2m - 1$ additional bits are chosen. If we get 2 valid guesses in each block and for each possible choice of the lower significant bits of k, we get up to 2^n bit patterns to test. This number of tests is exponential in the size of k, which would render our attack useless. However, Lemma 5.21 shows that for a large fraction of these possible valid guesses, i.e., for all pairs of valid guesses that satisfy Case 3, k follows immediately. It remains to show that the number of pairs of valid guesses, which satisfy Case 2, can be sufficiently bounded. We do not take care about Case 1, since our algorithm never tests the same bit pattern on the same faulty result $\tilde{Q}$. The following lemma will show that there can be at most polynomially many valid guesses, which do not yield k directly. This implies that we can check all of these polynomially many valid guesses and do not suffer an exponential blow-up.

Lemma 5.22 (Bounding the Number of Valid Guesses).
Assume that $c \cdot (n-2) + 1$ different valid guesses $\{x_1, x_2, \ldots, x_{c(n-2)+1}\}$ are given, where $c = (n/m)\log(2n)$. Then there are at least two values x_i, x_j, $i \neq j$, and a faulty result $\tilde{Q}$, such that $Ver(\tilde{Q}, x_i) = true = Ver(\tilde{Q}, x_j)$ and Case 3 of Lemma 5.21 allows to compute k.

Proof:

The lemma can be proven by a straightforward application of the pigeon hole principle.

Let $(\tilde{Q}, x)$ be a pair such that $0 \leq t \leq n-2$ for $t := l(x)$ and $\mathrm{Ver}(\tilde{Q}, x) =$ true. Let $\{y_{(1)}, y_{(2)}, \ldots, y_{(\sigma)}\}$ be a set of values such that $u_i := l(y_{(i)}) \leq n-2$, $x \neq y_{(i)}$ and $\mathrm{Ver}(\tilde{Q}, y_{(i)}) =$ true for all $1 \leq i \leq \sigma$. Additionally, let $y_{(i)} \neq y_{(j)}$ for all $i \neq j$.

We define

$$\sigma_0 := \max\{\sigma \mid \text{there exists a set } Y = \{y_{(1)}, y_{(2)}, \ldots, y_{(\sigma)}\} \text{ which contains no pair } (y_{(i)}, y_{(j)}) \text{ such that Case 3 of Lemma 5.21 allows to recover } k \text{ directly}\}.$$

The number σ_0 can be determined by counting the number of possible values $y_{(i)}$ such that Case 2 of Lemma 5.21 applies. Let $y \in Y$. Let us assume w.l.o.g. that $u := l(y) > t$. Otherwise, switch the role of the two patterns. Case 2 of Lemma 5.21 applies if $2^{u-t} l_t(x) = l_u(y)$. Therefore, y must have the same upper t bits as x and must have only zero bits for the lower $u - t$ bits. For every u, there is only one such pattern y. Since $0 \leq u \leq n-2$ and $u \neq t$, we have at most $n-2$ possible patterns for y such that Case 2 applies. Hence,

$\sigma_0 \leq n-2$. Since we have at most $\min(n, c = (n/m) \cdot \log(2n)) \leq c$ different values for $\tilde{Q}$, there are at most $c \cdot (n-2)$ possible valid guesses such that Case 3 never applies. One more, and k can be computed directly. This proves the claimed upper bound. □

With the two Lemmas 5.21 and 5.22, we can now derive an asymptotic bound for the overall costs until Algorithm 5.19 recovers the complete bit pattern of k.

Theorem 5.23 (Success of the Proposed Sign Change Attack).
Algorithm 5.19 succeeds to recover the secret scalar multiple k of bit length n in time $O(n \cdot 2^{2m} \cdot c \cdot M)$ with probability at least $1/2$. Here, $c = (n/m) \cdot \log(2n)$ and M is the maximal cost of a full scalar multiplication or a scalar multiplication including the induction of a Sign Change Fault.

Proof:

We will investigate the costs of all of the three phases of Algorithm 5.19 separately.

Phase 1. In the first phase, the total costs are fixed. Here, $c = (n/m)\log(2n)$ Sign Change Faults must be induced into full scalar multiplications.

Phase 2. In the second phase, the innermost loop has to be performed for all bit patterns $y \in K$. We will first derive the costs for an arbitrary but fixed choice of $y \in K$ with $u := l(y)$. It is clear from Algorithm 5.19 and a trivial application of induction that all $y \in K$ satisfy $(s-1) \cdot m < u \leq s \cdot m$. Algorithm 5.19 starts with $K = \{\epsilon\}$, i.e., $u = 0$. For any y, all bit patterns $y \circ x$, with $r := l(x)$, are tested, such that $s \cdot m < u + r \leq (s+1) \cdot m$. Hence, all y have a number of bits from a certain interval of length m. This implies that the longest x, which might be tested, has length $2m-1$. This happens if $u = (s-1) \cdot m + 1$ and $u + r = (s+1) \cdot m$. Since we test all bit patterns of all sizes less or equal to $2m-1$, we need to test at most $2^{2m} - 2$ different bit patterns for each choice of y.

For each pair (y, x), Algorithm 5.19 computes a value T_x, which can be done using $(u+r)$ field additions and $2(u+r)$ field multiplications to compute L, followed by 1 field division (e.g., via the Extended Euclidean Algorithm), 1 field subtraction and 2 elliptic curve scalar multiplications, 1 elliptic curve subtraction, and 1 elliptic curve addition. To present a short result, we treat the costs of all elliptic curve operations (multiplication, addition, and subtraction) as equal to the cost of a full scalar multiplication. We do the same for the costly operation of a field inversion, and assume that all needed field additions, subtractions and multiplications can be computed in the time needed for a single full scalar multiplication. This is a realistic assumption, since the same amount of field operations has to be performed in such an elliptic curve scalar multiplication as well. We get total costs of 6 full scalar multiplications to compute T_x. Afterwards, we need one elliptic curve point comparison to verify the guess T_x. We also treat this as a full scalar multiplication.

Therefore, the inner loop of Algorithm 5.19 (Lines 7 to 16) requires at most $7 \cdot (2^{2m} - 2)$ full scalar multiplications for each choice of y.

The total number of times the inner loop has to be executed depends on the overall number of y in all round sets K. We know that the correct guesses for all $\tilde{Q}$ yield valid guesses in some round set K, so we have at least $\lceil n/m \rceil$ many y if every investigated block of length m has been targeted by at least one Sign Change Fault. This happens with probability at

least 1/2. Every time the verification step finds a matching pair, i.e., a faulty final result $\tilde{Q}$ and a bit pattern $y \circ x$, this result goes into the next round set K'. Hence, there will be as many different values y to be handled altogether as there are valid guesses in the verification step. Lemma 5.22 showed that there are at most $c \cdot (n-2)$ many valid guesses without the possibility to recover k directly. One more may be needed to identify a pair of two valid guesses, which allows to recover k using Case 3 from Lemma 5.21.

Lemma 5.21 showed that the attack algorithm, Algorithm 5.19, can recover k much faster than normal if two bit patterns x and y as defined in Lemma 5.21, Case 3, occur. In order to catch this case, Algorithm 5.19 has to compare every new valid guess to the known guesses. If a valid guess is the first valid guess for a given faulty final result $\tilde{Q}$, it simply needs to be stored, otherwise, it needs to be determined whether a previous result satisfies Case 2 or Case 3 of Lemma 5.21. A naive implementation could choose to save all valid guesses to compare the terms $l_u(y)/2^u$ and $l_t(x)/2^t$ for all pairs of valid guesses (y, x) as defined in Lemma 5.21. This would require additional memory and computation time. However, these tests can be performed much faster.

Assume that three valid guesses for the same value $\tilde{Q}$ are given, i.e., $x = (x_0, x_1, \ldots, x_t)$, $y = (y_0, y_1, \ldots, y_u)$ and $z = (z_0, z_1, \ldots, z_v)$ such that $\mathrm{Ver}(\tilde{Q}, x) = \mathrm{Ver}(\tilde{Q}, y) = \mathrm{Ver}(\tilde{Q}, z) =$ true. If x and y satisfy Case 2 of Lemma 5.21, i.e., k cannot be recovered directly since $l_t(x)/2^t \equiv l_u(y)/2^u \bmod \#E$, we only need to compare z to x. If $l_t(x)/2^t \equiv l_v(z)/2^v \bmod \#E$, we also have $l_u(y)/2^u \equiv l_v(z)/2^v \bmod \#E$. The same holds for $l_t(x)/2^t \not\equiv l_v(z)/2^v \bmod \#E$. Therefore, we need to store at most one bit pattern x for all $\tilde{Q}$. To do so, we need to store at most $c \cdot (n-1)$ additional bits. Storing all values of $\tilde{Q}$ requires at most $c \cdot 2l(p)$ bits if the values $\tilde{Q}$ are affine points which are stored using the maximal length of $l(p)$ for each finite field coordinate. Since we have $k < \#E \leq p + 1 + 2\sqrt{p}$ according to Hasse's Theorem [Sil00], we have $n \leq l(p) + 1$ and the additional memory requirements vanish in the O-notation.

We also do not get a significant additional amount of operations. If we store $\tilde{Q}$ and the first occurrence of a valid guess for $\tilde{Q}$ as a pair, we have that valid guess at hand when we get a match of a new test candidate in Line 15 of Algorithm 5.19. We need to perform the verification step for every value $\tilde{Q}$ anyway. Therefore, no additional time is required for searching for matching valid guesses. The comparison of a known valid guess x defined as above with a new valid guess for the same $\tilde{Q}$, say y defined as above, requires to compute $l_t(x)/2^t - l_u(y)/2^u$. This can be done in the time needed to perform 3 additional full scalar multiplications (2 to bound the time to compute $1/2^t$ and $1/2^u$ and one for the remaining finite field operations and the comparison to zero). Therefore, the additional work for this check is negligible.

Hence, Phase 2 requires at most

$$(c \cdot (n-2) + 1) \cdot \left(10 \cdot (2^{2m} - 2)\right) = O\left(2^{2m} \cdot \frac{n^2}{m} \cdot \log(2n)\right)$$

times the time to perform a full scalar multiplication.

Phase 3. Phase 3 requires to test at most 2^m different bit patterns x for all $y \in K$. Every pattern $y \circ x$ requires one full scalar multiplication and one point comparison. Since the

last K contains at most $c \cdot (n-2) - \lceil n/m \rceil - 1$ candidates for y, we have total costs of

$$O\left(2^m \cdot \frac{n^2}{m} \cdot \log(2n)\right)$$

times the cost to perform a full scalar multiplication.

This implies that all 3 phases together can be performed in time

$$O\left(2^{2m} \cdot \frac{n^2}{m} \cdot \log(2n) \cdot M\right),$$

where M denotes the maximal cost of a full scalar multiplication and a full scalar multiplication with the ability to induce a Sign Change Fault. □

The result of Theorem 5.23 shows that the asymptotic costs of Algorithm 5.19 are not significantly higher than the costs of other attack algorithms presented in this chapter, e.g., Algorithm 5.6. The main difference is an additional factor of 2^m. In practice, the round set K will hardly contain more than just a few elements, which are all correct guesses. As mentioned before, we performed numerous experiments, which never witnessed a single false positive. Therefore, Montgomery's Binary Method is highly susceptible to Sign Change Faults.

5.4.4. Other Attacks

In Algorithm 5.16, the variables $P1_{(i)}$, $P2_{(i)}$, $P1_{(i+1)}$ and $P2_{(i+1)}$ are used. Obviously, if either $P1_{(i)}$ or $P2_{(i)}$ are affected by a fault, the results are the same as if $P1_{(i+1)}$ or $P2_{(i+1)}$ are affected by a fault in the following round. Therefore, we only need to investigate faults induced into $P1_{(i+1)}$ or $P2_{(i+1)}$.

In the previous section, we have presented in detail a permanent Sign Change Fault induced into $P1_{(i+1)}$. However, it is also possible to induce a transient fault into $P1_{(i+1)}$. Since $P1_{(i+1)}$ is used twice only in the case when $d_i = 0$, a permanent fault and a transient fault have the same effect for an attack if $d_i = 1$, but a different effect if $d_i = 0$.

Transient Attack Targeting $P1_{(i+1)}$

Prior to a fault being induced into $P1_{(i+1)}$, we have the two values $P1_{(i+1)} = H_{i+1}(k)$ and $P2_{(i+1)} = H_{i+1}(k) + P$ according to Corollary 5.17. We assume that the first usage of $P1_{(i+1)}$ is affected by the fault, which is used to compute the sum $P1_{(i+1)} + P2_{(i+1)}$. Let $P1_{(i+1)} \overset{\lightning}{\mapsto} \tilde{P}1_{(i+1)} = -H_{i+1}(k)$ be the faulty value after a transient Sign Change Fault was induced into $P1_{(i+1)}$. Since the effect of the fault is transient, the correct value for $P1_{(i+1)}$ is used when the variable is accessed another time. This only happens if $d_i = 0$. In Algorithm 5.16, we get the following different results for the values $P1_{(i)}$, $P2_{(i)}$ and Δ, depending on the bit value of d_i. These equations are derived in the same manner as shown in Lemma 5.18.

In the case where $d_i = 0$, we have

$$\begin{aligned}
P2_{(i)} &= \tilde{P}1_{(i+1)} + P2_{(i+1)} = -H_{i+1}(k) + H_{i+1}(k) + P = P, \\
P1_{(i)} &= 2P1_{(i+1)} = 2H_{i+1}(k) = H_i(k), \\
\text{and} \quad \Delta &= P2_{(i)} - P1_{(i)} = P - H_i(k), \\
\Rightarrow \qquad \tilde{Q} &\overset{(5.12)}{=} 2^i \cdot H_i(k) + l_{i-1}(k) \cdot (P - H_i(k))
\end{aligned}$$

$$\begin{aligned} &\overset{(5.4)}{=} Q - l_{i-1} \cdot \left(\frac{Q - L_{i-1}(k)}{2^i} \right) \\ &= -\left(\frac{l_i(k)}{2^i} - 1 \right) \cdot (Q - L_i(k)) + L_i(k) \qquad \text{(LSB version)} \\ &\overset{(5.3)}{=} Q \cdot (1 - h_i(k)) + 2^{i+1} H_{i+1}(k) \cdot h_i(k) \qquad \text{(MSB version)}, \end{aligned} \tag{5.21}$$

and in the case where $d_i = 1$, we have

$$\begin{aligned} P1_{(i)} &= \tilde{P}1_{(i+1)} + P2_{(i+1)} = -H_{i+1}(k) + H_{i+1}(k) + P = P, \\ P2_{(i)} &= 2P2_{(i+1)} = 2H_{i+1}(k) + 2P = H_i(k) + P, \\ \text{and} \quad \Delta &= P2_{(i)} - P1_{(i)} = 2H_{i+1}(k) + P = H_i(k), \\ \Rightarrow \quad \tilde{Q} &= 2^i \cdot P + l_{i-1}(k) \cdot H_i(k) \\ &= \left(\frac{l_i(k)}{2^i} - 1 \right) \cdot (Q - L_i(k)) + L_i(k) \qquad \text{(LSB version)} \\ &= Q \cdot h_i(k) - 2^{i+1} H_{i+1}(k) \cdot (1 + h_i(k)) \qquad \text{(MSB version)}. \end{aligned} \tag{5.22}$$

Not surprisingly, the LSB version for $\tilde{Q}$ in the case where $d_i = 1$ is equivalent to the formulas derived for a permanent fault, Equation (5.16). However, a transient fault in the case where $d_i = 0$ is different. For the LSB formula, the first term in (5.21) has the opposite sign. A similar effect can be observed for the MSB formula, however, we decided to state the formulas in a more compact way. The fact that we have two different equations depending on the bit value of d_i implies that the verification step must be adapted accordingly. Therefore, it becomes more complex. However, besides this conditional computation, the basic structure of the attack algorithm, Algorithm 5.19, can still be used.

In order to get a result about false positives similar to the results in Section 5.4.3, assume that we have two valid guesses for the same faulty final result $\tilde{Q}$. Here, we have three cases depending on whether both guesses assume $d_i = 0$, both assume that $d_i = 1$, or both assume different bit values. We will only show this attack for the LSB version, the MSB version works with the same arguments.

Case 1. Assume that the two bit patterns $x = (x_0, x_1, \ldots, x_{t-1}, 0)$ and $y = (y_0, y_1, \ldots, y_{u-1}, 0)$ are both valid guesses for a common $\tilde{Q}$. In this case, we use Equation (5.21), and we have

$$\left(\frac{l_u(y)}{2^u} - \frac{l_t(x)}{2^t} \right) \cdot k \equiv \frac{l_u(y)^2}{2^u} - \frac{l_t(x)^2}{2^t} \bmod \#E. \tag{5.23}$$

In this case, we get the same result as in Lemma 5.21. If $l_u(k)/2^u - l_t(x)/2^t \not\equiv 0 \bmod \#E$, k can be computed directly. Otherwise, the special form of this term allows to bound the number of pairs (x, y), which would not lead to a direct recovery in the same manner and with the same result as presented in Lemma 5.22. The result from Lemma 5.22 shows that we may have at most $c \cdot (n-2)$ valid guesses for the most significant bit value 0. Otherwise, k can be computed directly.

Case 2. Now assume that the two bit patterns $x = (x_0, x_1, \ldots, x_{t-1}, 1)$ and $y = (y_0, y_1, \ldots, y_{u-1}, 1)$ are both valid guesses for a common $\tilde{Q}$. By Equation (5.22) we have

$$\left(\frac{l_t(x)}{2^t} - \frac{l_u(y)}{2^u} \right) \cdot k \equiv 2l_u(y) - 2l_t(x) + \frac{l_t(x)^2}{2^t} - \frac{l_u(y)^2}{2^u} \bmod \#E. \tag{5.24}$$

This case yields the same results as Case 1, since the condition for a pair (x, y), which does not allow to compute k directly, is the same as above. The result from Lemma 5.22 holds in this case as well, showing that we may have at most $c \cdot (n-2)$ valid guesses for the most significant bit value 1 unless we can compute k directly.

Case 3. If we had only bit patterns which either all guess their most significant bit to be equal to 0 or to 1, we would only have either Case 1 or Case 2. Then, the results of Section 5.4.3 would hold for a transient fault as well. However, this cannot be guaranteed and the mix-up of both kinds of guesses creates the need for additional care. Let the two bit pattern $x = (x_0, x_1, \ldots, x_{t-1}, 0)$ and $y = (y_0, y_1, \ldots, y_{u-1}, 1)$ both be valid guesses for a common $\tilde{Q}$. Using Equation (5.21) for x and Equation (5.22) for y, we get

$$\left(\frac{l_t(x)}{2^t} + \frac{l_u(y)}{2^u} - 2\right) \cdot k \equiv \frac{l_u(y)^2}{2^u} + \frac{l_t(x)^2}{2^t} - 2l_u(y) \bmod \#E. \tag{5.25}$$

This allows to recover k directly unless $l_t(x)/2^t + l_u(y)/2^u \equiv 2 \bmod \#E$.

However, we do not need to care for Case 3. Since we can have at most $c \cdot (n-2)$ different patterns ending in 0 and $c \cdot (n-2)$ different patterns ending in 1, before we can compute k directly, we need to take care of at most twice as many false positives. This only doubles the amount of possible values $y \in K$, which are chosen in Line 7 of the attack algorithm, Algorithm 5.19. This additional workload vanishes in the O-notation. We also have a negligible increase in memory requirements, since now we need to store two patterns to compare with valid guesses. Therefore, the result of Theorem 5.23 holds for a transient fault induced into $P1_{(i+1)}$ as well.

Attacks Targeting $P2_{(i+1)}$.

The analysis for faults induced into the intermediate variable $P2_{(i+1)}$ is completely similar to the analysis presented in the last section. This is not a surprise, since Montgomery's Binary Method is symmetric in $P1$ and $P2$ for the two cases $d_i = 0$ and $d_i = 1$.

Permanent Faults. For a permanent Sign Change Fault induced into $P2_{(i+1)}$, we have

$$\begin{aligned}
P1_{(i+1)} &= H_{i+1}(k), \\
\tilde{P}2_{(i+1)} &= -H_{i+1}(k) - P, \\
\Rightarrow \quad \Delta &= \tilde{P}2_{(i+1)} - P1_{(i+1)} = -2H_{i+1}(k) - P \\
\Rightarrow \quad \tilde{Q} &\overset{(5.12)}{=} 2^{i+1} \cdot H_{i+1}(k) - l_i(k) \cdot (2H_{i+1}(k) + P) \\
&= \left(2^{i+1} - 2l_i(k)\right) \cdot H_{i+1}(k) - L_i(k) \\
&= \left(2^{i+1} - 2l_i(k)\right) \cdot \left(\frac{Q - L_i(k)}{2^{i+1}}\right) - L_i(k) \\
&= \left(1 - \frac{l_i(k)}{2^i}\right) \cdot (Q - L_i(k)) - L_i(k) && \text{(LSB version)} \\
&= 2^{i+1}H_{i+1}(k) - (2h_{i+1}(k) + 1) \cdot \left(Q - 2^{i+1}H_{i+1}(k)\right) && \text{(MSB version).}
\end{aligned}$$

These equations for $\tilde{Q}$ are the same equations as for a permanent fault in $P1_{(i+1)}$ as shown in Lemma 5.18, with just the opposite sign. This opposite sign does not matter in terms of

the analysis of false positives, hence, the result of Theorem 5.23 holds for a permanent fault in $P2_{(i+1)}$ with a new verification step of

15(a) If $\left(-\mathsf{T}_x = \tilde{\mathsf{Q}}\right)$ then

as well.

Transient Faults. Prior to a fault being induced into $P2_{(i+1)}$, we have the two values $P1_{(i+1)} = H_{i+1}(k)$ and $P2_{(i+1)} = H_{i+1}(k) + P$ according to Corollary 5.17. Let $P2_{(i+1)} \nmapsto \tilde{P}2_{(i+1)} = -H_{i+1}(k) - P$ be the faulty value after a transient Sign Change Fault was induced into $P2_{(i+1)}$. In Algorithm 5.16, we get the following different results for the values $P1_{(i)}$, $P2_{(i)}$ and Δ, depending on the bit value of d_i, in the same manner as shown for a transient fault in $P1_{(i+1)}$ above.

In the case where $d_i = 0$, we have

$$\begin{aligned}
& P2_{(i)} = P1_{(i+1)} + \tilde{P}2_{(i+1)} = H_{i+1}(k) - H_{i+1}(k) - P = -P, \\
& P1_{(i)} = 2P1_{(i+1)} = 2H_{i+1}(k) = H_i(k), \\
\text{and} \quad & \Delta = P2_{(i)} - P1_{(i)} = -P - H_i(k), \\
\Rightarrow \quad & \tilde{Q} = 2^i \cdot H_i(k) + l_{i-1}(k) \cdot (-P - H_i(k)) \\
& \quad = \left(1 - \frac{l_i(k)}{2^i}\right) \cdot (Q - L_i(k)) - L_i(k) \qquad \text{(LSB version)} \qquad (5.26) \\
& \quad = -Q \cdot (h_i(k) + 1) + 2^i H_i(k) \cdot (2 + h_i(k)) \qquad \text{(MSB version)},
\end{aligned}$$

and in the case where $d_i = 1$, we have

$$\begin{aligned}
& P1_{(i)} = P1_{(i+1)} + \tilde{P}2_{(i+1)} = H_{i+1}(k) - H_{i+1}(k) - P = -P, \\
& P2_{(i)} = 2P2_{(i+1)} = 2H_{i+1}(k) + 2P = H_i(k) + P, \\
\text{and} \quad & \Delta = P2_{(i)} - P1_{(i)} = 2H_{i+1}(k) + 3P = H_i(k) + 2P, \\
\Rightarrow \quad & \tilde{Q} = -2^i \cdot P + l_{i-1}(k) \cdot (H_i(k) + 2P) \\
& \quad = \left(\frac{l_i(k)}{2^i} - 1\right) \cdot (Q - L_i(k)) + 3L_i(k) - 2^{i+2}P \qquad \text{(LSB version)} \qquad (5.27) \\
& \quad = Q \cdot (h_i(k) + 2) - 2^i H_i(k) \cdot (2 + h_i(k)) - 2^i P \qquad \text{(MSB version)}.
\end{aligned}$$

Since the analysis of the number of false positives, which may occur unless k can be computed directly, only depends on the factor of Q in Equations (5.26) and (5.27), the analysis of a transient fault induced into $P2_{(i+1)}$ yields the same result as the analysis of a transient fault induced into $P1_{(i+1)}$. The factors of Q are the same for both variables. Therefore, the results from the analysis of a transient fault induced into $P1_{(i+1)}$ hold for this attack as well.

5.4.5. Recovering The Bits Starting From The MSB

In Lemma 5.18, we have presented formulas describing $\tilde{Q}$ both in terms of Q and the i least significant bits as well as in terms of Q and the $(n - i - 2)$ most significant bits. Hence, we can take the results from the attack presented in Section 5.4.2 as a basis for the recovery of k starting with the most significant bits. The attack algorithm is quite similar. The modified parts have been highlighted. To have as little changes as possible compared to Algorithm 5.19, we will denote valid guesses for the $u + 1$ most significant bits by $y = (y_0, y_1, \ldots, y_u)$. Here, we

use indices in reverse order of the indices of k, such that $y_j = k_{n-1-j}$ for all $0 \leq j \leq n-1$, e.g., $y_0 = k_{n-1}$. We start with $K = \{1\}$, since we know that $k_{n-1} = y_0 = 1$.

Algorithm 5.24: Sign Change Attack on $\mathsf{P1}_{(i+1)}$ in Algorithm 5.16, MSB version

Input: Access to Algorithm 5.16, n as the binary length of the secret key k, Q = kP the correct result, m a parameter for acceptable amount of offline work.
Output: k with probability at least 1/2.

\# **Phase 1:** Collect Faulty Outputs
1 Set c := (n/m) · log(2n)
2 Create c faulty outputs of Algorithm 5.16 by inducing SCFs in $\mathsf{P1}_{(i+1)}$ for random values of i.
3 Collect the set $\mathcal{S} = \{\tilde{Q} |\ \tilde{Q} \neq Q$ is a faulty output of Algorithm 5.16 on input P$\}$.
\# **Phase 2:** Inductive Retrieval of Secret Key Bits
4 Set s := 0 representing the current candidate bit length between $s \cdot m + 1$ and $(s+1) \cdot m$
5 Set $K := \{1\}$ to contain candidates for verified/"known" MSB-parts of k
6 While $(s \cdot m < n - 1)$ do
7 for all patterns y ∈ K do
8 Set u := l(y) # the algorithm ensures that $(s-1) \cdot m < u \leq s \cdot m$ if $s > 0$
9 Set K′ = ∅ # to contain candidates of length between $s \cdot m + 1$ and $(s+1) \cdot m$
10 For all r with $s \cdot m < u + r \leq (s+1) \cdot m$ do
11 For all test bit patterns $x = (x_{u+1}, x_{u+2}, \ldots x_{u+r})$ of length $r < 2m$ do
\# Compute the test candidate T_x using the pattern y ∘ x and Equation (5.17)
12 $$H := \sum_{j=0}^{u} y_j 2^{u+r-j} + \sum_{j=u+1}^{u+r} x_j 2^{u+r-j}$$
13 $$T_x := (1+2H) \cdot \left(Q - 2^{u+r} \cdot H \cdot P\right) - 2^{u+r} \cdot H \cdot P$$
\# Verification Step: Verify the test candidate using Equation (5.17)
14 For all $\tilde{Q} \in \mathcal{S}$ do
15 If $\left(T_x = \tilde{Q}\right)$ then
16 Add $y \circ x = (y_0, y_1, \ldots, y_u, x_{u+1}, \ldots, x_{u+r})$ to the set K′
17 Set K := K′ and s := s + 1 and continue at Line 6
\# **Phase 3:** Compute k from all candidates of length $u > n - 1 - m$
18 For all y ∈ K do
19 Set u := l(y)
20 Set r := n − 1 − u
21 For all test bit patterns $x = (x_{u+1}, x_{u+2}, \ldots x_{u+r})$ of length r < m do
22 If $\left(\sum_{j=0}^{u} y_j 2^{n-1-j} + \sum_{j=u+1}^{n-1} x_j 2^{n-1-j}\right) \cdot P = Q$ then
23 **Output** $(x_{n-1}, x_{n-2}, \ldots, x_{u+1}, y_u, y_{u-1}, \ldots, y_0)$
24 **Output** FAILURE

Algorithm 5.24 has been abbreviated in some minor details, similar to its archetype, Algorithm 5.19. Again, we do not explicitly incorporate the checking of valid guesses, which might reveal k directly. The possibility of computing k by false positives is the same which has been observed

for valid guesses in the last section and it will be described in detail below. We also do not describe that we only guess patterns $y \circ x$ with a maximal length of $n-1$ in Phase 2, i.e., in the loop of Lines 6–17. This has been left out to improve readability. However, it is useful in the proof of Lemma 5.25 below and adds only additional costs to search for the least significant bit k_0 by exhaustive search.

To get a result similar to Theorem 5.23, we need to be able to bound the number of false positives which do not allow to recover k directly. This can be done with the same considerations as for the LSB recovery, and it will be shown with the following Lemma, the twin of Lemma 5.21.

Lemma 5.25 (Properties of Valid Guesses).
Let $x = (x_{n-1}, x_{n-2}, \ldots, x_t)$ and $y = (y_{n-1}, y_{n-2}, \ldots, y_u)$ be two valid guesses for the $n-t$ and $n-u$ most significant bits of k. Let $1 \leq u \leq t \leq n-1$ and let $\tilde{Q} \in S$ be a faulty final result collected in Line 2 of Algorithm 5.24. Let

$$Ver(\tilde{Q}, x) := \begin{cases} true & if \quad \tilde{Q} = (1 + 2h_t(x)) \cdot \left(Q - 2^t h_t(x) P\right) - 2^t h_t(x) P \\ false & otherwise, \end{cases}$$

with $h_t(x) = \sum_{j=t}^{n-1} x_j 2^{j-t}$ as defined in Definition 5.4. If $Ver(\tilde{Q}, x) = true = Ver(\tilde{Q}, y)$, then we have one of the following cases:

1. *The two patterns x and y are equal, i.e., $x = y$ and $t = u$, or*
2. *we have $u < t$ and*
$$h_t(x) - h_u(y) \equiv 0 \bmod \#E,$$
i.e., the pattern y is the same as the pattern x shifted $t-u$ bits to the right, and reduced modulo $\#E$, or
3. *we have $u \leq t$ and $h_u(y) \not\equiv h_t(x) \bmod \#E$ and k can be computed directly as*
$$k \equiv \frac{(1 + h_t(x)) \cdot 2^t h_t(x) - (1 + h_u(y)) \cdot 2^u h_u(y)}{(h_t(x) - h_u(y))} \bmod \#E.$$

Proof:

The Lemma follows with the same arguments as in Lemma 5.21. If $\mathrm{Ver}(\tilde{Q}, x) = \mathrm{true} = \mathrm{Ver}(\tilde{Q}, y)$, we have

$$\begin{aligned} (1 + 2h_t(x)) \cdot (Q - 2^t h_t(x) P) - 2^t h_t(x) P &= (1 + 2h_u(y)) \cdot (Q - 2^u h_u(y) P) - 2^u h_u(y) P \\ \Rightarrow (1 + 2h_t(x) - 1 - 2h_u(y)) \cdot k &\equiv (1 + 2h_t(x)) \cdot 2^t h_t(x) + 2^t h_t(x) \\ &\quad - (1 + 2h_u(y)) \cdot 2^u h_u(y) - 2^u h_u(y) \\ \Rightarrow 2\,(h_t(x) - h_u(y)) \cdot k &\equiv (2 + 2h_t(x)) \cdot 2^t h_t(x) \\ &\quad - (2 + 2h_u(y)) \cdot 2^u h_u(y) \\ \Rightarrow (h_t(x) - h_u(y)) \cdot k &\equiv (1 + h_t(x)) \cdot 2^t h_t(x) - (1 + h_u(y)) \cdot 2^u h_u(y). \end{aligned} \tag{5.29}$$

If $h_t(x) \not\equiv h_u(y) \bmod \#E$, the multiplicative inverse of $(h_t(x) - h_u(y))$ exists modulo $\#E$ and k can be computed directly. If $t = u$ and $x \neq y$, we always have $h_t(x) \not\equiv h_u(y)$.

For the other cases, assume w.l.o.g. that $u < t$. In this case, we have $h_t(x) \equiv h_u(y)$ if the bits $(x_{n-1}, x_{n-2}, \ldots, x_t)$ and the bits $(y_{n-1-t+u}, y_{n-2-t+u}, \ldots, y_u)$ are equal and all other bits $(y_{n-1}, y_{n-2}, \ldots, y_{n-t+u})$ are zero. Hence, y must be a shift of x to the right and reduced modulo $\#E$. Similar to the proof of Lemma 5.21, we may assume that both $h_t(x)$ and $h_u(y)$ are smaller than k, i.e., $0 < t, u$, to ensure that $h_t(x) \equiv h_u(y)$ iff $h_t(x) = h_u(y)$. This requires to recover the least significant bit k_0 by exhaustive search, which is a negligible amount of additional work. □

The result of Lemma 5.25 shows that the maximal number of false positives, which may occur unless k can be recovered directly, can be bounded by the same bound as stated in Lemma 5.22. Hence, Theorem 5.23 holds for the MSB attack as well.

However, the MSB attack even yields better results. Since we know that $k_{n-1} = 1 = x_{n-1} = y_{n-1}$ for the two patterns x and y defined in Lemma 5.25, we know that y cannot be a shift of x to the right, since this requires y_{n-1} to be equal to zero. Hence, in this case, there are no false positives which do not allow to recover the bits of k directly via Case 3 of Lemma 5.25. Therefore, we get the following improved version of the main result.

Theorem 5.26 (Success of the Proposed Sign Change Attack).
Algorithm 5.19 succeeds to recover the secret scalar multiple k of bit length n in time $O(2^{2m} \cdot c \cdot M)$ with probability at least $1/2$. Here, $c = (n/m) \cdot \log(2n)$ and M is the maximal cost of a full scalar multiplication or a scalar multiplication including the induction of a Sign Change Fault.

Proof:

The proof is completely analogous to the proof of Theorem 5.23 with the only modification that for any specific $\tilde{Q}$, there can only be at most 2 valid guesses, since any two valid guesses allow to recover k directly using Case 3 of Lemma 5.25. Case 2 of Lemma 5.25 cannot occur. □

5.5. Concluding Remarks

All standard variants of repeated doubling have been shown to be vulnerable to Sign Change Attacks. The scheme of the attack follows the basic scheme from [BDL01], as presented in Chapter 2. However, we have always assumed in this chapter that all faulty results $\tilde{Q}$ have been created by attacks on algorithms with the same fixed input point P. This can be further relaxed in the same manner as in the attack described in Chapter 2. If the input point P is known to the adversary, it is possible to use a new input point for every faulty result $\tilde{Q}$. In this case, we need to collect pairs $(P_i, \tilde{Q}_i)$ in the first phase of all attack algorithms. Given such a pair, the test value T_x must be computed with the individual input point P_i to be compared to $\tilde{Q}_i$. This will prevent some possible speed-ups in the attacks, but it does not have any influence on the results.

Since we have shown that Sign Change Faults are a realistic fault model (in Section 4.5), the attacks described here are a real new threat for elliptic curve cryptosystems. These results have been published as a joint work with Johannes Blömer and Jean-Pierre Seifert in [BOS04].

Any new attack triggers research into appropriate countermeasures. The next chapter is devoted to the presentation of a new countermeasure for elliptic curve cryptosystem, which protects specifically against Sign Change Faults.

6. Securing Elliptic Curve Cryptosystems

The newly developed Sign Change Fault Model has been shown to be successful in recovering the secret scalar factor of an elliptic curve scalar multiplication. Therefore, countermeasures are needed. In this chapter, we will first briefly review the effectiveness of existing countermeasures developed against other side-channel attacks on elliptic curve cryptosystems in Section 6.1. Then, we will introduce a new countermeasure in Section 6.2, which has been designed to protect elliptic curve scalar multiplication against Sign Change Faults. We will analyze the effectiveness of this new countermeasure in Section 6.2.2.

6.1. Previously Proposed Countermeasures

We have given an overview over existing fault attacks on elliptic curve scalar multiplication in Section 4.2. Those attacks change coordinates or curve parameters randomly. As a result, almost certainly the computed value is not a valid point on the original elliptic curve. Hence, the standard countermeasure proposed by [BMM00] and [CJ03] validates a final result before it is returned by a device, i.e., it simply checks whether a computed value is a valid point on the original elliptic curve. Clearly, this countermeasure is defeated by Sign Change Faults, since these faults generate points which are bound to be on the original curve. Therefore, the standard countermeasure cannot be used to protect against Sign Change Faults. Moreover, if a device implements the standard countermeasure, this even helps the attacker by eliminating almost all faulty outputs, which result from an induced fault different from a sign change. This allows the Sign Change Fault Model 5.1 to be less precise in the control on location and in the number of affected bits.

A quite natural countermeasure against Sign Change Faults, which change the sign of the y-coordinate of an intermediate point, is to use Montgomery's binary method, where the y-coordinate is not needed (cf. [Mon87], [FGKS02]). However, several patents prevent its widespread application and endorse the use of other variants of repeated doubling.

Moreover, some variants of the ElGamal cryptosystem embed messages in both coordinates (cf. [GG99, § 20.6]) and may wish to use a scalar multiplication method, where both coordinates are computed simultaneously. Additionally, there are verification schemes, which need the y-coordinate according to [IT02].

As a consequence, variants of the well-known repeated doubling algorithm, mostly based on the non-adjacent form of the secret scalar factor, are widely employed in modern elliptic curve based systems.

One can also use randomization schemes to counteract a fault attack with Sign Change Faults. However, smartcard certification authorities often require that algorithms are secure against fault attacks even without randomization. Moreover, randomization schemes that only randomize the base point are not guaranteed to thwart our attacks, e.g., Coron's third countermeasure in [Cor99, §5.3] or the proposed elliptic curve isomorphism in [JT01b, §4.1] do not defeat Sign Change Attacks.

As an alternative countermeasure against Sign Change Attacks, we propose a modified scalar multiplication algorithm in the next section.

6.2. A New Countermeasure Against Sign Change Attacks

As a new countermeasure against Sign Change Attacks, we propose a modified repeated doubling algorithm, which we present as Algorithm 6.1. It adds little overhead at the benefit of checking the correctness of the final result. Moreover, it can be based on any scalar multiplication algorithm which does not need field divisions. Obviously, the latter prevents the use of affine coordinates. Hence, we will present our countermeasure in the context of projective coordinates in the remainder of this section. We will analyze it using the NAF-based version presented as Algorithm 5.2. The countermeasure has been motivated by Shamir's countermeasure against attacks on CRT-RSA exponentiations [Sha99], cf. Section 3.2.

We first explain the basic idea of our proposed countermeasure. For the modified algorithm, we assume that the curve $E = E_p$ is defined over a prime field $\mathbb{F}_p$, i.e., we have $E_p := E(\mathbb{F}_p)$. Furthermore, we choose a small prime t of about $60 - 80$ bits to form a "small" curve $E_t := E(\mathbb{F}_t)$. We now want to compute the scalar multiple kP in a way such that the result can easily be checked on E_t but also yields the correct result on E_p. To do so, we define an elliptic curve E_{pt} over the ring $\mathbb{Z}_{pt}$. This curve E_{pt} is defined with parameters A_{pt} and B_{pt} such that

$$\begin{aligned} & A_{pt} \equiv A_p \bmod p, && A_{pt} \equiv A_t \bmod t \\ \text{and} \quad & B_{pt} \equiv B_p \bmod p, && B_{pt} \equiv B_t \bmod t. \end{aligned}$$

Here, A_p and A_t denote the A-parameters and B_p and B_t denote the B-parameters in the Weierstrass Equation (4.5) of E_p and E_t respectively. Both A_{pt} and B_{pt} can be easily computed using the Chinese Remainder Theorem, although B_{pt} is not needed in the addition formula. As the base point $P_p := P$ is not guaranteed to exist on E_t, we also choose a base point P_t on E_t and use the combined point P_{pt} as the base point for the scalar multiplication in E_{pt}. Here, P_{pt} is computed using the Chinese Remainder Theorem in the same manner as A_{pt} and B_{pt}, i.e.,

$$\begin{aligned} & P_{pt}[u] \equiv P_p[u] \bmod p \\ \text{and} \quad & P_{pt}[u] \equiv P_t[u] \bmod t \end{aligned} \qquad \text{for all } u \in \{x, y, z\}.$$

We will refer to E_{pt} as the "combined curve" of E_p and E_t. Computing $Q = kP_{pt}$ on E_{pt} allows to verify the result on the small curve E_t.

Algorithm 6.1: Sign Change Attack Secure Scalar Multiplication

Input: A point P on E_p, and a secret key $1 < k < \text{ord}(P)$
Output: kP on E_p
\# **offline initialization (i.e., at production time)**
1 Choose a prime t and an elliptic curve E_t
2 Determine the combined curve E_{pt}
\# **main part**
3 Set Q := kP_{pt} on E_{pt} (e.g., using Algorithm 5.2)
4 Set R := kP_t on E_t (e.g., using Algorithm 5.2)
5 If R $\neq$ Q mod t then **output** FAILURE.
6 else **output** Q on E_p

Scalar Multiplication is used twice, once in Line 3 and once in Line 4. For the algorithm used, we assume that it features a check of the final result that returns $\mathcal{O}$ if the result is not a valid point on the curve (e.g., Line 5 of Algorithm 5.2). In the case where E_{pt} is used, we assume for simplicity that this check is performed both modulo p and modulo t, i.e., both on E_p and on E_t.

6.2.1. On the Choice of E_p and E_t.

For the security of our countermeasure against Sign Change Attacks, we assume that both E_p and E_t have prime order. Both curves are chosen independently, which allows to use recommended curves for E_p, see Table 4.1 for details on recommendations. We assume that E_p and P are chosen first. Afterwards, we choose the small curve E_t with prime order. Finding such a curve E_t is feasible as we will show in the following. More about finding curves with a desired order can be found in [BSS99, §VI.5] or [Kob91].

On Choosing E_t with Prime Order.

We suggest to determine E_t by choosing a triplet $(A_t, x_t, y_t) \in \mathbb{Z}_t^3$ uniformly at random until the curve E_t defined by that triplet has prime order. The order of such a curve E_t can be determined in polynomial time using Schoof's Algorithm [Sch95]. This curve E_t with $B_t := y_t^2 - x_t^3 - A_t x_t \bmod t$ and the point $P_t = (x_t : y_t : 1)$ is used to compute the combined curve E_{pt} and to check the result in Line 5 of Algorithm 6.1. The security analysis will show that the security depends on the order of P_t on E_t. This does not require E_t to be secret. Moreover, it also does not require $\#E_t$ to be prime. It is sufficient to choose a curve E_t and a point P_t such that the order of P_t on E_t is large. We will specify a minimal size for the order of P_t on E_t in Section 6.2.2.

Based on the following widely accepted conjecture, choosing $\#E_t$ as a prime is practical. Our conjecture is implied by the well-known Cramer's conjecture, stating that $\pi(x+\log^2(x)) - \pi(x) > 0$ for x sufficiently large [GK86]. Here, $\pi(x)$ is the prime counting function. Our conjecture has been introduced in a similar setting by Goldwasser and Kilian [GK86].

Conjecture 6.2 (Number of Primes in the Hasse Interval).
There exist constants $c_1, c_2 > 0$ such that

$$\pi(t + 2\sqrt{t}) - \pi(t - 2\sqrt{t}) \geq \frac{c_2\sqrt{t}}{\log^{c_1}(t)}.$$

One could avoid using Cramer's Conjecture by using a rather deep theorem of Heath-Brown from [HB78] in conjunction with choosing the prime t at random. However, this theorem uses very large constants, which allows to use it only for very large curve orders, clearly larger than the curve orders currently used. The following theorem states that we can find a small curve E_t with prime order efficiently if Conjecture 6.2 holds.

Theorem 6.3 (Choosing E_t with Prime Order).
Let $(A_t, x_t, y_t) \in \mathbb{Z}_t^3$ be chosen uniformly at random. (A_t, x_t, y_t) defines the curve E_t as

$$E_t : y_t^2 z_t \equiv x_t^2 + A_t x_t z_t^2 + B_t z_t^3 \bmod t,$$

where $B_t := y_t^2 - x_t^3 - A_t x_t \bmod t$. If Conjecture 6.2 is true, then there exists a constant $c > 0$ such that the probability that E_t defined by $(A_t, x_t, y_t) \in \mathbb{Z}_t^3$ has prime order is at least

$$\frac{c \cdot c_2}{\log^{1+c_1}(t)},$$

where c_1, c_2 are as in Conjecture 6.2.

Proof:

Given a triplet $(A_t, x_t, y_t) \in \mathbb{F}_t^3$, we want to determine the probability that the curve E_t defined by this triplet has prime order. As the parameter A_t is given and x_t and y_t define a point $P = (x_t : y_t : 1)$ on the curve, E_t is uniquely defined. As $z_t = 1$, the missing curve parameter B_t can easily be computed as $B_t = y_t^2 - x_t^3 - A_t x_t \bmod t$.

E_t is a valid elliptic curve only if the discriminant is non-zero, i.e., if $\gcd(4A_t^3 + 27B_t^2, t) = 1$. The possibility that a random triplet yields a zero discriminant is negligible: There are $t^2 - t$ different curves over $\mathbb{F}_t$, each of which contains at least $t - 2\sqrt{t}$ many points. As the curve is uniquely defined if A_t and any of its points P_t are fixed, there are at least $(t^2 - t) \cdot (t - 2\sqrt{t})$ many valid triplets. As there are t^3 triplets in $\mathbb{F}_t^3$ altogether, we have

$$\begin{aligned}\Pr[(A_t, x_t, y_t) \text{ yields a valid curve } E_t] &\geq \frac{(t^2 - t) \cdot (t - 2\sqrt{t})}{t^3} \\ &= 1 - \left(\frac{2}{\sqrt{t}} + \frac{1}{t} - \frac{1}{t\sqrt{t}}\right) \geq 1 - \frac{3}{\sqrt{t}}.\end{aligned}$$

As this probability is negligibly far from 1, we will assume that all t^3 triplets define valid curves for simplicity. By Hasse's Theorem, we know that all possible group orders lie in the interval $[t+1-2\sqrt{t}, t+1+2\sqrt{t}]$, cf. [Sil00]. The theorem of Deuring (cf. [Len87] or [Deu41]) states that there exists a constant $c' > 0$ such that there are at least $c' \cdot (t\sqrt{t})/\log(t)$ many elliptic curves for every given group order. This shows that the group orders are almost evenly distributed over all possible curves up to a factor of $1/\log(t)$. Among the $4\sqrt{t} + 1$ possible group orders, there are $\Delta\pi := \pi(t+1+2\sqrt{t}) - \pi(t+1-2\sqrt{t})$ prime group orders, where $\pi(x)$ is the prime counting function. Following Conjecture 6.2, we have $c_1, c_2 > 0$ such that $\Delta\pi \geq \frac{c_2\sqrt{t}}{\log^{c_1}(t)}$. Each of these curves is uniquely defined by its A parameter and any of its points, i.e., there are at least $t - 2\sqrt{t}$ many triplets that define any of these curves. This holds because each of these curves has at least $t + 1 - 2\sqrt{t}$ many points and $\mathcal{O}$ cannot be chosen in (A_t, x_t, y_t). Therefore, the probability that a random triplet $(A_t, x_t, y_t) \in \mathbb{F}_t^3$ yields an elliptic curve with prime order is

$$\begin{aligned}\Pr[\#E_t \text{ is prime}] &\geq \frac{\Delta\pi \cdot c' \cdot t\sqrt{t} \cdot (t - 2\sqrt{t})}{\log(t) \cdot t^3} \geq \frac{c'}{\log(t)} \cdot \frac{\sqrt{t} - 2}{t} \cdot \frac{c_2\sqrt{t}}{\log^{c_1}(t)} \\ &= \frac{c' \cdot c_2}{\log^{1+c_1}(t)} \cdot \left(1 - \frac{2}{\sqrt{t}}\right) = \frac{c \cdot c_2}{\log^{1+c_1}(t)},\end{aligned}$$

where for $t > 16$ we can choose $c = c'/2$. □

6.2.2. Analysis of the Countermeasure.

We first show that Algorithm 6.1 computes the correct result if no error occurs. Algorithm 6.1 uses E_t to check the result of the scalar multiplication from Line 3. If $Q = kP$ on E_{pt}, then $Q = kP$ on E_t as well. This is evident from modular arithmetic. Any out of the three exceptional cases in the addition formula ($P_1, P_2 = \mathcal{O}$ or $P_1 = -P_2$) can only occur if at least one of the intermediate results or the final result is equal to $\mathcal{O}$. However, $Q_i = \mathcal{O}$ on E_{pt} implies that $Q_i = \mathcal{O}$ on both E_p and E_t. As $k < \#E_p$, this may never happen. Hence, if no error occurred, Q is a valid point on both E_p and E_t. Given this fact, it is straightforward to see that $R = Q$ on E_t.

It remains to show that the proposed algorithm is secure against known Fault Attacks. For our analysis, we assume that Algorithm 5.2 has been chosen as the scalar multiplication algorithm, although the result holds for other scalar multiplication algorithms as well. To counteract fault attacks with random faults induced into any of the parameters used in Line 3 of Algorithm 6.1, we implement the countermeasure proposed by [BMM00] and [CJ03]. It requires checking whether the result of a scalar multiplication is a valid point on the original curve before returning a value. This check has been included as an integral part of Algorithm 5.2. Therefore, we concentrate on security against Sign Change Attacks on Line 3 of Algorithm 6.1 only. Our analysis proves security against Sign Change Faults only, it does not provide a general security proof or security reduction. As it has been discussed in the introduction, a mathematical framework which allows general security claims has yet to be established.

We use the Sign Change Fault Model as defined in Definition 5.1, i.e., a Sign Change Fault can be induced in any intermediate variable used by the scalar multiplication $Q = kP$ on E_{pt}. Sign Change Faults can only be induced in points of elliptic curves, the scalar k cannot be attacked. Furthermore, we assume that only a single Sign Change Fault can be induced during each computation of kP. We do not consider multiple attacks, as only correlated attacks in the same run of Algorithm 6.1 would yield an advantage for an adversary. However, such attacks are not a realistic scenario as explained in Chapter 1. The adversary can target a specific variable, e.g., Q'_i, but he cannot target a specific iteration i. As we are interested to show that a faulty value is returned with negligible probability, we only need to investigate Sign Change Attacks on the computation in Line 3 of Algorithm 6.1. Attacks in Line 4 cannot yield a faulty output as Q is not changed by Line 4. We first investigate the basic requirement for an error to be undetected by the countermeasure in Line 5.

Lemma 6.4 (Undetectable Sign Change Faults).
Let $Q = kP_{pt}$ be the correct result of the scalar multiplication in Line 3 of Algorithm 6.1 and let $\tilde{Q} = Q + \kappa_i \cdot P \neq Q$ be a faulty result from an attack on Line 3. Let $r_t := \#E_t$ be the group order of E_t, assumed to be prime. The faulty result $\tilde{Q}$ passes by the detection mechanism in Line 5, iff $r_t \mid \kappa_i$.

Proof:

Let R and Q denote the variables used in Algorithm 6.1. If $r_t \mid \kappa_i$, we have $\kappa_i P = \mathcal{O}$ on E_t. Therefore, the test $\mathsf{R} = \mathsf{Q}$ in Line 5 of Algorithm 6.1 yields $kP_t = Q + \mathcal{O}$ on E_t. As the correct result Q satisfies $Q = kP_t$ on E_t, this would not trigger a FAILURE output and the faulty value $\tilde{Q}$ would be returned. As $\tilde{Q} \neq Q$ on E_{pt} is assumed, we also have $\tilde{Q} \neq Q$ on E_p. This case results in a faulty output.

If $r_t \nmid \kappa_i$, we must show that $kP_t \neq \tilde{Q}$ on E_t. We know that for the correct value Q, it holds that $\mathsf{R} = Q$ on E_t. If $r_t \nmid \kappa_i$, we have $\kappa_i P \neq \mathcal{O}$ on E_t because r_t is the prime group order. Therefore, the order of P is r_t as well. Consequently, we have $\mathsf{R} \neq \mathsf{Q} = \tilde{Q}$ on E_t and the security alert in Line 5 is triggered. □

Lemma 6.5 (Number of Undetectable Sign Change Faults).
Let r_t be the group order of E_t, assumed to be prime. Let m be the blocksize used in Algorithm 5.6. Then a Sign Change Attack on Algorithm 6.1 needs a blocksize $m \geq \lfloor \log(r_t) \rfloor$ to be successful. Moreover, at most $(n-1)/\lfloor \log(r_t) \rfloor$ many undetectable faulty outputs exist. Here, it is $n = l(k)$.

Proof:

Assume that a Sign Change Fault was induced into Q_i' for some i, resulting in a faulty output $\tilde{Q}_1$. By Equation (5.6), we have

$$\tilde{Q}_1 = Q - 2H_{i+1}(k) = Q + \kappa_i P \qquad \text{where} \qquad \kappa_i := -2 \sum_{j=i+1}^{n-1} k_j 2^{j-i-1}.$$

We further assume that $r_t \mid \kappa_i$, i.e., $\tilde{Q}_1$ has not been detected as a faulty value according to Lemma 6.4. We now consider another faulty output $\tilde{Q}_2 \neq \tilde{Q}_1$ collected by Algorithm 5.6. Let $u \neq i$ denote the fault position, i.e., $\tilde{Q}_2 = Q + \kappa_u P$.

We claim that for all u with $|u-i| \leq \lfloor \log(r_t) \rfloor$, $u - i \neq 0$, it holds that $r_t \not| \kappa_u$. We consider the two cases $u < i$ and $u > i$. For $u < i$, we have

$$\begin{aligned} \kappa_u &= -2 \sum_{j=u+1}^{n-1} k_j 2^{j-u-1} = -2 \sum_{j=u+1}^{i} k_j 2^{j-u-1} - 2 \sum_{j=i+1}^{n-1} k_j 2^{j-i-1} \\ &= \kappa_i - 2 \cdot \sigma_u, \quad \text{where } \sigma_u := \sum_{j=u+1}^{i} k_j 2^{j-u-1}, \end{aligned} \tag{6.3}$$

and for $u > i$, we have

$$\kappa_u = -2 \sum_{j=u+1}^{n-1} k_j 2^{j-u-1} = \kappa_i + 2 \cdot \rho_u, \text{ where } \rho_u := \sum_{j=i+1}^{u} k_j 2^{j-i-1}.$$

The value $\tilde{Q}_2$ is returned only if it is an undetectable faulty result, which bypassed Line 5 of Algorithm 6.1. According to Lemma 6.4, this requires that $r_t \mid \kappa_u$. As we assume that $r_t \mid \kappa_i$, we need to analyze the case that $r_t \,|\sigma_u$ and $r_t \mid \rho_u$ respectively. We first investigate σ_u. Here, we have two cases: Either $\sigma_u = 0$ or $\sigma_u > 0$ over the integers. If $\sigma_u = 0$ over the integers, we have $\kappa_u = \kappa_i$ and $\tilde{Q}_1 = \tilde{Q}_2$. As this contradicts our assumption that $\tilde{Q}_1 \neq \tilde{Q}_2$, we may assume that σ_u is not equal to 0 over the integers. If the sum in Equation (6.3) is not equal to 0 over the integers, its absolute value must be at least as large as r_t in order to be a multiple of r_t.

Exactly the same consideration holds for ρ_u. Both $|\sigma_u|$ and $|\rho_u|$ are smaller than $2^{|u-i|}$. Therefore, we must have $|u - i| > \lfloor \log(r_t) \rfloor$ in order to have a chance that the sums $|\sigma_u|$ and $|\rho_u|$ are larger than or equal to r_t. Otherwise, we cannot have $r_t \mid \kappa_i$ and $r_t \mid \kappa_u$.

Algorithm 5.6 recovers bits in blocks of at most m bits. If it has found a valid test pattern, it starts at the position immediately following that test pattern and tries to recover the next block of length m starting at this position. If $m < \lfloor \log(r_t) \rfloor$, the arguments above shows that in this block there cannot be a faulty output $\tilde{Q}$ in the set S of collected faulty outputs that satisfies the verification step. Therefore, Algorithm 5.6 needs a minimal blocksize of $m = \lfloor \log(r_t) \rfloor$ in order to be able to reconstruct another faulty output $\tilde{Q}$.

As the fault positions of two undetected faulty outputs $\tilde{Q}_1$ and $\tilde{Q}_2$ are at least $\lfloor \log(r_t) \rfloor$ bits away from each other, we have a maximum of $(n-1)/\lfloor \log(r_t) \rfloor$ many different faulty outputs in the set collected by Algorithm 5.6. □

Lemma 6.5 shows that the proposed algorithm secures the scalar multiplication algorithm against the new Sign Change Faults if the group order of $\#E_t$ is large enough. A group order of $\#E_t > 2^{80}$ guarantees that the required block size of $m > 80$ exceeds the acceptable amount of offline work significantly. For many practical applications, $\#E_t > 2^{60}$ should already be enough.

The computational overhead is acceptable. Line 3 requires computations with 30 – 40 % larger moduli (for $l(p) = 192$ and $l(t) = 60 - 80$), Line 4 requires a scalar multiplication on a considerably smaller curve with the scalar factor $k \bmod \#E_t$, which is considerably smaller than k.

As the computations in Line 3 prohibit the use of inversions, projective coordinates must be used. We have stated our results for the basic version of projective coordinates but other weighted projective representations such as Jacobian or Hessian representations will do just as well.

6.2.3. Infective Computations

It has been explained in Chapter 3 that it is desirable to avoid single points of failure. As a possible approach to achieve this feature of an algorithm, we have reviewed the concept of *infective computations* introduced by [YKLM03] in Section 3.2.1. The explicit check proposed in Line 5 uses the zero flag as a single point of failure. However, there is an obvious way to easily modify the algorithm to use infective computations. Lines 4, 5 and 6 could be replaced by

4 Set $R := (1/k \bmod r_t) \cdot Q - P_t$ on E_t (using Algorithm 5.2)
5 Set $c := R[x] + R[y]$
6 Set $S := cQ$ on E_p (using Algorithm 5.2)
7 **Output** S

Note that if no error occurred, we have $R = \mathcal{O} = (0 : 1 : 0)$ in projective coordinates. Hence, $c = 1$ and the correct result is returned. This yields the following Algorithm, which uses infective computations instead of an explicit check.

Algorithm 6.6: Sign Change Attack Secure Scalar Multiplication with Infective Computations

Input: A point P on E_p, and a secret key $1 < k < \mathrm{ord}(P)$
Output: kP on E_p
\# **offline initialization (i.e., at production time)**
1 Choose a prime t and an elliptic curve E_t
2 Set $r_t := \mathrm{ord}(P_t)$ on E_t
3 Determine the combined curve E_{pt}
\# **main part**
4 Set $Q := kP_{pt}$ on E_{pt} (e.g., using Algorithm 5.2)
5 Set $R := (1/k \bmod r_t) \cdot Q - P_t$ on E_t (using Algorithm 5.2)
6 Set $c := R[x] + R[y]$
7 Set $S := cQ$ on E_p (using Algorithm 5.2)
8 **Output** S

Although an infective version of our countermeasure is possible, we do not recommend to use it. It has been observed in Chapter 3 that the security of an infective algorithm depends on the unpredictability of the final output. It must be computationally infeasible to trace a faulty final result back to secret data. This is only guaranteed if either the output is randomized using a random number generator or if a single induced fault yields a final faulty value, which cannot be

efficiently predicted by the adversary. If the errors, which may result from the induced faults, are from a small, efficiently sampleable set, an adversary might be able to break the system by trying all possible error patterns and comparing the result to the final output S. This yields the value c used to alter the output in Line 7 of Algorithm 6.6 to an unpredictable value. Once c is known, it is easy to recover the faulty result Q on E_p by computing $Q = 1/c \cdot S \bmod \#E_p$. Such an attack uses the same basic principles as the attack by Boneh, DeMillo, and Lipton used in Chapters 2 and 5. A similar approach has been presented by Wagner in [Wag04], who demonstrated that our countermeasure against Bellcore attacks against CRT-RSA is insecure for fault models, where faults only affect a small bounded number of bits (cf. Section 3.6). The number of Sign Change Faults inducable into intermediate variables used in the computation of Q on E_{pt} in Line 4 of Algorithm 6.6 clearly is efficiently sampleable — this fact has been used in order to develop the Sign Change Fault Attacks in Chapter 5. However, Algorithm 6.6 can be used, if hardware countermeasures guarantee that only fault attacks according to the Random Fault Model 1.9 can be mounted. The latter is currently assumed for modern high-end smartcards (as explained in detail in Section 1.4). Once again, one can see that a smartcard is best equipped with both, hardware and software countermeasures.

7. Sign Change Attacks — RSA Revisited

In Chapter 5, we have shown that Sign Change Faults can be used successfully against elliptic curve cryptosystems. In this chapter, we will show that in principle, the same concept can be used against the modular exponentiation algorithm in RSA, as well. Sign Change Faults are much harder to realize for finite field elements, but once accomplished, they allow successful attacks on RSA signatures, which bypass a variety of commonly used countermeasures. We have discussed a variety of approaches to induce Sign Change Faults into elliptic curve points in Section 4.5. At least one of these approaches, i.e., an attack on the crypto co-processor, can change finite field values if the non-adjacent form is used for finite field multiplications. This approach can be used to change the sign of finite field elements, even for other systems than elliptic curve based systems. Therefore, investigating whether Sign Change Faults can be applied to break RSA modular exponentiation is not a purely theoretic question.

In this chapter, we will show that Sign Change Faults can also be used to recover a secret RSA key given a fault model which is stronger than the fault models used for attacks on elliptic curve repeated doubling. Additionally, we will show that this attack does not apply to both versions of repeated squaring alike. The latter result is an example for a situation, where "downwards is better than upwards", which shows that there is no generally better version of repeated squaring. This result is a negation of a result in [FV03], where a certain differential power attack prompted the authors to state that "upwards is better than downwards".

Algorithm 7.1: Left-to-right Rep. Squaring

Input: a message $m \in \mathbb{Z}_N$, a secret RSA key $0 \leq d$, where $l(d)$ denotes the binary length of d, i.e., the number of bits of d, and an RSA modulus N

Output: $m^d \bmod N$

\# **init**

1 Set $y_{l(d)} := 1$

\# **main**

2 For k from $l(d) - 1$ downto 0 do

3 Set $y'_k := y^2_{k+1} \bmod N$

4 If $d_k = 1$ then set $y_k := y'_k \cdot m \bmod N$
else set $y_k := y'_k \bmod N$

5 **Output** y_0

Algorithm 7.2: Right-to-left Rep. Squaring

Input: a message $m \in \mathbb{Z}_N$, a secret RSA key $0 \leq d$, where $l(d)$ denotes the binary length of d, i.e., the number of bits of d, and an RSA modulus N

Output: $m^d \bmod N$

\# **init**

1 Set $y_{-1} := 1$

2 Set $z_{-1} := m$

\# **main**

3 For k from 0 to $l(d) - 1$ do

4 If $d_k = 1$ then set $y_k := y_{k-1} \cdot z_{k-1} \bmod N$
else set $y_k := y_{k-1} \bmod N$

5 Set $z_k := z^2_{k-1} \bmod N$

6 **Output** $y_{l(d)-1}$

Let us first recall the standard repeated squaring algorithms, which we have presented before as Algorithms 2.2 and 2.3. The versions presented here have been rewritten using indexed variables, in order to ensure that different uses of the same variable can be clearly distinguished.

The chances of inducing a Sign Change Fault, which changes the final result, are clearly limited

by the two Algorithms 7.2 and 7.1, since intermediate results are squared. We will investigate the chance that a Sign Change Fault yields a faulty final result for both Algorithm separately. Our overall assumption is that during one execution of any of the two algorithms, only a single fault can be induced.

7.1. Sign Change Faults in Left-to-Right Repeated Squaring

We first investigate Algorithm 7.1. Here, the intermediate variables y_k, y'_k, y_{k+1}, and m are used. Transient faults and permanent faults have a different effect only for attacks on m, all other variables are used only once.

Sign Change in y_{k+1}. A sign change in y_{k+1} results in the correct value for y'_k, since $y_{k+1}^2 = -y_{k+1}^2$. Therefore, a faulty result cannot be returned.

Sign Change in y'_k. A sign change in y'_k yields the negated, i.e., faulty, value $\tilde{y}_k = -y'_k \cdot m^{d_k}$. If this happens during the last iteration, where $k = 0$, the faulty final result $-m^d \bmod N$ is returned. Otherwise, the sign is eliminated in the next round, where $y'_{k-1} = y_k^2$.

Sign Change in y_k. The situation for a sign change in y_k is completely the same as for a sign change in y'_k. It only yields a faulty final result of $-m^d$ if $k = 0$.

Sign Change in m. The value m is used in every iteration. Therefore, transient and permanent errors have a different effect. Although any sign change in m yields a negated value $\tilde{y}_k = -y_k$, this sign is eliminated in the next round, where the value $y_{k-1} = y_k^2$ is computed. Therefore, a sign change of m has no effect on the final result unless $k = 0$. Hence, a transient fault only changes the final result to $-m^d \bmod N$ if it happens in iteration $k = 0$, while a permanent fault guarantees that $-m$ is used in iteration $k = 0$. Since we have $d_0 = 1$ for an RSA secret key, the correct iteration, which has been hit by a fault, cannot be determined. Therefore, we may treat a permanent fault as a transient fault in iteration $k = 0$.

Summary. Summarizing the above considerations, it is clear that any sign change in any variable is corrected by the squaring in Line 3 unless it happens in the last iteration. However, this does not allow to derive any information, since we know that RSA exponents are odd, hence, $d_0 = 1$. This implies that the value $y'_1 = m^d/m \bmod N$ is known to an adversary and he already knows all the values a fault induced during this last iteration could reveal. Therefore, Algorithm 7.1, the left-to-right repeated squaring algorithm, is secure against sign changes.

7.2. Sign Change Faults in Right-to-Left Repeated Squaring

For the right-to-left repeated squaring algorithm, we will analyze the effect of a sign change of any of the intermediate variables y_{k-1}, y_k, z_{k-1}, z_k in the same manner as in the previous section. Here, however, we will see that sign changes on certain variables yield a faulty result regardless of the iteration.

Sign Change in y. A sign change in y_{k-1} results in the faulty value $\tilde{y}_k = -y_k$. This value is multiplied with the correct powers of m in the remainder of the execution, hence, the final result is bound to be faulty, as well. In this case, we have $\tilde{y}_{l(d)-1} = -m^d \bmod N$. A sign change in y_k has the exact same effect as an attack on y_{k-1}.

Sign Change in z. Although z_{k-1} is used twice, both in Line 4 and in Line 5, it is not necessary to investigate both transient and permanent faults. A sign change fault induced into z_{k-1} in Line 4 changes the value y_k to $-y_k$. This only holds if z_{k-1} is used in Line 4, i.e., when $d_k = 1$. Since $z_k = z_{k-1}^2$, the value z_k is always the correct value, regardless of sign changes in any of the two versions of z_{k-1} and regardless of a permanent or transient effect. Only if the variable is affected prior to its use in Line 4, the flipped sign in y_k is not corrected and the final result is the faulty value $y_{l(d)-1} = -m^d \bmod N$. A sign change in z_k has exactly the same effect as a fault induced into z_{k-1}. Faults occurring during Line 5 cannot have any effect on the final result if they occur in the last iteration.

Summary. We see that sign change faults induced into the intermediate variables used in Algorithm 7.2 yield the negated final result for any affected iteration. However, we have also seen that the final faulty result is independent from the affected iteration, it is always $-m^d \bmod N$ if the physical attack was successful. Therefore, the only information that an adversary can get by targeting an intermediate variable is whether an error occurred or not. We usually refer to such fault attacks as oracle attacks, since the adversary only gets a yes/no answer for his question, whether his attack was successful or not (see Section 1.6.2). He does not get a faulty result, which allows to compute any secrets directly. The value $-m^d \bmod N$ cannot be used to derive any information about the secret exponent d. This is clear since any adversary is assumed to know the correct signature $m^d \bmod N$, hence, he can compute $-m^d \bmod N$ himself. This does not require fault attacks.

In order to retrieve information about the secret key using the oracle answer, an adversary needs to be able to mount an attack, where the effect of an induced fault depends heavily on the individual secret key bits. This condition is not met for attacks on y_{k-1} and y_k. They are used in every line, regardless of the secret key bit. For a fault induced into z_{k-1} however, this is different. If the variable z_{k-1} is affected by a fault, the final result $y_{l(d)-1}$ is correct if $d_k = 0$ and faulty if $d_k = 1$. Therefore, if the adversary knows the iteration in which z_{k-1} or z_k is hit, the oracle answer reveals the value of the secret key bit d_k. This resembles a c-safe error as introduced in Section 1.6.2.

In the following, we present the fault model needed for such an attack.

Definition 7.3 (Fault Model: RSA Sign Change Fault Model).
We define the RSA Sign Change Fault Model *with the following parameters.*

location:	*complete control (can target the sign of a specific variable)*
timing:	*precise control (can target a specific iteration in Algorithm 7.2)*
number of bits:	*random number of faulty bits (the sign is flipped modulo N, i.e., all bits of the affected variable may change)*
fault type:	*Sign Change Faults (see Definition 4.6)*
probability:	*certain (every physical attack results in a fault of the desired kind)*
duration:	*transient and permanent faults (both variants are possible, however, it is assumed that an adversary cannot use both kinds during one fault attack)*

The success probability of a Sign Change Fault induction may be lowered from certain to any non-negligible probability. In such a case, the adversary has to perform many attacks on the same iteration, until he can expect to have induced at least one successful Sign Change Fault. This requires that the adversary knows at least a non-negligible lower bound for this success probability and increases his run time significantly. Such an attack represents a Monte-Carlo-algorithm, where every detected fault yields $d_k = 1$ with probability 1, while a zero may be detected incorrectly, since it is possible that all attacks were unsuccessful. With the same consideration, control on location and on timing could be slightly relaxed.

We have seen that Sign Change Attacks are possible in principle on RSA as well. While the left-to-right repeated squaring version ("downwards") is secure against Sign Change Faults, the right-to-left repeated squaring version can be attacked using Fault Model 7.3. Therefore, in this setting, "downwards is better than upwards".

However, one should bear in mind that such attacks face two drawbacks in practice. On the one hand, Sign Change Faults are much harder to realize for RSA than for elliptic curve cryptosystems. On the other hand, it requires an implementation of repeated squaring, which allows c-safe errors using Fault Model 7.3. Even if the success probability of a Sign Change Fault induction is lowered, Definition 7.3 represents a very strong adversary, which may be assumed to be extremely hard to realize for modern high-end smartcards used today. Moreover, the attack on plain RSA as presented in Chapter 2 can be applied assuming less powerful adversaries.

8. Conclusion and Open Problems

In this thesis, we have presented new fault attacks and new countermeasures. As a basis for these results, we have developed a new and comprehensive characterization of known fault models in Chapter 1. This serves as a basis for both, fault attacks and countermeasures.

Fault Attacks. For the RSA cryptosystem, we have investigated the famous result for attacks on the right-to-left repeated squaring algorithm presented by Boneh, DeMillo, and Lipton in [BDL01]. In Chapter 2, we have shown that the original attack contains two minor flaws, which are due to implicit assumptions, which are not guaranteed by the original attack algorithm. We have corrected these flaws. Moreover, we have presented an updated version of the proof of the success probability for the original attack, which we consider to be more convincing than the original proof. We have also extended the attack to the left-to-right repeated squaring algorithm, proving that both versions are susceptible to fault attacks in the same way. These results round off the analysis of fault attacks on plain RSA in the manner described in [BDL01].

For elliptic curve cryptosystems, fault attacks have been considered to be less successful than attacks on the RSA system. Previously reported attacks, e.g., in [BMM00] and [CJ03], represent weak attacks, since they can be fended off easily. However, in Chapter 4, we have shown that contrary to popular believe, the standard countermeasure proposed in [BMM00] and [CJ03] does not render fault attacks useless. We have shown that undetectable faults can be induced for special inputs. Moreover, we have shown that undetectable faults can also be realized for arbitrary inputs using a new fault type, Sign Change Faults. We have shown that this fault type can be induced in practice. Hence, it poses a new threat for elliptic curve cryptosystems using repeated doubling. In Chapter 5, we have defined a new fault model, which uses the new Sign Change Faults. We have applied the new fault model to prove that fault attacks are possible on a wide variety of algorithms for computing scalar multiples of elliptic curve points. All standard algorithms are susceptible to fault attacks in the same manner as plain RSA. Hence, we have generalized the results from Boneh, DeMillo, and Lipton [BDL01] to elliptic curve cryptosystems.

The new Sign Change fault type has been applied to RSA in Chapter 7. It can be used successfully for attacks on the right-to-left repeated squaring algorithm, but not for attacks on the left-to-right repeated squaring algorithm. However, the attacks require a very strong fault model. Nevertheless, our results show that Sign Change Faults are a threat for RSA as well.

Our results about fault attacks show that a larger variety of algorithms and cryptosystems can be successfully attacked by fault attacks than previously known.

Countermeasures. The threat posed by fault attacks is real. Hence, efficient algorithmic countermeasures are needed. We have developed countermeasures for the two most popular and most widely deployed cryptosystems, RSA and elliptic curve cryptosystems.

For RSA, we have concentrated on CRT-RSA, a fast variant of repeated squaring in Chapter 3. This variant is very popular on smartcards due to its advantage in speed. However, it is highly susceptible to fault attacks. As a consequence, we have proposed a new algorithm, which protects the CRT-RSA scheme against known fault attacks. The new algorithm trades speed for security, however, the loss in speed is low, while the gain in security is crucial. We have proved that the new algorithm is secure against fault attacks for the most realistic adversary, and we have proposed modifications, which provide security even in the case of more powerful yet less realistic adversaries. Therefore, our new algorithm is an effective countermeasure, ready for use. The question whether our approach can also be applied for countermeasures for plain RSA is left as an open problem.

Following our new attacks on elliptic curve cryptosystems, we have also proposed a countermeasure against these attacks in Chapter 6. We have presented an algorithm, which can secure every scalar multiplication algorithm for elliptic curves, which does not need field inversions. This restriction is practical, since projective coordinates can be used to represent elliptic curve points. Projective coordinates allow to compute point multiples without using field inversions. We have proved that our new algorithm is secure against all known fault attacks on scalar multiplication algorithms. Therefore, this algorithm is an effective countermeasure as well, ready to be used.

Open Problems. The major problem of fault attacks is to define a model to prove security, which is stronger than the ad-hoc security model used for proofs in this thesis, and which is practical. In principle, the computational security model used for classic security analysis (presented in the introduction) can be used to prove security against fault attacks for special adversaries. However, up to date such proofs are only possible for highly inefficient schemes. Such schemes are not usable in practice, hence, efficient secure schemes are needed. Currently, it is an open problem how to formalize security against fault attacks and how to capture the information provided by this side-channel in a mathematical framework in a practical way. Hence, research in this area should be intensified. As we have explained in Chapter 1, fault attacks are used in practice, hence, once digital signature smartcards become more popular, fault attacks will be a serious threat. This indicates the urge to not only investigate systems for their susceptibility to fault attacks and to develop efficient countermeasures, but also to develop a better theoretical understanding of fault attacks and side-channel attacks in general.

A. Detailed Fault Analysis of Affine Elliptic Curve Addition

A.1. Attacks Targeting $\lambda = (y_1 - y_2)/(x_1 - x_2)$

The parameter λ, the slope of the line defined by P_1 and P_2, is computed using Equation (4.10), which states that

$$\lambda = \frac{y_1 - y_2}{x_1 - x_2}. \tag{4.10}$$

Equation (4.10) computes λ using 5 variables, which can be targeted in a fault attack. As any fault induced into the result λ has the same effect as if λ would be affected the next time it is used, the analysis of a fault induced into λ will be presented later. We first investigate faults in the two parameters y_2 and x_2. Both are only used once in the whole addition formula. We present the analysis of faults affecting y_1 and x_1 afterwards.

- **Fault induced into y_2:**
  Assume that y_2 is targeted by an induced fault, resulting in $y_2 \not\mapsto y_2 + e$ with $e \in \mathbb{F}_p$. As y_2 is only used once, permanent and transient faults have the same effect and need not be considered separately. The induced fault enforces the usage of the faulty values $\tilde{\lambda}$, $\tilde{x}_3$, and $\tilde{y}_3$ in the computation. They take the values

$$\tilde{\lambda} = \frac{y_1 - y_2 - e}{x_1 - x_2} = \frac{y_1 - y_2}{x_1 - x_2} + \hat{e} = \lambda + \hat{e} \qquad \text{where } \hat{e} := \frac{-e}{x_1 - x_2}$$

$$\tilde{x}_3 = \tilde{\lambda}^2 - x_1 - x_2 = \lambda^2 + 2\lambda\hat{e} + \hat{e}^2 - x_1 - x_2 = x_3 + 2\lambda\hat{e} + \hat{e}^2$$

$$\begin{aligned}\tilde{y}_3 &= -y_1 + \tilde{\lambda}\,(x_1 - \tilde{x}_3) = -y_1 + \lambda\left(x_1 - x_3 - 2\lambda\hat{e} - \hat{e}^2\right) + \hat{e}\left(x_1 - x_3 - 2\lambda\hat{e} - \hat{e}^2\right) \\ &= y_3 + \hat{e}\left(x_1 - x_3 - 2\lambda\hat{e} - \hat{e}^2 - 2\lambda^2 - \lambda\hat{e}\right) = y_3 + \hat{e}\left(x_1 - x_3 - (\hat{e} + \lambda)\,(2\lambda + \hat{e})\right).\end{aligned}$$

  The resulting faulty point $\tilde{P}_3 = (\tilde{x}_3, \tilde{y}_3)$ is a valid faulty point only if it satisfies the Weierstrass Equation (4.2), i.e., if $\tilde{y}_3^2 - \tilde{x}_3^3 - A\tilde{x}_3 - B \equiv 0 \bmod p$. This can be used to compute $\hat{e}$. We have

$$\begin{aligned}&\tilde{y}_3^2 - \tilde{x}_3^3 - A\tilde{x}_3 - B \\ &= \hat{e} \cdot \left(\hat{e} - \frac{2y_2}{x_1 - x_2}\right) \cdot \left(\hat{e}^2 \cdot (x_2 - x_1) + 2\hat{e} \cdot (y_2 - y_1) + \right. \\ &\qquad \left. \frac{x_1^3 - 3x_1^2x_2 - Ax_1 + Ax_2 - 2y_2^2 + 2x_2^3 + 2y_1y_2}{x_1 - x_2}\right) + R, \\ &\quad \text{where } R = y_3^2 - x_3^3 - Ax_3 - B + \hat{e} \cdot \frac{2y_1(y_1^2 - y_2^2 - x_1^3 + x_2^3 + Ax_2 - Ax_1)}{(x_1 - x_2)^2}, \\ &= (x_2 - x_1) \cdot \hat{e} \cdot \left(\hat{e} - \frac{2y_2}{x_1 - x_2}\right) \cdot \left(\hat{e} + \lambda + \sqrt{2x_1 + x_2}\right) \cdot \left(\hat{e} + \lambda - \sqrt{2x_1 + x_2}\right), \end{aligned} \tag{A.1}$$

by standard arithmetic (using Maple). Here, we have $R \equiv 0$. This becomes evident by applying the Weierstrass equation, using $y_i^2 = x_i^3 + Ax_i + B$ for all $i \in \{1, 2, 3\}$. It can be applied since we know that P_1, P_2, and the correct result P_3 are valid points on the elliptic curve. Now, Equation (A.1) shows that we have between 2 and 4 solutions for $\hat{e}$, since we assume that $x_1 \not\equiv x_2$:

$$\hat{e} \in \left\{0,\ \frac{2y_2}{x_1 - x_2},\ -\lambda \pm \sqrt{2x_1 + x_2}\right\} \tag{A.2}$$

$$\Rightarrow\ e \in \left\{0, -2y_2,\ (x_1 - x_2) \cdot \left(\lambda \pm \sqrt{2x_1 - x_2}\right)\right\}. \tag{A.3}$$

The first value $e = 0$ can be neglected as it represents an error-free computation. The second value $e = -2y_2$ represents the case where $-P_2$ instead of P_2 is used in the addition formula. The third and fourth value exist only if the square root exists. This depends on the input points P_1 and P_2.

- **Fault induced into x_2:**
  Let us now assume that x_2 is affected by a fault, resulting in $x_2 \overset{\lightning}{\mapsto} x_2 + e$ with $e \in \mathbb{F}_p$. Again, x_2 is only used once, hence, permanent and transient faults have the same effect and need not be considered separately. This results in the following faulty value $\tilde{\lambda}$:

$$\tilde{\lambda} = \frac{y_1 - y_2}{x_1 - x_2 - e} = \frac{y_1 - y_2}{x_1 - x_2} + \hat{e} = \lambda + \hat{e} \qquad \text{where } \hat{e} := \frac{e(y_1 - y_2)}{(x_1 - x_2)^2 - e(x_1 - x_2)}$$

  The latter shows that we can write $\tilde{\lambda} = \lambda + \hat{e}$ just as in the analysis of a fault induced into y_2. If $e \equiv (x_1 - x_2)$, the device will encounter an arithmetical error (division by zero), which will be detected as a general error. Hence, $e \equiv (x_1 - x_2)$ cannot create a valid faulty point, and we may assume that $e \not\equiv (x_1 - x_2) \bmod p$. Since the value x_2 is not used again, a fault induced into x_2 yields the exact same values for $\tilde{x}_3$ and $\tilde{y}_3$, and therefore also the same values for $\hat{e}$ as in Equation (A.2):

$$\hat{e} \in \left\{0,\ \frac{2y_2}{x_1 - x_2},\ -\lambda \pm \sqrt{2x_1 + x_2}\right\}$$

$$\Rightarrow\ e \in \left\{0,\ 2y_2 \cdot \frac{x_1 - x_2}{y_1 + y_2},\ (x_1 - x_2) \pm \frac{(y_1 - y_2)}{\sqrt{2x_1 + x_2}}\right\}.$$

  Again, the solution $e = 0$ does not yield a valid faulty point, since it yields the correct point. It is therefore not a desired solution for e. The values for e exist only if the denominators exist in $\mathbb{F}_p$, i.e., if they are not equal to zero, and if the square root exists. Both depends on the input points P_1 and P_2. Consequently, there are between 0 and 3 values for e such that a valid faulty point $\tilde{P}_3$ is returned.

- **Fault induced into y_1:**
  The analysis for faults induced into y_1 has been presented in detail in Section 4.3.1.1. However, we will repeat the analysis briefly for completeness.

  For a fault induced into the variable $y_1 \overset{\lightning}{\mapsto} y_1 + e$, $e \in \mathbb{F}_p$, we need to investigate both transient and permanent faults, because y_1 is used again in the computation of y_3, i.e., Equation (4.9).

Transient Faults. Given a transient fault, we have

$$\tilde{\lambda} = \frac{y_1 + e - y_2}{x_1 - x_2} = \frac{y_1 - y_2}{x_1 - x_2} + \hat{e} = \lambda + \hat{e} \qquad \text{where } \hat{e} := \frac{e}{x_1 - x_2}$$

This represents the same situation which has been analyzed for a fault induced into y_2. We can write $\tilde{\lambda}$ as $\tilde{\lambda} = \lambda + \hat{e}$, and the values of $\tilde{x}_3$ and $\tilde{y}_3$ are unchanged compared to a fault affecting y_2 as well, since the effect of the fault in y_1 is transient. Therefore, we know all possible solutions for $\hat{e}$ from Equation (A.2):

$$\begin{aligned} \hat{e} &\in \left\{0,\ \frac{2y_2}{x_1 - x_2},\ -\lambda \pm \sqrt{2x_1 + x_2}\right\} \\ \Rightarrow\ e &\in \left\{0,\ 2y_2,\ (x_1 - x_2) \cdot \left(-\lambda \pm \sqrt{2x_1 + x_2}\right)\right\}. \end{aligned} \tag{A.4}$$

These results for e show that one transient fault which produces a valid faulty point depends on the value of y_2, even if it is induced into y_1. This is caused by the fact that y_1 is used later for the computation of y_3 in Equation (4.9). One might expect that negating y_1 with the error $e = -2y_1$ would yield a valid point, since the valid input point $(x_1, -y_1)$ is used. However, this does not yield a valid faulty point $(\tilde{x}_3, \tilde{y}_3)$. For this to happen, the value y_1 used in Equation (4.9) would have to be faulty as well. This requires a permanent fault, which we investigate now.

Permanent Faults. Given a permanent fault in y_1, the computations of $\tilde{\lambda}$ and $\tilde{x}_3$ are the same as shown for a fault induced into y_2, but the computation of $\tilde{y}_3$ is different. We get the faulty output $\tilde{P}_3 = (\tilde{x}_3, \tilde{y}_3)$ with the values

$$\tilde{x}_3 = x_3 + 2\lambda\hat{e} + \hat{e}^2$$

$$\begin{aligned} \tilde{y}_3 &= -y_1 - e + \tilde{\lambda}\,(x_1 - \tilde{x}_3) = y_3 - e + \hat{e}\,(x_1 - x_3 - \hat{e}\,(1 + \lambda)\,(2\lambda + \hat{e})) \\ &= \quad y_3 + \hat{e}\,(x_1 - x_3 - \hat{e}\,(1 + \lambda)\,(2\lambda + \hat{e})) - \hat{e}\,(x_1 - x_2)\,. \end{aligned}$$

As shown before, we get solutions for $\hat{e}$ by applying the Weierstrass equation (4.2), and solving the resulting equation for $\hat{e}$. The computation is completely analogous to the analysis for an attack targeting y_2, where we described the computation in greater detail. We have

$$\begin{aligned} &\tilde{y}_3^2 - \tilde{x}_3^3 - A\tilde{x}_3 - B = \\ &(x_1 - x_2) \cdot \hat{e} \cdot \left(\hat{e} + \frac{2y_1}{x_1 - x_2}\right) \cdot \left(\hat{e} + \lambda + \sqrt{x_1 + 2x_2}\right) \cdot \left(\hat{e} + \lambda - \sqrt{x_1 + 2x_2}\right), \end{aligned}$$

with standard arithmetic (using Maple). The possible solutions for e are

$$e \in \left\{0,\ -2y_1,\ (x_1 - x_2) \cdot \left(-\lambda \pm \sqrt{x_1 + 2x_2}\right)\right\}. \tag{A.5}$$

These solutions are completely symmetric to the solutions for a fault induced into y_2.

- **Fault induced into x_1:**
  Finally, we assume that x_1 is targeted by an attack, resulting in $x_1 \not\mapsto x_1 + e$ with $e \in \mathbb{F}_p$.

  Transient Faults. We first investigate a transient fault, i.e., we assume that when x_1 is used in Equation (4.8) — the computation of x_3 — it has the correct value again. We get the following value for $\tilde{\lambda}$:

$$\tilde{\lambda} = \frac{y_1 - y_2}{x_1 + e - x_2} = \frac{y_1 - y_2}{x_1 - x_2} + \hat{e} = \lambda + \hat{e} \qquad \text{where } \hat{e} := \frac{-e(y_1 - y_2)}{(x_1 - x_2)^2 + e(x_1 - x_2)}$$

  Again, the value $e \equiv (x_1 - x_2)$ raises an arithmetic exception, which will be detected by the smartcard. It cannot used to create a valid faulty result and we may assume that this case does not occur. As both $\tilde{x}_3$ and $\tilde{y}_3$ are computed using only correct values besides $\tilde{\lambda}$, we know the solutions for $\hat{e}$ from the analysis of a fault induced into y_2, Equation (A.2). We get

$$\begin{aligned} \hat{e} &\in \left\{0,\ \frac{2y_2}{x_1 - x_2},\ -\lambda \pm \sqrt{2x_1 + x_2}\right\} \\ \Rightarrow\ e &\in \left\{0,\ -2y_2 \cdot \frac{x_1 - x_2}{y_1 + y_2},\ -(x_1 - x_2) \pm \frac{(y_1 - y_2)}{\sqrt{2x_1 + x_2}}\right\}. \end{aligned}$$

  This result is the same as for a fault induced into x_2, with opposite signs.

  Permanent Faults. Now, we investigate the result of a permanent fault induced into x_1. Here, we also use the faulty value $\tilde{x}_1 = x_1 + e$ in Equations (4.8) and (4.9). We get

$$\tilde{\lambda} = \frac{y_1 - y_2}{x_1 + e - x_2}$$

$$\tilde{x}_3 = \tilde{\lambda}^2 - x_1 - e - x_2 = \left(\frac{y_1 - y_2}{x_1 + e - x_2}\right)^2 - x_1 - x_2 - e$$

$$\begin{aligned} \tilde{y}_3 &= -y_1 + \tilde{\lambda}\,(x_1 + e - \tilde{x}_3) \\ &= -y_1 + \left(\frac{y_1 - y_2}{x_1 + e - x_2}\right) \cdot \left(x_1 + e - \left(\frac{y_1 - y_2}{x_1 + e - x_2}\right)^2 + x_1 + x_2 + e\right) \\ &= -y_1 + \left(\frac{y_1 - y_2}{x_1 + e - x_2}\right) \cdot (2x_1 + x_2 + 2e) - \left(\frac{y_1 - y_2}{x_1 + e - x_2}\right)^3. \end{aligned}$$

  As noted before, the value $e \equiv (x_2 - x_1)$ yields an arithmetic error. Therefore, we may assume that $e \not\equiv (x_2 - x_1)$. To compute solutions for e such that $\tilde{P}_3 = (\tilde{x}_3, \tilde{y}_3)$ is a valid faulty point on E, we apply the Weierstrass equation (4.2), which must hold, i.e., we solve $\tilde{y}_3^2 - \tilde{x}_3^3 - A\tilde{x}_3 - B = 0$. This is an equation with a leading term of e^6. Consequently, we get six roots for e in the complex plane. The first one is the trivial root, $e = 0$. Two other roots are derived from the fact that changing the x-coordinate of the input point P_1 might result in at most two different faulty points $\tilde{P}_1$, which are valid point on the elliptic curve E. They represent the intersections of the horizontal line $y = y_1$ with the elliptic curve E, which are different from P_1. In this case, the resulting point $\tilde{P}_3$ is certainly a valid faulty

point on E, because both input points are. In order to compute these points, we need to compute all solutions of the Weierstrass Equation (4.2), i.e., we need to solve the equation

$$y_1^2 \equiv x_1^3 + Ax_1 + B \equiv x^3 + Ax + B. \tag{A.6}$$

It has three roots for x. The corresponding values for e are easily derived by $e = x - x_1$ for any solution x of Equation (A.6). The first solution, $x = x_1 \Rightarrow e = 0$, is known, the others yield the values

$$e = -\frac{3x_1}{2} \pm \sqrt{-\frac{3}{4}x_1^2 - A}.$$

Of the other three roots, two are complex numbers, which cannot yield a possible solution for e in $\mathbb{F}_p$ unless the imaginary part is equal to 0. The last root represents a valid value for e if the square root and cubic root exist for the given values of x_1, x_2, y_1, and y_2. The formulas, however, are somewhat lengthy. They have been computed using Maple and standard arithmetic.

$$e = \frac{1}{2} \cdot \alpha + \frac{2x_2^2}{\alpha} - x_1,$$
$$\text{where } \alpha := \sqrt[3]{4(y_1 - y_2)^2 - 8x_2^3 + 4(y_1 - y_2) \cdot \sqrt{(y_1 - y_2)^2 - 4x_2^3}},$$
$$\text{and} \quad e_{\text{complex}} = -\frac{\alpha}{2} - \frac{x_2^2}{\alpha} - x_1 \pm \frac{1}{2}i\sqrt{3} \cdot \left(\frac{\alpha}{2} - \frac{2x_2^2}{\alpha}\right).$$

A.2. Attacks Targeting $x_3 = \lambda^2 - x_1 - x_2$

The parameter x_3 is computed using Equation (4.8), which states that

$$x_3 = \lambda^2 - x_1 - x_2. \tag{4.8}$$

The value x_2 is not used again after this line, hence, transient and permanent faults have the same effect. The other values, x_3, x_1, and λ, are used again, therefore, transient and permanent faults must be investigated separately. Note that as any fault induced into x_3 in this line of computation yields the same result as if it was induced into x_3 in Equation (4.4), it will be analyzed later.

- **Fault induced into x_1:**
  If a fault is induced into x_1 in Equation (4.3), we have $x_1 \mapsto\!\!\!\!\!\!\not\;\; x_1 + e$, $e \in \mathbb{F}_p$, i.e.,

$$\begin{aligned} & \tilde{x}_3 = \lambda^2 - (x_1 + e) - x_2 = x_3 - e \\ \text{and} \quad & \tilde{y}_3 = -y_1 + \lambda \cdot (x_1 - (x_3 - e)) = y_3 + \lambda e, && \text{in case of a transient fault,} \\ \text{or} \quad & \tilde{y}_3 = -y_1 + \lambda \cdot (x_1 + e - (x_3 - e)) = y_3 + 2\lambda e && \text{in case of a permanent fault.} \end{aligned}$$

Transient Faults. We first investigate a transient fault. In order for $(\tilde{x}_3, \tilde{y}_3)$ to be a valid point on the curve, it has to satisfy the Weierstrass Equation (4.2), i.e., it must hold that

$$\tilde{y}^2 \equiv \tilde{x}^3 + A\tilde{x} + B \bmod p.$$

$$\Rightarrow \quad e^3 + e^2\left(\lambda^2 - 3x_3\right) + e\left(3x_3^2 + A + 2y_3\lambda\right) \equiv 0 \bmod p$$
$$\Rightarrow \quad e \in \left\{0, \frac{3x_3 - \lambda^2}{2} \pm \frac{1}{2}\sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}\right\} \bmod p. \tag{A.7}$$

Again, these formulas have been computed using standard arithmetic. Since $e \equiv 0$ does not yield a valid faulty point, we only have possible values for e if the square root exists.

Permanent Faults. If the fault is permanent, the result follows in the same manner. Here, e must satisfy

$$e \in \left\{0, \frac{3x_3 - 2\lambda^2}{2} \pm \frac{1}{2}\sqrt{16\lambda^4 - 24\lambda^2 x_3 - 3x_3^2 - 4A - 16y_3\lambda}\right\} \bmod p. \tag{A.8}$$

In both cases the result $e = 0$ represents the correct result, it cannot yield a valid faulty result.

- **Fault induced into x_2:**
  If $x_2 \not\mapsto x_2 + e$, $e \in \mathbb{F}_p$, is hit during a fault attack, we have the same situation as for a fault induced into x_1, with the only difference that transient faults and permanent faults yield the same result as a transient fault on x_1. This is due to the fact that x_2 is no longer used after the computation of x_3. Hence, we have $\tilde{x}_3 = x_3 - e$ and $\tilde{y}_3 = y_3 + \lambda e$. Therefore, we get the solutions

$$e \in \left\{0, \frac{3x_3 - \lambda^2}{2} \pm \frac{1}{2}\sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}\right\} \bmod p.$$

- **Fault induced into λ:**
  The analysis of a fault being induced into λ has been presented in detail in Section 4.3.1.2. However, we will repeat the analysis briefly for completeness.

  For faults affecting λ, we investigate three different cases. First, the variable λ could have been affected prior to its first use in Equation (4.8). The effect of the induced fault could be transient or permanent. But then, it could also be that the result λ^2 could have been affected by the fault. We will investigate a fault affecting λ first. In this case, the faulty value $\lambda + e$ is used and we get the faulty results

$$\begin{aligned}
& \tilde{x}_3 = (\lambda + e) - x_1 - x_2 = x_3 + 2\lambda e + e^2 \\
\text{and} \quad & \tilde{y}_3 = -y_1 + \lambda \cdot \left(x_1 - (x_3 + 2\lambda e + e^2)\right) \\
& \quad = y_3 - 2\lambda^2 e - \lambda e^2 && \text{in case of a transient fault,} \\
\text{or} \quad & \tilde{y}_3 = -y_1 + (\lambda + e) \cdot \left(x_1 - (x_3 + 2\lambda e + e^2)\right) \\
& \quad = y_3 + e\left(x_1 - x_3 - 2\lambda e - e^2 - 2\lambda^2 - \lambda e\right) \\
& \quad = y_3 + e\left(x_1 - x_3 - (2\lambda + e) \cdot (e + \lambda)\right) && \text{in case of a permanent fault.}
\end{aligned}$$

Transient Faults. Let $\hat{e} := -2\lambda e - e^2$. In this case, the resulting faulty point $\tilde{P}_3$ is equal to $(x_3 - \hat{e}, y_3 + \lambda\hat{e})$. This situation is the same as the one analyzed for a transient fault

induced into x_1 in the computation of x_3. The analysis resulted in Equation (A.7), which yields the condition

$$\hat{e} \in \left\{0, \frac{3x_3 - \lambda^2}{2} \pm \frac{1}{2}\sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}\right\} \bmod p.$$

As $\hat{e}$ is a quadratic equation in e, this yields 6 possible solutions for e, namely

$$e \in \left\{0, -2\lambda, -\lambda \pm \sqrt{\frac{3}{2}(\lambda^2 - x_3) \pm \sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}}\right\} \bmod p. \quad \text{(A.9)}$$

Permanent Faults. For permanent faults in λ, the situation is the same as for a fault induced into y_2 in the computation of $\lambda = (y_2 - y_1)/(x_2 - x_1)$. It has been analyzed there yielding the conditions stated in Equation (A.2):

$$e \in \left\{0, \frac{2y_2}{x_1 - x_2}, -\lambda \pm \frac{\sqrt{y_1^2 - y_2^2 + x_1^3 - 3x_1^2 x_2 - Ax_1 + Ax_2 - 2x_2^3}}{x_1 - x_2}\right\}$$

Attack targeting λ^2. As the value λ is squared in the computation of x_3, it is also possible that the final result λ^2 is affected by an induced fault. In this case, we have $\lambda^2 \mapsto\!\!\!\!\!\!\not\;\;\lambda^2 + e$, for some $e \in \mathbb{F}_p$. As the value λ^2 is only used once, we do not need to differentiate between transient and permanent faults. This situation can be expressed by the formulas

$$\begin{aligned} x_3' &= \lambda^2 + e - x_1 - x_2 = x_3 + e \\ y_3' &= -y_1 + \lambda \cdot (x_1 - x_3 - e) = y_3 - \lambda e. \end{aligned}$$

Setting $\hat{e} = -e$ yields the faulty result $\tilde{P}_3 = (x_3 - \hat{e}, y_3 + \lambda\hat{e})$. It has been analyzed before when a transient fault in x_1 in the computation of x_3 has been assumed. The analysis resulted in Equation (A.7), which states the condition

$$\begin{aligned} & \hat{e} \in \left\{0, \frac{3x_3 - \lambda^2}{2} \pm \frac{1}{2}\sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}\right\} \bmod p. \\ \Rightarrow \quad & e \in \left\{0, \frac{\lambda^2 - 3x_3}{2} \pm \frac{1}{2}\sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}\right\} \bmod p. \end{aligned}$$

A.3. Attacks Targeting $y_3 = -y_1 + \lambda \cdot (x_1 - x_3)$

In Equation (4.9), y_3 is computed using the four variables y_1, λ, x_1, and x_3 as

$$y_3 = -y_1 + \lambda\,(x_1 - x_3)\,. \quad \text{(4.9)}$$

The three variables y_1, λ, x_1 and y_3 are no longer used in the addition formula after this line of computation, therefore, we need not differentiate between transient and permanent faults. The value of x_3, however, is used afterwards when the resulting point $P_3 = (x_3, y_3)$ is returned by the addition formula. Therefore, in the case of x_3, a differentiation between transient and permanent faults is necessary.

- **Fault induced into y_1:**
  If a fault is induced into y_1 in Equation (4.9), we have

  $$\tilde{y}_3 = -y_1 + e + \lambda \cdot (x_1 - x_3) = y_3 + e,$$

  for $e \in \mathbb{F}_p$.

  We know that for a fixed x_3-coordinate only the intersections of the vertical line at $x = x_3$ with the elliptic curve E yield valid points. For any line, there exist at most three intersections with the elliptic curve, which are the correct point $P_3 = (x_3, y_3)$, the negated point $-P_3 = (x_3, -y_3)$ and the point at infinity $\mathcal{O}$. As $\mathcal{O}$ is a special point specially treated in affine coordinates, it cannot be represented by a tuple from $\mathbb{F}_p^2$. Therefore, we may assume that it cannot be created by any fault which changes coordinates within $\mathbb{F}_p$. This implies that the only possibility for e to yield a valid faulty point $\tilde{P}_3$ on E is that the sign of y_3 is flipped by that fault. This requires that $\tilde{y}_3 := y_3 + e \equiv -y_3 \bmod p$, hence,

  $$e \equiv -2y_3 \bmod p. \tag{A.10}$$

- **Fault induced into λ:**
  Given a fault induced into the slope λ in Equation (4.9), we have

  $$\tilde{y}_3 = -y_1 + (\lambda + e) \cdot (x_1 - x_3) = y_3 + e \cdot (x_1 - x_3)\,,$$

  where $e \in \mathbb{F}_p$.

  The only possibility for $\tilde{P}_3 = (x_3, \tilde{y}_3)$ to be a valid point on E is that $\tilde{y}_3 = -y_3$. This implies that $-2y_3 = e \cdot (x_1 - x_3)$. If $x_1 - x_3 = 0$, which may only happen if $P_1 = \pm P_3$, the error has no effect, and the correct result y_3 is returned. We may therefore assume that $x_1 \neq x_3$ and conclude that a valid faulty point on E results only for

  $$e \equiv -2y_3/(x_1 - x_3) \bmod p \qquad \text{if } x_1 \neq x_3.$$

- **Fault induced into x_1:**
  A fault induced into the point coordinate x_1 in Equation (4.9) yields

  $$\tilde{y}_3 = -y_1 + \lambda \cdot (x_1 + e - x_3) = y_3 + e\lambda,$$

  with $e \in \mathbb{F}_p$.

  In order for $\tilde{y}_3$ to be equal to $-y_3$, the error e must satisfy

  $$e \equiv -2y_3/\lambda \bmod p \qquad \text{if } y_1 \neq y_2.$$

  Note that if $\lambda \equiv 0$, the error would not have any effect and the correct value would be returned.

- **Fault induced into x_3:**
  As the variable x_3 is used after the computation for Equation (4.9), we differentiate between a transient fault in x_3 and a permanent fault in x_3. For a transient fault, the point $\tilde{P}_3 = (x_3, \tilde{y}_3)$ is returned, for a permanent fault, the output is $\tilde{P}_3 = (\tilde{x}_3, \tilde{y}_3) = (x_3 + e, \tilde{y}_3)$.
  In both cases, we have

  $$\tilde{y}_3 = -y_1 + \lambda \cdot (x_1 - x_3 - e) = y_3 - e\lambda,$$

  for $e \in \mathbb{F}_p$.

Transient Faults. Given a transient fault, a valid faulty point $\tilde{P}_3$ is returned only if $\tilde{y}_3 = -y_3$, hence, if

$$e \equiv 2y_3/\lambda \bmod p \qquad \text{if } y_1 \neq y_2.$$

Not surprisingly, this is the same result as for a fault induced into x_1 with opposite signs. Again, we may assume that $\lambda \neq 0$ as otherwise the error would not have any effect and the correct value would be returned.

Permanent Faults. Given a permanent fault, a faulty point $\tilde{P}_3 = (\tilde{x}_3, \tilde{y}_3)$ is returned, where

$$\begin{aligned}\tilde{x}_3 &= x_3 + e \\ \tilde{y}_3 &= -y_1 + \lambda \cdot (x_1 - x_3 - e) = y_3 - e\lambda.\end{aligned}$$

Again, we have $e \in \mathbb{F}_p$. With $\hat{e} := -e$, this situation is the same as the one analyzed for a transient fault induced into x_1 in the computation of x_3. Equation (A.7) states the condition

$$\begin{aligned} & \hat{e} \in \left\{0, \frac{3x_3 - \lambda^2}{2} \pm \frac{1}{2}\sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}\right\} \bmod p. \\ \Rightarrow \qquad & e \in \left\{0, \frac{\lambda^2 - 3x_3}{2} \pm \frac{1}{2}\sqrt{\lambda^4 - 6\lambda^2 x_3 - 3x_3^2 - 4A - 8y_3\lambda}\right\} \bmod p. \end{aligned}$$

- **Fault induced into y_3:**
  Given a fault induced into the final y-coordinate y_3 in Equation (4.4), we have

  $$\tilde{y}_3 = y_3 + e,$$

  where $e \in \mathbb{F}_p$.

  Again, the only possibility for $\tilde{P}_3 = (x_3, \tilde{y}_3)$ to be a valid point on E is that $\tilde{y}_3 = y_3 + e = -y_3$. This implies that

  $$e \equiv -2y_3 \bmod p. \tag{A.11}$$

Bibliography

[ABF+02] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert, *Fault attacks on RSA with CRT: Concrete results and practical countermeasures*, Cryptographic Hardware and Embedded Systems — CHES 2002 (B. S. Kaliski, Jr., Ç. K. Koç, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2523, Springer-Verlag, 2002, pp. 260–275.

[AK96] Ross J. Anderson and Markus G. Kuhn, *Tamper resistance – a cautionary note*, Proceedings of the Second USENIX Workshop on Electronic Commerce (Oakland, California), USENIX Association, November 18-21 1996, pp. 1 – 11.

[AK97] Ross J. Anderson and Markus G. Kuhn, *Low cost attacks on tamper resistant devices*, Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997 (M. Lomas et al., ed.), Lecture Notes in Computer Science, vol. 1361, Springer-Verlag, 1997, pp. 125–136.

[ANS99] ANSI X9.63-199x: *Public key cryptography for the financial services industry: Key agreement and key transport using elliptic curve cryptography*, american bankers association, 8 January 1999, Working Draft, `http://cnscenter.future.co.kr/resource/crypto/standard/ansi_x9/x9-63-01-08-99.pdf`.

[Bar96] Paul Barrett, *Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor*, Advances in Cryptology — CRYPTO'86 (A. M. Odlyzko, ed.), Lecture Notes in Computer Science, vol. 263, Springer-Verlag, 1996, pp. 311–323.

[BCN+04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan, *The sorcerer's apprentice guide to fault attacks*, Cryptology ePrint Archive, Report 2004/100, 2004, `http://eprint.iacr.org/2004/100/`.

[BD00] Dan Boneh and Glenn Durfee, *Cryptanalysis of RSA with private key d less than* $N^{0.292}$, IEEE Transactions on Information Theory **46** (2000), no. 4, 1339–1349.

[BDH+98] Feng Bao, Robert H. Deng, Yongfei Han, Albert B. Jeng, A. Desai Narasimhalu, and Teow-Hin Ngair, *Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults*, Security Protocols Workshop 1997 (B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, eds.), Lecture Notes in Computer Science, vol. 1361, Springer-Verlag, 1998, pp. 115–124.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton, *On the importance of checking cryptographic protocols for faults*, Advances in Cryptology — EUROCRYPT'97 (W. Fumy, ed.), Lecture Notes in Computer Science, vol. 1233, Springer-Verlag, 1997, pp. 37–51.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton, *On the importance of eliminating errors in cryptographic computations*, J. Cryptology **14** (2001), no. 2, 101–119. MR 1 830 690

[BM04] Johannes Blömer and Alexander May, *A generalized Wiener attack on RSA*, Public Key Cryptography — PKC 2004 (F. Bao, R. Deng, and J. Zhou, eds.), Lecture Notes in Computer Science, vol. 2947, Springer-Verlag, 2004, pp. 1–13.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller, *Differential fault attacks on elliptic curve cryptosystems*, Advances in Cryptology — CRYPTO 2000 (Mihir Bellare, ed.), Lecture Notes in Computer Science, vol. 1880, Springer-Verlag, 2000, pp. 131–146.

[Boo51] Andrew D. Booth, *A signed binary multiplication technique*, Quart. Journ. Mech. and Applied Math. **IV** (1951), no. 2, 236–240.

[BOPV03] Lejla Batina, Siddika B. Örs, Bart Preneel, and Joos Vandewalle, *Hardware architectures for public key cryptography*, Integration, the VLSI Journal **34** (2003), no. 1–2, 1–64.

[BOS03] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert, *A new CRT-RSA algorithm secure against Bellcore attacks*, Conference on Computer and Communications Security — CCS 2003 (V. Atluri and P. Liu, eds.), ACM SIGSAC, ACM Press, 2003, pp. 311–320.

[BOS04] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert, *Sign change fault attacks on elliptic curve cryptosystems*, Cryptology ePrint Archive, Report 2004/227, 2004, http://eprint.iacr.org/2004/227/.

[BR95a] Mihir Bellare and Phillip Rogaway, *Optimal asymmetric encryption*, Advances in cryptology — EUROCRYPT '94, Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 92–111. MR 98i:94016

[BR95b] Mihir Bellare and Phillip Rogaway, *Optimal Asymmetric Encryption — how to encrypt with RSA*, http://www-cse.ucsd.edu/users/mihir/papers/oae.ps.gz, an extended abstract of this paper has been published as [BR95a], 1995.

[BS97] Eli Biham and Adi Shamir, *Differential fault analysis of secret key cryptosystems*, Advances in Cryptology — CRYPTO '97 (B. S. Kaliski, Jr., ed.), Lecture Notes in Computer Science, vol. 1294, Springer-Verlag, 1997, pp. 513–525.

[BS03] Johannes Blömer and Jean-Pierre Seifert, *Fault based cryptanalysis of the Advanced Encryption Standard (AES)*, Financial Cryptography — FC 2003, Lecture Notes in Computer Science, vol. 2742, Springer-Verlag, 2003, pp. 162–181.

[BSS99] Ian Blake, Gadiel Seroussi, and Nigel Smart, *Elliptic curves in cryptography*, London Mathematical Society Lecture Note Series, vol. 265, Cambridge University Press, 1999.

[CC86] David V. Chudnovsky and Gregory V. Chudnovsky, *Sequences of numbers generated by addition in formal groups and new primality and factorization tests*, Advances in Applied Mathematics **7** (1986), no. 4, 385–434.

[CCD00] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous, *Differential power analysis in the presence of hardware countermeasures*, Cryptographic Hardware and Embedded Systems — CHES 2000, Lecture Notes in Computer Science, vol. 1965, Springer-Verlag, 2000, pp. 252–263.

[CCJ03] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye, *Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity*, Cryptology ePrint Archive, Report 2003/237, 2003, http://eprint.iacr.org/2003/237/.

[Cer00] Certicom, *Current public-key cryptographical systems — the elliptic curve cryptosystem*, A Certicom Whitepaper, July 2000, http://www.inf.pucrs.br/~eduardob/disciplinas/tc/luis_aristeu/docs/EccWhite2.pdf.

[CJ03] Mathieu Ciet and Marc Joye, *Elliptic curve cryptosystems in the presence of permanent and transient faults*, Cryptology ePrint Archive, Report 2003/028, 2003, http://eprint.iacr.org/2003/028/.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi, *Towards sound approaches to counteract power-analysis attacks*, Advances in Cryptology — CRYPTO '99 (M. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer-Verlag, 1999, pp. 398–412.

[CKN00] Jean-Sébastien Coron, Paul C. Kocher, and David Naccache, *Statistics and secret leakage*, Financial Cryptography — FC 2000 (Y. Frankel, ed.), Lecture Notes in Computer Science, vol. 1962, Springer-Verlag, 2000, p. 157 ff.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono, *Efficient elliptic curve exponentiation using mixed coordinates*, Advances in Cryptology — ASIACRYPT'98 (K. Ohta and D. Pei, eds.), Lecture Notes in Computer Science, vol. 1514, Springer-Verlag, 1998, pp. 51–65.

[Cor99] Jean-Sébastien Coron, *Resistance against differential power analysis for elliptic curve cryptosystems*, Cryptographic Hardware and Embedded Systems — CHES '99 (Ç.K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1717, Springer-Verlag, 1999, pp. 292–302.

[CQ82] Christophe Couvreur and Jean-Jacques Quisquater, *Fast decipherment algorithm for RSA public-key cryptosystem*, Electronic Letters **18** (1982), no. 21, 905–907.

[Deu41] Max Deuring, *Die Typen der Multiplikatorenringe elliptischer Funktionenkörper*, Abh. Math. Sem. Hansischen Univ. **41** (1941), 197–272.

[DH76] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **IT-22** (1976), no. 6, 644–654.

[DLS81] Peter J. Downey, Benton L. Leong, and Ravi Sethi, *Computing sequences with addition chains*, SIAM J. Comput. **10** (1981), no. 3, 638 – 646.

[Dot02] Emmanuelle Dottax, *Fault attacks on NESSIE signature and identification schemes*, NESSIE public report Nr. NES/DOC/ENS/WP5/031/1, 8 Oct 2002, https://www.cosic.esat.kuleuven.ac.be/nessie/reports/phase2/SideChan_1.pdf.

[EK90] Ömer Eğecioğlu and Çetin K. Koç, *Fast modular exponentiation*, Communication, Control, and Signal Processing (1990), 188 – 194.

[ElG85] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory **31** (1985), 469–472.

[FGKS02] Wieland Fischer, Christophe Giraud, Erik W. Knudsen, and Jean-Pierre Seifert, *Parallel scalar multiplication on general elliptic curves over $\mathbb{F}_p$ hedged against non-differential side-channel attacks*, Cryptology ePrint Archive, Report 2002/007, 2002, http://eprint.iacr.org/2002/007/.

[FIP00] *U.S. Department of Commerce/National Institute of Standards and Technology (NIST): Digital Signature Standard (DSS), Federal Information Processing Standards Publication (FIPS PUB 186.2)*, 2000, http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf.

[FS02] Wieland Fischer and Jean-Pierre Seifert, *Note on fast computation of secret RSA exponents*, 7th Australasian Conference on Information Security and Privacy — ACISP 2002 (L. Batten and J. Seberry, eds.), Lecture Notes in Computer Science, vol. 2384, Springer-Verlag, 2002, pp. 136–143.

[FSM99] FSTC eCheck Initiative: *full FSML 1.5 reference specification*, 1999, http://www.echeck.org/library/ref/fsml-v1500a.pdf.

[FV03] Pierre-Alain Fouque and Frederic Valette, *The doubling attack — why upwards is better than downwards*, Cryptographic Hardware and Embedded Systems — CHES 2003 (Colin D. Walter, Çetin K. Koç, and Christof Paar, eds.), Lecture Notes in Computer Science, vol. 2779, Springer-Verlag, 2003, pp. 269–280.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel, *Using memory errors to attack a virtual machine*, Proceedings of the IEEE Symposium on Security and Privacy, May 2003, pp. 154–164.

[Gal99] Robert Gallant, *Faster elliptic curve cryptography using efficient endomorphisms*, Presentation at ECC'99, 1999, http://www.cacr.math.uwaterloo.ca/conferences/1999/ecc99/gallant.ps.

[Gar59] Harvey L. Garner, *The residue number system*, IRE Transactions on Electronic Computers **EC-8** (1959), no. 6, 140–147.

[GG99] Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, Cambridge University Press, 1999.

[GK86] Shafi Goldwasser and Joe Kilian, *Almost all primes can be quickly certified*, ACM Symposium on Theory of Computing — STOC '86, ACM Press, 1986, pp. 316–329.

[GK04] Christophe Giraud and Erik W. Knudsen, *Fault attacks on signature schemes*, Information Security and Privacy: 9th Australasian Conference — ACISP 2004 (Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, eds.), Lecture Notes in Computer Science, vol. 3108, Springer-Verlag, 2004, pp. 478–491.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier, *Electromagnetic analysis: Concrete results*, Cryptographic Hardware and Embedded Systems — CHES 2001 (Ç. K. Koç, D. Naccache, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2162, Springer-Verlag, 2001, pp. 251–272.

[Gor98] Daniel M. Gordon, *A survey of fast exponentiation methods*, Journal of Algorithms **27** (1998), 129 – 146, Article No. AL970913.

[HB78] David R. Heath-Brown, *The differences between consecutive primes*, J. London Math. Soc. **18** (1978), no. 2, 7–13.

[HL22] Godfrey H. Hardy and John E. Littlewood, *Some problems of 'Partitio Numerorum' III: On the expression of a number as a sum of primes*, Acta Mathematica, vol. 44, 1922, pp. 1–70.

[HMV04] Darrel Hankerson, Alfred J. Menezes, and Scott A. Vanstone, *Guide to elliptic curve cryptography*, Springer Professional Computing, Springer-Verlag, 2004.

[HP98] Helena Handschuh and Pascal Pailler, *Smart card crypto-coprocessors for public-key cryptography*, Proc. of CARDIS '98, Lecture Notes in Computer Science, vol. 1820, Springer-Verlag, 1998, pp. 372–379.

[HPS99] Helena Handschuh, Pascal Paillier, and Jacques Stern, *Probing attacks on tamper-resistant devices*, Cryptographic Hardware and Embedded Systems — CHES '99 (Ç. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1717, Springer-Verlag, 1999, pp. 303–315.

[Hwa79] Kai Hwang, *Computer arithmetic: principles, architecture and design*, John Wiley & Sons, 1979.

[IEE98] IEEE P1363/D3 (Draft Version 3), *Standard specifications for public key cryptography*, May 1998.

[IEE99] IEEE P1363/D13 (Draft Version 13), *Standard specifications for public key cryptography*, November 1999.

[IPS01] IPSec Working Group: *Additional ECC groups for IKE*, March 2001, Working Group draft, `http://cnscenter.future.co.kr/resource/ietf/wg-draft/draft-ietf-ipsec-ike-ecc-groups-03.txt`.

[ISO02a] International Organization for Standardization, *ISO/IEC 7816-2: Dimensions and location of the contacts*, 2002, `http://www.iso.ch`.

[ISO02b] International Organization for Standardization, *ISO/IEC 7816-3: Electronic signals and transmission protocols*, 2002, `http://www.iso.ch`.

[IT02] Tetsuya Izu and Tsuyoshi Takagi, *A fast parallel elliptic curve multiplication resistant against side channel attacks*, Public Key Cryptography — PKC 2002 (D. Naccache and P. Paillier, eds.), Lecture Notes in Computer Science, vol. 2274, Springer-Verlag, 2002, pp. 280–296.

[IT03] Tetsuya Izu and Tsuyoshi Takagi, *Exceptional procedure attack on elliptic curve cryptosystems*, Public Key Cryptography — PKC 2003 (Y. G. Desmedt, ed.), Lecture Notes in Computer Science, vol. 2567, Springer-Verlag, 2003, pp. 224–239.

[Itk02] Gene Itkis, *Intrusion-resilient signatures: Generic constructions, or defeating strong adversary with minimal assumptions*, Security in Communication Networks — SCN 2002 (S. Cimato, C. Galdi, and G. Persiano, eds.), Lecture Notes in Computer Science, vol. 2576, Springer-Verlag, 2002, pp. 102–118.

[Itk03] Gene Itkis, *Cryptographic tamper evidence*, Proceedings of CCS 2003 (Vijay Atluri and Peng Liu, eds.), ACM SIGSAC, ACM press, Oct 2003, pp. 255–364.

[JBF02] Matthias Jacob, Dan Boneh, and Edward Felten, *Attacking an obfuscated cipher by injecting faults*, Digital Rights Management — DRM 2002, ACM CCS-9 Workshop (J. Feigenbaum, ed.), Lecture Notes in Computer Science, vol. 2696, Springer-Verlag, 2002, pp. 16–31.

[JKQ97] Marc Joye, François Koeune, and Jean-Jacques Quisquater, *Further results on Chinese remaindering*, Tech. Report CG-1997/1, UCL Microelectronics Laboratory — Crypto Group, March 1997, `http://www.dice.ucl.ac.be/crypto/tech_reports/CG1997_1.ps.gz`.

[JLQ99] Marc Joye, Arjen K. Lenstra, and Jean-Jacques Quisquater, *Chinese remaindering cryptosystems in the presence of faults*, Journal of Cryptology **12** (1999), no. 4, 241–245.

[JQ96] Marc Joye and Jean-Jacques Quisquater, *Attacks on systems using Chinese remaindering*, Tech. Report CG-1996/9, UCL Microelectronics Laboratory — Crypto Group, 1996, `http://www.dice.ucl.ac.be/crypto/tech_reports/CG1996_9.ps.gz`.

[JQ97a] Marc Joye and Jean-Jacques Quisquater, *Faulty RSA encryption*, Tech. Report CG-1997/8, UCL Microelectronics Laboratory — Crypto Group, 1997, `http://www.dice.ucl.ac.be/crypto/tech_reports/CG1997_8.ps.gz`.

[JQ97b] Marc Joye and Jean-Jacques Quisquater, *RSA-type signatures in the presence of transient faults*, Tech. Report CG-1997/7, UCL Microelectronics Laboratory — Crypto Group, July 1997, `http://www.dice.ucl.ac.be./crypto/tech_reports/CG1997_7.ps.gz`.

[JQ01] Marc Joye and Jean-Jacques Quisquater, *Hessian elliptic curves and side-channel attacks*, Cryptographic Hardware and Embedded Systems — CHES 2001 (Ç. K. Koç, D. Naccache, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2162, Springer-Verlag, 2001, pp. 412–420.

[JQBD97] Marc Joye, Jean-Jacques Quisquater, Feng Bao, and Robert H. Deng, *RSA-type signatures in the presence of transient faults*, Cryptography and Coding (M. Darnell, ed.), Lecture Notes in Computer Science, vol. 1355, Springer-Verlag, 1997, pp. 155–160.

[JQYY02] Marc Joye, Jean-Jacques Quisquater, Sung-Ming Yen, and Moti Yung, *Observability analysis: Detecting when improved cryptosystems fail*, Topics in Cryptology — CT-RSA 2002 (B. Preneel, ed.), Lecture Notes in Computer Science, vol. 2271, Springer-Verlag, February 2002, pp. 17–29.

[JT01a] Marc Joye and Christophe Tymen, *Compact encoding of non-adjacent forms with applications to elliptic curve cryptography*, Cryptographic Hardware and Embedded Systems — CHES 2001 (Ç. K. Koç, D. Naccache, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2162, Springer-Verlag, 2001, pp. 377–390.

[JT01b] Marc Joye and Christophe Tymen, *Protections against differential analysis for elliptic curve cryptography — an algebraic approach*, Advances in Cryptology — CRYPTO 2001 (Ç. K. Koç, D. Naccache, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2162, Springer-Verlag, 2001, pp. 377–390.

[JY00] Marc Joye and Sung-Ming Yen, *Optimal left-to-right binary signed-digit recoding*, IEEE Transactions on Computers **49** (2000), no. 7, 740–748.

[Kal03] Burt S. Kaliski, Jr., *TWIRL and RSA key size*, 2003, RSA Laboratories Technical Notes, `http://www.rsasecurity.com/rsalabs/node.asp?id=2004`.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun, *Differential power analysis*, Advances in Cryptology — CRYPTO '99 (M. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer-Verlag, 1999, pp. 388–397.

[KK99] Oliver Kömmerling and Markus G. Kuhn, *Design principles for tamper-resistant smartcard processors*, Proceedings of the USENIX Workshop on Smartcard Technology — Smartcard '99, USENIX Association, 1999, pp. 9–20.

[KKT04] Mark Karpovsky, Konrad J. Kulikowski, and Alexander Taubin, *Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard*, International Conference on Dependable Systems and Networks — DSN'04, IEEE Computer Society, 2004, pp. 93–101.

[Kob87] Neil Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation **48** (1987), no. 177, 203–209.

[Kob91] Neil Koblitz, *Constructing elliptic curve cryptosystems in characteristic 2*, Advances in Cryptology — CRYPTO '90 (A. J. Menezes and S. A. Vanstone, eds.), Lecture Notes in Computer Science, vol. 537, Springer-Verlag, 1991, pp. 156–167.

[Kob92] Neil Koblitz, *CM-curves with good cryptographic properties*, Advances in Cryptology — CRYPTO '91, Lecture Notes in Computer Science, vol. 576, Springer-Verlag, 1992, pp. 279–287.

[Koc96a] Osman Kocar, *Hardwaresicherheit von Mikrochips in Chipkarten*, Datenschutz und Datensicherheit **20** (1996), no. 7, 421–424.

[Koc96b] Paul C. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, Advances in Cryptology — CRYPTO '96 (N. Koblitz, ed.), Lecture Notes in Computer Science, vol. 1109, Springer-Verlag, 1996, pp. 104–113.

[Kor93] Israel Koren, *Computer arithmetic algorithms*, Prentice-Hall, 1993.

[KR97] Burton S. Kaliski, Jr. and Matthew J. B. Robshaw, *Comments on some new attacks on cryptographic devices*, Bulletin 5, RSA Laboratories, July 1997, `ftp://ftp.rsasecurity.com/pub/pdfs/bulletn5.pdf`.

[KW03] Chris Karlof and David Wagner, *Hidden markov model cryptanalysis*, Cryptographic Hardware and Embedded Systems — CHES 2003 (C. D. Walter, ed.), Lecture Notes in Computer Science, vol. 2779, Springer-Verlag, 2003, pp. 17–34.

[LD99] Julio López and Ricardo Dahab, *Improved algorithms for elliptic curve arithmetic in GF(2^n)*, Selected Areas in Cryptography — SAC'98 (S. Tavares and H. Meijer, eds.), Lecture Notes in Computer Science, vol. 1556, Springer-Verlag, 1999, pp. 201–212.

[Len87] Hendrik W. Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Mathematics **126** (1987), 649–673.

[Mac61] O. L. MacSorley, *High-speed arithmetic in binary computers*, Proceedings of the IRE **49** (1961), no. 1, 67–91.

[Mil85] Victor S. Miller, *Use of elliptic curves in cryptography*, Advances in Cryptology — CRYPTO '85 (H. C. Williams, ed.), Lecture Notes in Computer Science, vol. 218, Springer-Verlag, 1985, pp. 417–426.

[MO90] François Morain and Jorge Olivos, *Speeding up the computations on an elliptic curve using addition-subtractions chains*, Theoretical Informatics and Applications (1990), no. 24, 531–543.

[Mon85] Peter L. Montgomery, *Modular multiplication without trial division*, Math. Comp. (1985), no. 44, 519–521.

[Mon87] Peter L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation **48** (1987), no. 177, 243–264.

[MR04] Silvio Micali and Leonid Reyzin, *Physically observable cryptography*, Theory of Cryptography Conference — TCC 2004 (M. Naor, ed.), Lecture Notes in Computer Science, vol. 2951, Springer-Verlag, 2004, pp. 278–296.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of applied cryptography*, 5th (2001) ed., CRC Press, October 1996, `http://www.cacr.math.uwaterloo.ca/hac/`.

[NES03] *NESSIE consortium (`http://www.cryptonessie.org`): Portfolio of recommended cryptographic primitives*, 27 February 2003, `https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/decision-final.pdf`.

[NM96] David Naccache and David M'Raïhi, *Cryptographic smart cards*, IEEE Micro **16** (1996), no. 3, 14–24.

[Nor96] Eugene Normand, *Single event upset at ground level*, IEEE Transactions on Nuclear Science **43** (1996), no. 6, 2742–2750.

[OH03] Katsuyuki Okeya and Dong-Guk Han, *Side channel attack on Ha-Moon's countermeasure of randomized signed scalar multiplication*, Progress in Cryptology — INDOCRYPT 2003 (T. Johansson and S. Maitra, eds.), Lecture Notes in Computer Science, vol. 2904, Springer-Verlag, 2003, pp. 334–348.

[ORT+96] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh, *Field testing for cosmic ray soft errors in semiconductor memories*, IBM Journal of Research and Development **40** (1996), no. 1, 41–50.

[OS00] Katsuyuki Okeya and Kouichi Sakurai, *Power analysis breaks elliptic curve cryptosystems even secure against the timing attack*, Progress in Cryptology — INDOCRYPT 2000 (B. Roy and E. Okamoto, eds.), Lecture Notes in Computer Science, vol. 1977, Springer-Verlag, 2000, pp. 178–190.

[OT04] Katsuyuki Okeya and Tsuyoshi Takagi, *SCA-resistant and fast elliptic scalar multiplication based on wNAF*, IEICE Trans. Fundamentals **E87-A** (2004), no. 1, 75–84.

[Ott01] Martin Otto, *Brauer addition–subtraction chains*, Diplomarbeit, Universität–Gesamthochschule Paderborn, 2001.

[Pag02] Dan Page, *Theoretical use of cache memory as a cryptanalytic side-channel*, Cryptology ePrint Archive, Report 2002/169, 2002, `http://eprint.iacr.org/2002/169/`.

[Pet97] Ivars Peterson, *Chinks in digital armor – exploiting faults to break smart-card cryptosystems*, Science News **151** (1997), no. 5, 78–79, `http://www.sciencenews.org/sn_arc97/2_1_97/bob1.htm`.

[PH78] Stephen C. Pohlig and Martin Hellman, *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*, IEEE Transactions on Information Theory **IT-24** (1978), no. 1, 106–110.

[PKC02] RSA Security Inc., *Public-Key Cryptography Standards (PKCS): PKCS #1 v2.1*, 14 Jun 2002, `http://www.rsasecurity.com/rsalabs/node.asp?id=2124`.

[PQ03] Gilles Piret and Jean-Jacques Quisquater, *A differential fault attack technique against SPN structures, with application to the AES and* KHAZAD, Cryptographic Hardware and Embedded Systems — CHES 2003 (Colin D. Walter, Çetin K. Koç, and Christof Paar, eds.), Lecture Notes in Computer Science, vol. 2779, Springer-Verlag, 2003, pp. 77–88.

[PV04] Dan Page and Frederik Vercauteren, *Fault and side-channel attacks on pairing based cryptography*, Cryptology ePrint Archive, Report 2004/283, 2004, `http://eprint.iacr.org/2004/283/`.

[QS01] Jean-Jacques Quisquater and David Samyde, *Electromagnetic analysis (EMA) measures and countermeasures for smart cards*, E-Smart Smartcard Programming and Security (I. Attali and T. Jensen, eds.), Lecture Notes in Computer Science, vol. 2140, Springer-Verlag, 2001, pp. 200–210.

[QS02] Jean-Jacques Quisquater and David Samyde, *Eddy current for magnetic analysis with active sensor*, Proceedings of E-Smart 2002, September 2002.

[RE00] Wolfgang Rankl and Wolfgang Effing, *Smart card handbook*, 2nd ed., John Wiley & Sons, 2000.

[Rei60] George W. Reitwiesner, *Binary arithmetic*, Advances in Computers (1960), no. 1, 231–308.

[RFC98a] The Internet Society's Network Working Group: *The internet key exchange (IKE)*, November 1998, `http://www.faqs.org/rfcs/rfc2409.html`.

[RFC98b] The Internet Society's Network Working Group: *The OAKLEY key determination protocol*, November 1998, `http://www.faqs.org/ftp/rfc/pdf/rfc2412.txt.pdf`.

[RR01] Josyula R. Rao and Pankaj Rohatgi, *Empowering side-channel attacks*, Cryptology ePrint Archive, Report 2001/037, 2001, `http://eprint.iacr.org/2001/037/`.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21** (1978), no. 2, 120–126.

[SA02] Sergei Skorobogatov and Ross Anderson, *Optical fault induction attacks*, Cryptographic Hardware and Embedded Systems — CHES 2002 (B. S. Kaliski, Jr., Ç. K. Koç, and C. Paar, eds.), Lecture Notes in Computer Science, vol. 2523, Springer-Verlag, 2002, pp. 2–12.

[Sch95] Rene Schoof, *Counting points on elliptic curves over finite fields*, J. Théorie des Nombres de Bordeaux (1995), no. 7, 219–254.

[Sch96] Claus Peter Schnorr, *Security of 2^t-root identification and signatures*, Advances in Cryptology — CRYPTO '96 (N. Koblitz, ed.), Lecture Notes in Computer Science, vol. 1109, Springer-Verlag, 1996, pp. 143–156.

[Sch00a] Werner Schindler, *A timing attack against RSA with the Chinese remainder theorem*, Cryptographic Hardware and Embedded Systems — CHES 2000 (Ç. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1965, Springer-Verlag, 2000, pp. 109–124.

[Sch00b] Bruce Schneier, *Secrets & lies — digital security in a networked world*, Wiley Computer Publishing, 2000.

[SEC00] Standards for Efficient Cryptography Group (SECG), *SEC 2: Recommended elliptic curve domain parameters*, 2000, `http://www.secg.org/collateral/sec2_final.pdf`.

[Sed87] Holger Sedlak, *The RSA cryptography processor*, Advances in Cryptology — EUROCRYPT '87 (D. Chaum and W. L. Price, eds.), Lecture Notes in Computer Science, vol. 304, Springer-Verlag, 1987, pp. 95–108.

[Sed89] Holger Sedlak, *Cryptographic method and cryptographic processor for carrying out the method*, 1989, US Patent No. 4,870,681.

[SET02] The SETI@home project, *Current total statistics*, June 28th 2002, `http://setiathome.ssl.berkeley.edu/totals.html`.

[Sha95] Adi Shamir, *RSA for paranoids*, CryptoBytes (the technical newsletter of the RSA laboratories) **1** (1995), no. 3, pp. 1,3, and 4, `http://www.rsa.com/rsalabs/cryptobytes/`.

[Sha99] Adi Shamir, *Method and apparatus for protecting public key schemes from timing and fault attacks*, 1999, US Patent No. 5,991,415, Nov. 23, 1999.

[Sig01a] *Gesetz zur digitalen Signatur (Artikel 3 des Gesetzes zur Regelung der Rahmenbedingungen für Informations- und Kommunikationsdienste)*, published in BGBl I 2001, 876 (16 May 2001), 2001, `http://bundesrecht.juris.de/bundesrecht/sigg_2001/index.html`.

[Sig01b] *Verordnung zur elektronischen Signatur*, published in BGBl I 2001, 3074 (16 Nov 2001), 2001, `http://bundesrecht.juris.de/bundesrecht/sigv_2001/`.

[Sil00] Joseph H. Silverman, *The arithmetic of elliptic curves*, Graduate Texts in Mathematics, vol. 109, Springer-Verlag, 2000.

[Sma04] Nigel Smart, *Protocols I*, 2004, Talk given at the ECC Summer School held in conjunction with ECC 2004, September 13 – 17, 2004, Bochum, Germany.

[ST04] Adi Shamir and Eran Tromer, *Acoustic cryptanalysis — on nosy people and noisy machines*, presented at the Rump session of EUROCRYPT 2004, 2004, `http://www.wisdom.weizmann.ac.il/~tromer/acoustic/`.

[Wag04] David Wagner, *Cryptanalysis of a provably secure CRT-RSA algorithm*, Conference on Computer and Communications Security — CCS 2004, ACM SIGSAC, ACM Press, 2004, pp. 92–97.

[Wal03] Colin D. Walter, *Security constraints on the Oswald-Aigner exponentiation algorithm*, Cryptology ePrint Archive, Report 2003/013, 2003, `http://eprint.iacr.org/2003/013/`.

[WAP01] Wireless Application Protocol Forum: *Wireless transport layer security specification*, 2001, `http://www.wmlclub.com/docs/especwap2.0/WAP-261-WTLS-20010406-a.pdf`.

[Was03] Lawrence C. Washington, *Elliptic curves — number theory and cryptography*, Discrete Mathematics and Its Applications, Chapman & Hall/CRC, 2003.

[Wei00] Steve H. Weingart, *Physical security devices for computer subsystems: A survey of attacks and defences*, Advances in Cryptology — CRYPTO 2000 (Ç. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1965, Springer-Verlag, 2000, pp. 302–317.

[WQ90] Dominique de Waleffe and Jean-Jacques Quisquater, *CORSAIR, a smart card for public-key cryptosystems*, Advances in Cryptology — CRYPTO '90, Lecture Notes in Computer Science, vol. 537, Springer-Verlag, 1990, pp. 503–513.

[YJ00] Sung-Ming Yen and Marc Joye, *Checking before output may not be enough against fault-based cryptanalysis*, IEEE Transactions on Computers **49** (2000), no. 9, 967–970.

[YKLM01a] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sangjae Moon, *A countermeasure against one physical cryptanalysis may benefit another attack*, Information Security and Cryptology — ICISC 2001 (K. Kim, ed.), Lecture Notes in Computer Science, vol. 2288, Springer-Verlag, 2001, pp. 414–427.

[YKLM01b] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sangjae Moon, *RSA speedup with residue number system immune against hardware fault cryptanalysis*, Information Security and Cryptology — ICISC 2001 (K. Kim, ed.), Lecture Notes in Computer Science, vol. 2288, Springer-Verlag, 2001, (a journal version has been published as [YKLM03]), pp. 397–413.

[YKLM03] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sangjae Moon, *RSA speedup with chinese remainder theorem immune against hardware fault cryptanalysis*, IEEE Transactions on Computers **52** (2003), no. 4, 461–472.

[YMH03] Sung-Ming Yen, Sangjae Moon, and Jae-Cheol Ha, *Hardware fault attack on RSA with CRT revisited*, Information Security and Cryptology — ICISC 2002 (P. J. Lee and C. H. Lim, eds.), Lecture Notes in Computer Science, vol. 2587, Springer-Verlag, 2003, pp. 374–388.

[YWK04] Bo Yang, Kaijie Wu, and Ramesh Karri, *Scan based side channel attack on data encryption standard*, Cryptology ePrint Archive, Report 2004/083, 2004, `http://eprint.iacr.org/2004/083/`.

[ZCM+96] J. F. Ziegler, H. W. Curtis, F. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus, *IBM experiments in soft fails in computer electronics (1978-1994)*, IBM Journal of Research and Development **40** (1996), no. 1, 3–18.

[ZM96] Yuliang Zheng and Tsutomu Matsumoto, *Breaking smart card implementations of elgamal signature and its variants*, 1996, Presented at the Rump Session of ASIACRYPT '96.

[ZM97] Yuliang Zheng and Tsutomu Matsumoto, *Breaking real-world implementations of cryptosystems by manipulating their random number generation*, Proceedings of the 1997 Symposium on Cryptography and Information Security (Fukuoka, Japan), January 29 – February 1 1997.

Symbols, Acronyms and Notation

$\stackrel{\lightning}{\longmapsto}$	indicates a mapping according to some fault model
$\in_R$	indicates a random choice i.i.d. according to some distribution
$<P>$	denotes the cyclic group generated by the element P
$\#X$	the size of the set or sequence X or the order of the group X (number of elements)
$\circ$	denotes concatenation of bit sequences
$a \mid b$	indicates that a divides b
$a \nmid b$	indicates that a does not divide b
$\tilde{x}$	denotes a faulty value of the variable x
ALU	Arithmetic Logical Unit
base field	referring to the finite field used to define an elliptic curve
BEU	Bus Encryption Unit
binary field	any finite field $\mathbb{F}_{2^t}$ with characteristic 2
CPU	Central Processing Unit
ECDDH	Elliptic Curve Decisional Diffie-Hellman Problem
ECDHP	Elliptic Curve (Computational) Diffie-Hellman Problem
ECDLP	Elliptic Curve Discrete Logarithm Problem
ϵ	the empty word/bit string
$\mathbb{F}_q$	same as $GF(q)$, the Galois Field or Finite Field with q elements
$\varphi(N)$	Euler's totient function of N, i.e., the number of coprime numbers $1 \leq i < N$.
$\mathcal{G}_i(\tilde{S})$	set of guesses (see Definition 2.29)
$\mathcal{G}_i^{(1)}(\tilde{S})$	set of guesses surviving Test 1 (see Definition 2.30)
$\mathcal{G}_i^{(2)}(\tilde{S})$	set of guesses surviving Test 2 (see Definition 2.34)
gcd	greatest common divisor
$h_i(d)$	high part of an integer d (see Definitions 2.17 and 5.4)
$H_i(d)$	same as $h_i(d) \cdot P$ (see Definition 5.4)
i.i.d.	independent identically distributed
$\mathbb{K}$	indicating an arbitrary field
$l(n)$	the binary length of the integer n
$l_i(d)$	low part of an integer d (see Definitions 2.17 and 5.4)
$L_i(d)$	same as $l_i(d) \cdot P$ (see Definition 5.4)
lcm	least common multiple
LSB	least significant bit
MSB	most significant bit
$\mathbb{N}$	the set of all non-negative integers, i.e., including 0
$\mathcal{NAF}$	the non-adjacent form of an integer
prime field	any finite field $\mathbb{F}_p$, where p is prime
$\mathbb{Q}$	the set of all rational numbers or fractions of integers

RSA	cryptosystem proposed by Rivest, Shamir, and Adleman in [RSA78]
$\mathcal{S}$	a set containing faulty outputs of an attacked algorithm
w.l.o.g.	without loss of generality
$\mathbb{Z}$	the set of all integers (positive and negative numbers)
$\mathbb{Z}_N$	the set of all integers modulo N, i.e., $\mathbb{Z}/N\mathbb{Z} = \{0, 1, \ldots, N-1\}$