Efficiency and Implementation Security of Code-based Cryptosystems

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zu Erlangung des Grades
Doctor rerum naturalium (Dr. rer. nat.)
von

Dipl.-Phys. Falko Strenzke

geboren in Wolfenbüttel.

Referenten:	Prof. Dr. Johannes Buchmann
	Prof. Dr.-Ing. Christof Paar
Tag der Einreichung:	28.08.2013
Tag der mündlichen Prüfung:	12.11.2013
Hochschulkennziffer:	D17

Darmstadt 2013

Abstract

This thesis studies efficiency and security problems of implementations of code-based cryptosystems. These cryptosystems, though not currently used in the field, are of great scientific interest, since no quantum algorithm is known that breaks them essentially faster than any known classical algorithm. This qualifies them as cryptographic schemes for the quantum-computer era, where the currently used cryptographic schemes are rendered insecure.

Concerning the efficiency of these schemes, we propose a solution for the handling of the public keys, which are, compared to the currently used schemes, of an enormous size. Here, the focus lies on resource-constrained devices, which are not capable of storing a code-based public key of communication partner in their volatile memory. Furthermore, we show a solution for the decryption without the parity check matrix with a passable speed penalty. This is also of great importance, since this matrix is of a size that is comparable to that of the public key. Thus, the employment of this matrix on memory-constrained devices is not possible or incurs a large cost.

Subsequently, we present an analysis of improvements to the generally most time-consuming part of the decryption operation, which is the determination of the roots of the error locator polynomial. We compare a number of known algorithmic variants and new combinations thereof in terms of running time and memory demands. Though the speed of pure software implementations must be seen as one of the strong sides of code-based schemes, the optimisation of their running time on resource-constrained devices and servers is of great relevance.

The second essential part of the thesis studies the side channel security of these schemes. A side channel vulnerability is given when an attacker is able to retrieve information about the secrets involved in a cryptographic operation by measuring physical quantities such as the running time or the power consumption during that operation. Specifically, we consider attacks on the decryption operation, which either target the message or the secret key. In most cases, concrete countermeasures are proposed and evaluated. In this context, we show a number of timing vulnerabilities that are linked to the algorithmic variants for the root-finding of the error locator polynomial mentioned above. Furthermore, we show a timing attack against a vulnerability in the Extended Euclidean Algorithm that is used to solve the so-called key equation during the decryption operation, which aims at the recovery of the message. We also present a related practical power analysis attack. Concluding, we present a practical timing attack that targets the secret key, which is based on the combination of three vulnerabilities, located within the syndrome inversion, a further suboperation of the decryption, and the already mentioned solving of the key equation.

We compare the attacks that aim at the recovery of the message with the analogous attacks against the RSA cryptosystem and derive a general methodology for the discovery of the underlying vulnerabilities in cryptosystems with specific properties.

Furthermore, we present two implementations of the code-based McEliece cryptosystem: a smart card implementation and flexible implementation, which is based on a previous open-source implementation. The previously existing open-source implemen-

tation was extended to be platform independent and optimised for resource-constrained devices. In addition, we added all algorithmic variants presented in this thesis, and we present all relevant performance data such as running time, code size and memory consumption for these variants on an embedded platform. Moreover, we implemented all side channel countermeasures developed in this work.

Concluding, we present open research questions, which will become relevant once efficient and secure implementations of code-based cryptosystems are evaluated by the industry for an actual application.

Zusammenfassung

Die vorliegende Arbeit befasst sich mit Effizienz- und Sicherheitsproblemen bei der Implementierung von Code-basierten Public-Key Verschlüsselungsverfahren. Diese Verfahren, obwohl derzeit nicht in Verwendung, sind von großem wissenschaftlichen Interesse, da kein Quantenalgorithmus bekannt ist, der diese entscheidend effizienter brechen kann als ein klassischer Algorithmus. Dies macht sie zu Kandidaten für die Quantencomputerära, in der die heute verwendeten Public-Key Verschlüsselungsverfahren unsicher werden.

Bezüglich der Effizienz wird eine Lösung für die Handhabung der bei dieser Art von Verfahren im Vergleich mit den heute im Einsatz befindlichen extrem großen öffentlichen Schlüsseln vorgestellt. Dabei liegt der Fokus auf ressourcenbeschränkten Geräten, die nicht in der Lage sind, den öffentlichen Schlüssel eines Kommunikationspartners im flüchtigen Speicher zu halten. Ferner wird eine Lösung aufgezeigt, mit der die Entschlüsselung auch ohne die Parity Check Matrix mit vertretbaren Geschwindigkeitseinbußen möglich ist. Dies ist ebenso von großer Bedeutung, da diese Matrix von vergleichbarer Größe ist wie der öffentliche Schlüssel, und somit eine Verwendung derselben auf speicherarmen Geräten nicht möglich oder mit hohen Kosten verbunden ist.

Es wird auch eine Untersuchung zur Verbesserung der Laufzeit der im Allgemeinen rechenintensivsten Teiloperation bei der Entschlüsselung, dem Finden der Nullstellen des Fehlerlokatorpolynoms, vorgestellt. Hierbei werden verschiedene bekannte algorithmische Varianten und neue Kombinationen derselben in Bezug auf Laufzeit und Speicheranforderungen verglichen. Obwohl die Geschwindigkeit reiner Softwareimplementierungen gerade als eine der Stärken der Code-basierten Verfahren angesehen werden muss, ist die Optimierung der Laufzeit sowohl auf ressourcenbeschränkten Plattformen als auch auf Servern von großer Relevanz.

Der zweite wesentliche Teil der Arbeit befasst sich mit der Seitenkanalsicherheit dieser Verfahren. Seitenkanalschwachstellen bedeuten, dass ein Angreifer während einer kryptographischen Operation durch die Messung physikalischer Größen wie der Laufzeit der Berechnung oder der Leistungsaufnahme des Geräts während der Operation Informationen über an der Berechnung beteiligte geheime Informationen erhält. Dabei werden Angriffe auf die Entschlüsselungsoperation betrachtet, die entweder auf die Rekonstruktion der Nachricht oder des geheimen Schlüssels abzielen, und in den meisten Fällen konkrete Gegenmaßnahmen vorgeschlagen und evaluiert. In diesem Zusammenhang werden eine Reihe von Laufzeitschwachstellen aufgezeigt, welche mit manchen der bereits oben erwähnten algorithmischen Varianten für das Finden der Nullstellen des Fehlerlokatorpolynoms zusammenhängen. Des Weiteren wird ein Laufzeitangriff gegen eine Schwachstelle des bei der Entschlüsselung verwendeten erweiterten Euklidischen Algorithmus zur Lösung der sogenannten Key Equation vorgestellt, der auf die Rekonstruktion der Nachricht abzielt. Hierzu wird auch eine praktische Leistungsaufnahmeattacke vorgeführt. Schließlich wird eine praktische Laufzeitattacke zur Rekonstruktion des geheimen Schlüssels vorgestellt, die auf der Kombination von drei Schwachstellen innerhalb der Syndrominvertierung, einer weiteren Teiloperation der Entschlüsselung, und der schon erwähnten Lösung der Key Equation beruht.

Die Angriffe, die die Rekonstruktion der Nachricht zum Ziel haben, werden mit ihrem Analogon für das RSA Public Key Verschlüsselungsverfahren verglichen, und es wird eine allgemeine Methodologie für das Auffinden der in solchen Fällen zugrunde liegenden Schwachstellen entwickelt.

Ferner werden zwei Implementierungen des Code-basierten McEliece Verschlüsselungsverfahrens vorgestellt: Eine Smartcard-Implementierung und eine auf einer quelloffenen PC Implementierung basierende flexible Implementierung. Die bestehende quelloffene Implementierung wurde dahingehend erweitert, dass sie plattformunabhänig wurde und für ressourcenbeschränkte Geräte optimiert wurde. Außerdem wurden dort alle in dieser Arbeit vorgestellten algorithmischen Varianten integriert, für die jeweils alle relevanten Performanzdaten wie Laufzeit, Code-Größe und Speicherverbrauch auf einem eingebetteten System vorgestellt werden. Ferner wurden die erarbeiteten Seitenkanalgegenmaßnahmen implementiert.

Zuletzt werden in dieser Arbeit offen bleibende Forschungsfragen vorgestellt, die Relevanz erhalten werden, sobald effiziente und sichere Implementierungen Code-basierter Verfahren von der Industrie für den Einsatz evaluiert werden.

1. Introduction **1**

2. Preliminaries **7**
2.1. The Patterson Algorithm for the Decoding of Goppa Codes 7
2.2. The McEliece PKC 8
2.3. The Niederreiter PKC 8

3. Optimization for Resource-constrained Devices **11**
3.1. On-Line public Operation for Code-based Schemes 11
3.1.1. The Storage Problem on Memory Constrained Devices 12
3.1.2. Public Key Infrastructures 12
3.1.3. Description of the On-line Public Operation 13
3.1.4. Transmission Rates 15
3.1.5. Example Implementation 16
3.1.6. Non-interactive Version of the Protocol 17
3.1.7. Simulation of higher Transmission Rates 17
3.1.8. Experimental Results 18
3.1.9. Column-wise vs. Row-wise Matrix-Vector Multiplication 18
3.1.10. Code-based Signature Schemes 20
3.2. McEliece Decryption without the Parity Check Matrix 20
3.2.1. Optimized Algorithm for the Syndrome Computation without the Parity Check Matrix 21
3.2.2. Implementation and Performance Results 22
3.3. Efficient Root-Finding during the Decryption 23
3.3.1. Remarks about the $\mathbb{F}_{2^m}$ Operations 23
3.3.2. Variants of Root Finding 24
3.3.2.1. Exhaustive Evaluation with and without Division 24
3.3.2.2. Berlekamp Trace Algorithm 25
3.3.2.3. Root Finding with linearised Polynomials 26
3.3.2.4. New Hybrid Variants 27
3.3.3. Performance of the Root-finding Variants 28
3.4. Comparison of the McEliece and Niederreiter PKCs in Terms of Efficiency 30
3.4.1. Public Key Size and Encryption Speed 30
3.4.2. Private Key Size and Decryption Speed 30
3.4.3. Message and Ciphertext Sizes 31

4. Side Channel Security 33
4.1. Message-aimed Side Channel Attacks against the Decryption Operation . 33
4.1.1. Timing Vulnerabilities in the Root-Finding based on the Degree of the Error Locator Polynomial 33
4.1.2. Timing Vulnerability of the Key Equation solving EEA and Countermeasures 35
4.1.2.1. Identification of the Vulnerability 36
4.1.2.2. Timing Countermeasure 38
4.1.2.3. Implementation and Verification of the Countermeasure . 38
4.1.3. A related Simple Power Analysis Attack against the Key Equation Solving EEA 40
4.1.3.1. Measurement Setup 40
4.1.3.2. Attacks against the insecure Implementation 41
4.1.3.3. Countermeasure 41
4.1.4. Vulnerability in Root-Finding with exhaustive Evaluation and Division 42
4.2. Side Channel Attacks against the secret Support 44
4.2.1. Timing Attacks against the EEA 46
4.2.1.1. Properties of the Syndrome Inversion 46
4.2.1.2. Linear Equations from $w = 4$ Error Vectors 47
4.2.1.3. Cubic Equations from $w = 6$ Error Vectors 50
4.2.1.4. Enlargement of the Timing Differences by the Key Equation Solving EEA 51
4.2.1.5. The Zero Element of the Support from $w = 1$ Error Vectors 52
4.2.1.6. Combining the "$w = 1$", "$w = 4$", and "$w = 6$" Vulnerabilities to a practical Attack 53
4.2.1.7. Experimental Results 57
4.2.1.8. Effect of Countermeasures against other Attacks 60
4.2.1.9. Possible Extensions of the Attack 61
4.2.1.10. The Problem of Countermeasures 62
4.2.2. Timing Attacks against Root-Finding Algorithms 62
4.2.2.1. Vulnerability of *eval-div-rf* 62
4.2.2.2. Vulnerability of *dcmp-rf* 64
4.3. Fault Attacks 66
4.3.1. Fault Attack Vulnerability revealing the Degree of the Error Locator Polynomial 66
4.3.2. Fault Attack Vulnerability revealing Information about the Number of Roots of the Error Locator Polynomial 66
4.4. Transferability of the Vulnerabilities and Countermeasures to the Niederreiter PKC 67

4.5. Relation of the Side Channel Vulnerabilities to those of other Cryptosystems 67
4.5.1. Message-aimed Side Channel Attacks against Cryptosystems with homomorphic Properties 67
4.5.1.1. Manger's Attack against RSA-OAEP 68
4.5.1.2. Homomorphic Properties of RSA and the McEliece Cryptosystem 70
4.5.1.3. Comparison of Message-aimed Side Channel Attacks against RSA and McEliece 70
4.5.1.4. Methodology for the Analysis of public Key Cryptosystems with homomorphic Properties 72
4.5.2. Blinding Countermeasures for Code-based Cryptosystems 73

5. Embedded Implementations of the McEliece PKC 75
5.1. A Flexible Platform independent Implementation of the McEliece PKC . 75
5.1.1. Description of the Implementation 75
5.1.2. Performance Results 75
5.2. A Smart Card Implementation of the McEliece PKC 79
5.2.1. Description of the Implementation 79
5.2.2. Performance Results 81

6. Open Problems 83
6.1. Potential Cache-Timing Vulnerabilities in Code-Based Decryption Operations 83
6.2. Countermeasures Against the Low-Weight Error Vector Attacks 83
6.3. Side Channel Security of *BTA-rf* 84
6.4. Side-Channel Secure Implementation of *dcmp-div-rf* 86
6.5. The Problem of the Optimal Root-Finding Algorithm for Embedded Implementations with Hardware Support 88

7. Conclusion 89

A. Appendix 97
A.1. Cubic Equations involving less than four Basis Elements are impossible . 97

The Author's Publications

The following is a list of the author's publications that build the basis of this thesis.

[1] Strenzke, F., Tews, E., Molter, H., Overbeck, R., Shoufan, A.: Side Channels in the McEliece PKC. In: The third international Workshop on Post-Quantum Cryptography, PQC 2008. Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008)

[2] Shoufan, A., Strenzke, F., Molter, H., Stöttinger, M.: A Timing Attack against Patterson Algorithm in the McEliece PKC. In: Information, Security and Cryptology, ICISC 2009. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2009)

[3] Strenzke, F.: A Timing Attack against the secret Permutation in the McEliece PKC. In: The third international Workshop on Post-Quantum Cryptography, PQC 2010. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2010)

[4] Strenzke, F.: A Smart Card Implementation of the McEliece PKC. In: Workshop in Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices, WISTP 2010. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2010)

[5] Strenzke, F.: Message-aimed Side Channel and Fault Attacks against Public Key Cryptosystems with homomorphic Properties. In: Journal of cryptographic Engineering (2011)

[6] Molter, H.G., Stötinger, M., Shoufan, A., Strenzke, F.: A Simple Power Analysis Attack on a McEliece Cryptoprocessor. In: Journal of Cryptographic Engineering (2011)

[7] Strenzke, F.: Fast and Secure Root-Finding for Code-based Cryptosystems. In: The 11th International Conference on Cryptology and Network Security, CANS 2012. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2012)

[8] Strenzke, F.: Solutions for the Storage Problem of McEliece Public and Private Keys on Memory-constrained Platforms. In: Proceedings of the 15th international conference on Information Security, ISC 2012. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2012)

[9] Strenzke, F.: Timing Attacks against the Syndrome Inversion in Code-based Cryptosystems. In: The fifth international Workshop on Post-Quantum Cryptography, PQC 2013. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2013)

1. Introduction

In the past, the science of cryptography has been known to be crucial to military and confidential governmental communication security. With the rise of the internet the role cryptography plays has changed significantly. Today, in the whole world, computer users rely on cryptographic security mechanisms when they access the internet and use secure login to mail servers and web sessions secured by SSL/TLS. Furthermore, information technology becomes more and more pervasive as countless types of machines and devices like cars, smartphones, and control units for infrastructure components are connected to the internet for numerous types of services. For instance, many of these devices rely on public key cryptography to authenticate software updates. Accordingly, in the past years, the media coverage of information technology security incidents has increased, as it becomes more and more obvious that almost all citizens are potentially affected when bank account data and other confidential information is leaked to malicious parties or critical infrastructures such as power plants are threatened by sabotage that is possible by exploiting software vulnerabilities.

However, none of the past incidents have shaken the pillars of information security, which are formed by the public key primitives that provide authenticity and confidentiality in message exchange: these are mainly RSA as a signature and encryption scheme, Diffie-Hellmann as a key exchange scheme, Elgamal as an encryption scheme, and DSA and ECDSA as signature schemes. Though the growth of computational power through technological advances today demands the use of larger key sizes than at the time of the proposal of the these schemes, their security in principle is still unquestioned.

But this situation is threatened by a new technology: quantum information technology. This technology is founded on the science of quantum information theory, which describes the information processing that is possible when the information carriers follow the laws of quantum mechanics. While in principle these laws apply to all matter, it demands special conditions for these effects to be actually observed. A computer that works under these conditions is referred to as a quantum computer. In 1997, Peter W. Shor published a quantum algorithm, i.e. an algorithm that runs on a quantum computer but not on a classical computer, which is able to break all schemes based on the factorization and discrete logarithm problems [10]. Unfortunately, this encompasses all the schemes listed above as the current pillars for information security.

However, today, no quantum computer can be build that is actually able carry out these algorithms in order to break keys of realistic sizes. Currently, scientists are still struggling for the breakthrough in quantum information technology that enables them to realise useful quantum computers. No one can tell whether or when this breakthrough is going to happen. The only the thing that is clear is that once it happens, our pillars of information security will fall. As the dependency of the technologically highly developed

societies on secure information technology is ceaselessly growing, the impact of this milestone event can hardly be overestimated. Furthermore, it must be assumed that certain parties involved in quantum technology research, like for instance secret agencies, will not necessarily inform the public about their progress in this field. Accordingly, it can be assumed that it is only a question of time until this topic will receive greater awareness by authorities and the public; and the pending problems and uncertainties will have to be answered by concrete measures. The final goal is of course to find public key schemes that are resistant to quantum algorithms, and thus can be used to replace the current ones. Such schemes are commonly referred to as post-quantum cryptographic schemes.

Concerning digital signatures, which are used to provide a proof that a certain message originates from a specific sender, and encryption, the purpose of which is to ensure the confidentiality of messages, there exists a large difference concerning the impact of the realisation of quantum computers: assuming that a document was signed using an algorithm which must be feared to be broken within a short time, it is possible to renew the signature using another algorithm that is still regarded as secure. For encryption schemes, there is no similar escape: any message encrypted today and sent over an unsecured channel can be intercepted and stored, and decrypted once it is technologically possible to break the employed encryption algorithm.

For this reason, it is especially important to find public key encryption schemes, also referred to as public key cryptosystems (PKCs), that can be regarded as secure in the presence of quantum computers and can be used as a replacement for the currently used PKCs. As a consequence, we have chosen the McEliece [11] and Niederreiter [12] PKCs as the topic of our thesis. The McEliece scheme has been proposed already in 1978. But due to its large public key size, it has almost never experienced any use in real world applications and not received much interest from researchers. The Niederreiter scheme was proposed a few years later but shares all basic properties with the McEliece scheme. Both schemes are built on error correcting codes and are considered immune to quantum computer attacks [13], and thus, are of great interest as candidates for post-quantum encryption schemes. Accordingly, since recently the post-quantum question has experienced increased attention, they have received growing interest from researchers in the past years and been analysed with respect to efficiency on various platforms [14, 15, 16, 17]. Furthermore, some researchers have investigated the side channel security of code-based cryptosystems [18, 19].

Concerning the efficiency of code-based cryptosystems, as for any cryptographic scheme, the two main aspects are the running times of the operations and the memory demands.

The running times of the encryption and decryption operation are the strong side of these schemes. Nevertheless, there are certain applications where maximal optimization of the running time of software implementations is of great interest: the first is the implementation of cryptosystems for servers. Today, the server-side implementation of SSL/TLS in large-scale services is known to be a big cost problem. Thus, improvements in the running time always are of great importance for this purpose. The second application of cryptosystems where reduction of running times is of major relevance is given in the context of resource constrained platforms such as smart cards. On such platforms,

the currently employed public key schemes such as RSA or ECDSA can only be run in passable time if hardware support is present. Accordingly, it is of great interest to know what running times can be achieved on such platforms by pure software implementations of code-based schemes. This helps to estimate the cost for developing platforms that support code-based cryptosystems and could replace today's RSA and elliptic curve centered smart cards.

The other great performance aspect, that of memory requirements, is per se the greater challenge for code-based cryptosystems. This is primarily due to the large public key size that characterizes these schemes. Due to the fact that communication partners usually vary considerably in public key infrastructure scenarios, it would be the best choice to store the public keys in the RAM. However, the amount of RAM typically available on devices such as smart cards or other low cost microcontrollers is far too small to hold these keys. Accordingly, one would have to store them in non-volatile memory, which comes at the disadvantages of the slower writing speed and the smaller number of write cycles in the memory module's lifetime, not to mention the cost of keeping available such an amount of memory. But also the private key size as it is found in the implementation choices made in recent publications of the McEliece scheme poses a big cost problem. Many low cost embedded microcontrollers would be incapable of storing such a large McEliece private key at all.

In this thesis, we address both of these fundamental performance aspects. Concerning the running times, we turn to the most time-consuming part of the code-based decryption operation, which is the finding of the roots of the error locator polynomial. We analyse a number of different previously published variants for this computational task. Some of these algorithms have already been employed in implementations of code-based cryptosystems presented in recent publications. Others have so far only been proposed for error correcting codes in their genuine field of application. We also introduce new hybrid algorithms. We find that considerable differences in the performance exist between the basic variants and that the hybrid algorithms allow a further improvement in the running time.

Furthermore, we address the challenge of the huge public key size of these schemes, which amounts to at least 100 KB for code parameters that provide reasonable security. Specifically, we turn to the scenario where a resource constrained device needs to perform the encryption operation using the public key of a communication partner. In contrast to previous works, where the public key is stored on the device [4, 15, 16], we propose a different approach: the public key is processed on-line as it is input into the encrypting device. This requires only a small amount of RAM and removes the need to reserve a large portion of non-volatile memory. As a result, we find that even current embedded CPUs allow for a computation speed that is not matched by the technologically possible transmission rates available in the context of smart cards. This means that a solution that gives a better performance in this scenario makes it necessary to improve the transmission rates of the smart card interface.

The private key size of both the McEliece and the Niederreiter scheme is in the same range as that of the public keys. However, for the McEliece scheme, this is an implementation choice: the large size stems from the parity check matrix, which is used to speed

up decryptions in all previously published implementations of the scheme [4, 16, 17, 20], but it is not an essential part of the private key. Because the storage of a large private key of more than 100 kB poses a significant burden for resource constrained systems such as smart cards, we explore the impact on the running time if the decryption is performed without this matrix. To this end, we develop an optimized algorithm for this computational task. In an example implementation on an embedded CPU we show that, for code parameters with a security level of about 120 bit, the decryption without the parity check matrix is about half as fast as if the parity check matrix was used.

The second major part of this thesis addresses the side channel security of code-based schemes. Side channel security is a very important implementation aspect of any cryptographic algorithm. A side channel is given when a physical observable quantity that is measured during the operation of a cryptographic device allows an attacker to gain information about a secret that is involved in the cryptographic operation. The usual observables used in this respect are the duration of the operation (timing attacks [21]), or the power consumption as a function over time (power analysis attacks[22]), where targets of such attacks can be secret keys as well as messages.

Code-based schemes are not yet used in the field, however, it is important to study their side channel security already today. There are two main reasons for this: first of all, the amount and severity of side channel problems in a cryptographic scheme should be taken into account during the evaluation of its suitability. Concerning the cache-timing attacks against AES, the evaluation can be regarded to have been incomplete in this point [23]. This means, at the point where a certain committee decides about the future post-quantum cryptosystem to replace RSA, they will have to carefully weigh all implementation aspects of the candidates. But this presumes that at least a basic side channel analysis of all candidate schemes is available. The second reason for the necessity of an understanding of the scheme's side channel issues is that once the decision for its field application is taken, it is likely that there will be a tight time schedule for the detailed algorithmic specifications. On the one hand, the design of smart card hardware and software including the security evaluations will take years, on the other hand the advances in the field of quantum computing could possibly become threatening abruptly. From these considerations, it becomes apparent that at least the fundamental side channel issues must be known for the community to be able to work out concrete algorithmic specifications for the secure implementation of code-based schemes within a limited time frame.

As a consequence, in this thesis we perform a thorough side-channel analysis of code-based schemes. It focusses on the decryption operation, as it involves the message and the secret key as secrets. Accordingly, we performed an analysis of side channel vulnerabilities leaking either secret. For both types of attacks we found a number of vulnerabilities. For each vulnerability, we give the theoretical derivations and present experimental results. Where appropriate, we also provide a concrete attack and countermeasures.

All the message-aimed attacks exploit the effect that the number of bit flip errors in the ciphertext, the introduction of which is part of the regular encryption operation of these schemes, has a pervasive effect on the course of the decryption process. Different parts of this algorithm exhibit dependencies of their timing on this ciphertext property. This

allows a simple attack scheme, where the attacker manipulates an original ciphertext he wishes to decrypt and measures its decryption time. The information he gains through this process allows him to recover the message. For this type of attack, we present two timing attacks, exploiting timing differences stemming from different parts of the decryption algorithm. Our timing attacks were the first side channel attacks against code-based cryptosystems in the literature. Furthermore, we present a simple power analysis attack that is straightforwardly derived from the latter timing attack. As our last contribution to the topic of message-aimed attacks, we show that one of the root-finding variants that are already subject of the above-mentioned performance analysis exhibits a dramatic timing vulnerability with respect to message-aimed attacks.

We also analyse the relation of the message-aimed timing attacks against code-based cryptosystems to those against the RSA cryptosystem. In the course of this, we derive a methodology for the analysis of public key cryptosystems with certain properties with respect to this kind of attack. Furthermore, we discuss the situation of generic countermeasures for such attacks. We find that for the code-based schemes, such countermeasures are possible only at the expense of security, which has to be compensated by larger code parameters.

We also present two types of vulnerabilities that allow attacks against the secret key. For the first type, we present a practical attack that combines three different side channel vulnerabilities of the code-based decryption. All of these vulnerabilities employ specially crafted ciphertexts which feature only a small number of errors. Subtle differences in the control flow allow the attacker to collect linear and cubic equations, which eventually lead to the recovery of the whole private key. As the second type of key-aimed vulnerability, we demonstrate a timing vulnerability against one of the root-finding variants, which potentially also allows the complete recovery of the secret key. These are the first key-aimed timing attacks against code-based cryptosystems in the literature.

Concluding our contributions, we present two embedded implementations. The first one is an implementation based on a previous open source implementation. It features all the algorithmic variants (i.e. decryption with and without the parity check matrix, and the various root-finding variants) and side channel countermeasures presented in this thesis. We provide a detailed overview of the time and memory requirements of all these algorithmic variants on an embedded platform. The results show that the best algorithmic choice allows for efficient implementations on resource constrained platforms even without dedicated hardware support for code-based cryptosystems. The second one is an implementation on an actual smart card with fewer algorithmic optimizations. It shows that even under this condition and without hardware support, it is possible to implement the encryption and decryption operations with running times adequate at least for certain applications.

Before we come to the outline of this thesis' structure, we have to mention an important detail that applies to all our analyses: In this work, we restrict ourselves to the McEliece and Niederreiter schemes built on binary irreducible Goppa Codes as originally proposed for the McEliece scheme. Recently, a number of attempts have been made to reduce the large public key size of these schemes, by employing codes that exhibit a certain structure [24, 25, 26]. We justify the omission of these schemes in our analysis with the following

reasons: since a number of these attempts has already been shown to result in insecure cryptosystems [27, 28, 29], we conclude that all the recent proposals that reduce the key size using other codes than in the original McEliece scheme will have to prevail for some time until they can be granted the same trust as the original scheme, the security of which is still unquestioned after more than 30 years. For the sake of completeness, it is worth mentioning that there has been, in principle, a substantial reduction of the security of code-based PKCs [30], however this only applies to code parameter choices that are relevant for code-based signature schemes [31].

The structure of this thesis is as follows: In Chapter 2, we provide the fundamentals about the McEliece and Niederreiter cryptosystems. Next, in Chapter 3, we present our contributions to the performance aspects of the scheme. Afterwards, in Chapter 4, we present our results concerning the side channel security of the scheme. The two embedded implementations mentioned above are the topic of Chapter 5. Open problems that are not solved in this thesis are addressed in Chapter 6. Finally, we give a conclusion in Chapter 7.

2. Preliminaries

In this section we introduce the McEliece and Niederreiter public key cryptosystems. The McEliece scheme was introduced R.J. McEliece in 1978 [11]. It thus appeared about at the same time as the RSA scheme. H. Niederreiter introduced his scheme in 1986 [12]. However, neither the McEliece or the Niederreiter scheme has experienced much attention until the post-quantum cryptography question arose. This is due to the fact that, as we will see in the later chapters, the key sizes of these schemes are far greater than those of RSA or Elgamal cryptosystems.

In the following, we give first give basic preliminaries about Goppa Codes (Section 2.1) and then introduce the code-based PKCs (Sections 2.2 and 2.3).

2.1. The Patterson Algorithm for the Decoding of Goppa Codes

Goppa codes [32] are a class of linear error correcting codes. The McEliece PKC makes use of irreducible binary Goppa codes, so we will restrict ourselves to this subclass.

Definition 2.1.1. *Let $\mathbb{F}_{2^m}$ denote the finite field of order 2^m.*

Definition 2.1.2. *Let the polynomial $g(Y) = \sum_{i=0}^{t} g_i Y^i \in \mathbb{F}_{2^m}[Y]$ be monic and irreducible over $\mathbb{F}_{2^m}[Y]$, and let m, t be positive integers. Then $g(Y)$ is called a* Goppa polynomial *(for an irreducible binary Goppa code).*

Then an irreducible binary Goppa code is defined as $\mathcal{C}(g(Y), \Gamma) = \{\vec{c} \in \mathbb{F}_2^n | S_{\vec{c}}(Y) := \sum_{i=0}^{n-1} \frac{c_i}{Y-\alpha_i} = 0 \mod g(Y)\}$, where $n \leq 2^m$, $S_{\vec{c}}(Y)$ is the syndrome of $\vec{c}$, $\Gamma = (\alpha_i | i = 0, \ldots, n-1)$, the support *of the code, where the α_i are pairwise distinct elements of $\mathbb{F}_{2^m}$, and c_i are the entries of the vector $\vec{c}$.*

The code defined in such way has length n, dimension $k \geq n - mt$ and can correct up to t errors. In the following, however, we only consider the case $k = n - mt$.

As for any linear error correcting code, for a Goppa code there exists a generator matrix $G \in \mathbb{F}_2^{n \times k}$ and a parity check matrix $H \in \mathbb{F}_2^{mt \times n}$ [33]. Given these matrices, a message $\vec{m} \in \mathbb{F}_2^k$ can be encoded into a codeword $\vec{c}$ of the code by computing $\vec{c} = \vec{m}G$, and the syndrome $\vec{s} \in \mathbb{F}_2^{mt}$ of a (potentially distorted) codeword can be computed as $\vec{s} = \vec{c}H^T$. Here, we do not give the formulas for the computation of these matrices as they are of no importance for the understanding of the attack developed in this work. The interested reader, however, is referred to [33].

The error correction is performed with the Patterson Algorithm [34] given in Algorithm 1. In Step 1 of this algorithm, the syndrome vector is computed by multiplying the

potentially distorted code word by the parity check matrix, and then turned into the syndrome polynomial $S(Y)$ by interpreting it as an $\mathbb{F}_{2^m}^t$ element and multiplying it with the vector of powers of Y. The Patterson Algorithm, furthermore, uses an algorithm for finding roots in polynomials over $\mathbb{F}_{2^m}$ (root_find()), and the Extended Euclidean Algorithm (EEA) for polynomials with a break condition based on the degree of the remainder, Algorithm 9, given later in this work, when its details become relevant. Please note that all polynomials appearing in the algorithms have coefficients in $\mathbb{F}_{2^m}$.

The root finding can be implemented in a number of different ways, as we will see in Section 3.3.2. One important property of the error locator polynomial computed in the course of the Patterson Algorithm given that for w, the Hamming weight of the error vector $\vec{e}$, it holds that $w \leq t$, is

$$\sigma(Y) = \prod_{j \in \mathcal{E}} (Y - \alpha_j) = \sum_{i=0}^{w} \sigma_i Y^i, \tag{2.1}$$

where $\mathcal{E} = \{E_1, E_2, \dots E_w\}$ is the set of those indexes in the error vector $\vec{e}$ having value one.

2.2. The McEliece PKC

In this section, we give a brief overview of the McEliece PKC. The McEliece *secret key* consists of the Goppa polynomial $g(Y)$ of degree t and the support $\Gamma = (\alpha_0, \alpha_1, \dots, \alpha_{n-1})$; together, they define the secret code $\mathcal{C}$. The *public key* is given by the public $n \times k$ generator matrix $G_p = TG$ over $\mathbb{F}_2$, where G is a generator matrix of the secret code $\mathcal{C}$ and T is a non-singular $k \times k$ matrix over $\mathbb{F}_2$, the purpose of which is to bring G_p into reduced row echelon form, i.e. $G_p = [\mathbb{I}|G_2]$, which results in a more compact public key [14]. The *encryption* operation allows messages $\vec{m} \in \mathbb{F}_2^k$. A random vector $\vec{e} \in \mathbb{F}_2^n$ with Hamming weight $\mathrm{wt}(\vec{e}) = t$ has to be created. Then the ciphertext is computed as $\vec{z} = \vec{m}G_p + \vec{e}$.

The *decryption* is given in Algorithm 2. It makes use of the error correction algorithm, given by the Patterson Algorithm [34], shown in Algorithm 1. Note that the parity check matrix H employed here is not an essential part of the private key but rather a precomputed value.

Neither the McEliece nor the Niederreiter scheme is secure against adaptive chosen ciphertext attacks [35, 36]. Thus, a CCA2 conversion has to be applied; suggestions can be found in [36].

2.3. The Niederreiter PKC

In the Niederreiter PKC [12], the public key consists of the public parity check matrix $H_p = TH_s$, where H_s is the parity check matrix of the private code and $H_p \in \mathbb{F}_2^{(n-k) \times n}$, and T is chosen equivalently to its counterpart in the McEliece scheme. Furthermore, as in the McEliece scheme, H_p can be put in systematic form. Then, the public key will be of the same size as for the McEliece cryptosystem. The Niederreiter encryption is

Algorithm 1 The McEliece error correction with the Patterson Algorithm (err_corr($\vec{z}, g(Y), \Gamma$))

Input: the distorted code word $\vec{z} \in \mathbb{F}_2^n$, the secret Goppa polynomial $g(Y)$ and secret support $\Gamma = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$
Output: the error vector $\vec{e} \in \mathbb{F}_2^n$
1: $S(Y) \leftarrow \vec{z}H^\top \left(Y^{t-1}, \cdots, Y, 1\right)^\top$
2: $U(Y) \leftarrow S^{-1}(Y)$
3: $\tau(Y) \leftarrow \sqrt{U(Y) + Y} \mod g(Y)$
4: $(a(Y), b(Y)) \leftarrow \text{EEA}\left(g(Y), \tau(Y), \lfloor \frac{t}{2} \rfloor\right)$
5: $\sigma(Y) \leftarrow a^2(Y) + Yb^2(Y)$
6: $\mathcal{E} = \{E_0, \ldots, E_{t-1}\} \leftarrow \text{rootfind}(\sigma(Y))$ // if α_i is a root, then $\mathcal{E}$ contains i
7: $\vec{e} \leftarrow \vec{v} \in \mathbb{F}_2^n$ with $v_i = 1$ if and only if $i \in \mathcal{E}$
8: **return** $\vec{e}$

Algorithm 2 The McEliece Decryption Operation

Input: the McEliece private key $g(Y)$, Γ and the ciphertext $\vec{z} \in \mathbb{F}_2^n$
Output: the message $\vec{m} \in \mathbb{F}_2^k$
1: $\vec{e} \leftarrow \text{err_corr}(\vec{z}, g(Y), \Gamma)$
2: $\vec{m}' \leftarrow \vec{z} + \vec{e}$
3: $\vec{m} \leftarrow$the first k bits of $\vec{m}'$
4: return $\vec{m}$

depicted in Algorithm 3. The message is encoded into an error vector of weight t and the ciphertext is the corresponding syndrome, which can only be decoded by the holder of the private key. The encoding of the message is done with so-called constant-weight-word encoding, for instance as given in [37].

Algorithm 3 The Niederreiter Encryption Operation

Input: the Niederreiter public key $H \in \mathbb{F}_2^{(n-k)\times n}$ and the message m
Output: the ciphertext $z \in \mathbb{F}_2^{n-k}$
1: encode the message m into $e \in \mathbb{F}_2^n$, where $\text{wt}\,(e) = t$, using an appropriate algorithm (constant-weight-word encoding)
2: $z \leftarrow eH^T$

Algorithm 4 The Niederreiter decryption Operation

Input: the Niederreiter private key T, Γ, $g(Y)$ and the ciphertext $\vec{z} \in \mathbb{F}_{2^m}^{n-k}$
Output: the message m
1: $\vec{z}' \leftarrow T^{-1}\vec{z}$
2: $\vec{e} \leftarrow \mathrm{err_corr}(\tilde{z}', g(Y), \Gamma)$
3: decode $\vec{e}$ to $\vec{m}$ using an appropriate algorithm (constant-weight-word decoding)

3. Optimization for Resource-constrained Devices

In this chapter, we give contributions for the implementation of code-based PKCs on resource-constrained devices. The constraint that poses the greatest challenges to the realization of these schemes is that of memory. This is because the huge code-based public keys exceed the RAM size of embedded devices such as smart cards, and, furthermore, their storage on non-volatile memory is costly and inefficient. A solution to this problem that removes the need for the storage of public keys on the memory-constrained device in a public key infrastructure scenario altogether is given in Section 3.1.

Section 3.2 addresses the corresponding problem of the private key size, which poses a storage problem on memory-constrained devices. Implementations presented in previous works make use of the parity check matrix, which is not an essential part of the private key, but allows for fast decryption. However, the large size of this matrix poses a big cost problem. Also, many low cost microcontrollers would not even be able to store this matrix alone in their non-volatile memory for reasonable security parameters of the scheme. Accordingly, we present a contribution that demonstrates that contrary to this usual implementation choice, it is possible to greatly reduce the McElice private key size by omitting the parity check matrix and still achieve an acceptable running time of the decryption.

The third contribution, given in Section 3.3, addresses a running time improvement of the decryption operation of code-based PKCs, which, though in principle this performance aspect is not a problem of these schemes, is relevant on computationally weak platforms and useful for throughput maximization on servers. The running time improvement is achieved by determination of the most efficient algorithms for the generally most time consuming computational task during the code-based decryption operation: the finding of the roots of the error locator polynomial.

Finally, based on these results and those of other recent publications, in Section 3.4, we compare the efficiency of the McEliece and Niederreiter PKCs.

3.1. On-Line public Operation for Code-based Schemes

Since the large public key size is the major drawback of code-based schemes, we investigate how the key size affects the efficiency of the encryption operation, or the public operation in general, on embedded devices [8]. To this end, we first review some basics about public key infrastructures (PKI). Afterwards, we give the proposal for the realization of the public operation and present an example implementation on an embedded device.

3.1.1. The Storage Problem on Memory Constrained Devices

In order to illustrate the severity of the storage problem of code-based public keys on memory-constrained devices, we analyse the situation for smart cards, since these platforms are of utmost relevance for the adoption of a cryptosystem in a high security context. Typically, smart cards have less than 20 KB of RAM, while the available amount of non-volatile memory (NVM), e.g. flash-memory, can be as large as 512 KB [38, 39]. If a public key of a communication partner shall be temporarily stored on the device for the purpose of performing e.g. an encryption, it would have to be stored in the NVM since it exceeds the size of the RAM many times over. Specifically, the public keys will be at least 100 KB large for reasonable security parameters; for instance, this is the size resulting from the parameters for a security level of 102 bits given later in Table 3.1. The works [16, 4, 15] all describe implementations of code-based encryption schemes on embedded devices, where the public key is stored in the device's NVM. The drawbacks of storing such an amount of data in the device's NVM are first of all the cost of keeping such a large amount of memory available for this purpose but also the much slower writing speed compared to RAM access and the limited number of total write cycles in the memory module's lifetime. In order to circumvent these problems, we show in the following that the public operations can be executed by only storing very small parts of the public keys at any given time during the operation. This was already addressed in [17]. However, our approach also considers that these operations will mostly be carried out in a public key infrastructure context, which implies the verification of user public key certificates against issuer certificates. This is explained in some more detail in the next section.

3.1.2. Public Key Infrastructures

In a public key infrastructure (PKI), the trustworthiness of a public key is always verified against a trust anchor. From the trust anchor, which is usually a certification authority (CA) certificate, to the user certificate, there is a certificate chain involved. The trustworthiness of a certificate lower in the chain is guaranteed by its authentic digital signature created by the respective issuer, verifiable via the corresponding public key contained in the issuer certificate.

For the case of public key encryption, it means that a user A's public key intended for encryption is contained in the user certificate. A user B willing to encrypt a message for A thus goes through the following steps:

1. retrieve A's public key encryption certificate Enc-Cert_A (for example by accessing a database or asking A directly)

2. verify the authenticity of Enc-Cert_A by checking the signature on the certificate against the trust anchor (CA certificate) – in case of deeper CA hierarchies the whole certificate chain must be verified

3. encrypt the secret message using Enc-Cert_A and send it to A

For simplicity, in the following, we restrict ourselves to the case where the encryption certificate is directly signed by the trust anchor.

Since in this work we will address problems and solutions for embedded devices such as smart cards, we wish to point out why it is necessary to be able to carry out not only the private operations of a public key scheme (i.e. decryption or signature generation) but also the public operations on such devices. The reason is that devices such as smart cards are more and more used as standalone devices, which, for instance, reveal certain files only to parties which properly authenticate to them. This application context, which is found especially in electronic travel documents and electronic health cards, makes it necessary for smart cards to be able to carry out encryption as well as decryption operations.

In a naive approach, the public operation, which we here assume to be an encryption operation, would be realized by first retrieving the public key (embedded into a public key certificate containing also a signature) of the communication partner, storing it on the device, computing the hash value of the certificate's to-be-signed (TBS) data (which includes the code-based public key), verifying the signature, and finally encrypting the designated message using the certificate's public key. Consequently, in the following section, we explain how to implement the public operations of code-based schemes without storing full public keys on the device.

3.1.3. Description of the On-line Public Operation

The on-line public operation solves the following problem: an embedded device holding a CA certificate as trust anchor is supposed to encrypt a message for another party, which owns a public key encryption certificate containing a code-based public key. This encryption certificate is signed by the CA and accordingly must be verified against the CA certificate by the device. If the verification succeeds, the message shall be encrypted using the encryption certificate and be sent to the other party. This shall be done without storing the full encryption certificate, because of the problems explained in Section 3.1.1.

In our application scenario, we make the following assumptions:

1. We assume the usage of X.509 certificates [40]. Such a certificate consists of the sequence of the TBS data, followed by a field containing information about the signature algorithm and finally the signature. The signature ensures the authenticity of the TBS data, and is calculated based on their hash value, using a hash algorithm as specified in the preceding information field. Please note that the signature algorithm used to sign the user certificate needs not to be code-based (in which case the trust anchor certificate would contain a large code-based key itself). Instead, a hash based signature scheme [41] could be used. These schemes are also considered quantum computer resistant and feature extremely small public keys. Note, however, that the assumption of X.509 certificates is not a necessary precondition for our approach. We use this choice merely to be able to give a concrete description of the procedure. Any other solution which includes signed public keys can be covered by the approach.

2. We assume the usage of a hash function that allows the processing of the message efficiently in an on-line manner. This means that the memory necessary at any point during the hash computation should be small. This is a reasonable assumption, because all practically relevant hash functions must be designed to have this property. In the following description, we choose algorithms from the SHA-2 family, which operate on blocks of a maximal size of 128 bytes.

3. The transmission of the certificate occurs byte-wise, this means that every completely transmitted byte becomes immediately available to the software running on the smart card. However, an alternative condition where only portions of a number bytes become available to the software, does not change the scheme significantly. This is because, through buffering, in the second case the complete transmission is only delayed by a small constant.

4. As stated in Section 3.1.2, in our example we only use a single CA level. This could be easily generalized to certificate chains of arbitrary length. In this case, the device needs the whole chain of these CA certificates up to the trust anchor during the process. These additional certificates could for instance be fed into the device after the matrix multiplication.

The solution we propose is depicted in Figure 3.1. It works as follows:

1. The starting point is that the device wants to encrypt some message $\vec{m}$, which was generated on the device, with the McEliece public key of a communication partner. As explained in Section 2.2, the first part of the McEliece encryption is the multiplication of the message vector $\vec{m}$ by the public key matrix G.

2. Then, the X.509 certificate is input into the encryption device. The certificate starts with the TBS data, and while this data is received, the device computes the hash value of the TBS data and stores the relevant data from the certificate.

3. At the point where the code-based public key begins inside the TBS data, the device starts to carry out the matrix-vector multiplication row-wise:

$$r_j = \sum_i G_{ij} m_i.$$

   Note that the multiplication in $\mathbb{F}_2$ (logical AND) as well as the summation (logical XOR) can be efficiently carried out machine-word-wise.

4. After having received all TBS data, the device the holds matrix-vector product $\vec{r}$ and the hash value of the TBS data. Once it has also received the signature, it uses the TBS hash value and the stored CA certificate to verify the authenticity of the encryption certificate. If the verification succeeds, it completes the encryption process, which for example, in the case of McEliece, encompasses the creation and addition of the error vector, and finally outputs the ciphertext. If the verification fails, the device deletes the matrix-vector product and outputs an error message.

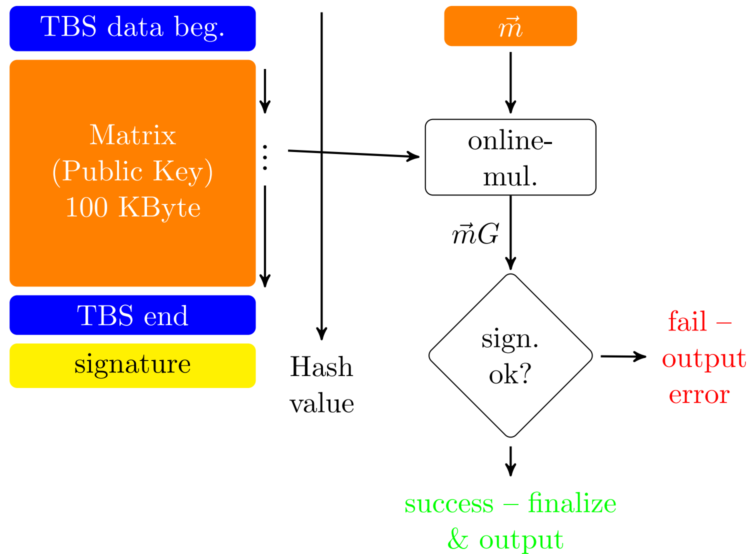


Figure 3.1.: Illustration of the on-line public operation for code-based cryptosytems.

Because only very small parts of the matrix are stored at any point in time, we call this approach “on-line public operation”. The result of this operation is exactly the same as that of the naive approach with the storage of the matrix presented in the previous section.

3.1.4. Transmission Rates

In this section, we give an overview of transmission rates available for embedded systems, especially smart card microcontrollers. For instance, an SLE66CLX360PE [39] smart card platform from Infineon Technologies AG features an ISO/IEC 14443 compliant contactless interface that can transmit up to 106 KB/s. This allows the transmission of a McEliece public key of size 100 KB, which corresponds to the code parameters for about 100-bit security given in Table 3.4, in about 1s, which can be considered at least acceptable for certain applications.

In the future, contactless transmission rates may be about 837.500 bytes/s [42], i.e. about 8 times higher than the rate considered above[1]. In the following, we will show that it is still feasible to sustain such a high transmission rate at typical smart card CPU speeds of about 30 MHz, if adequate hardware support is available on the device. Note that, in this case, there are still about 35 CPU cycles available between the receipt of two bytes.

[1]In the referenced work, this transmission rate is actually only achieved in the direction from the card to the reader. However, we want to use it merely as an orientation for the transmission rates achievable in the near future.

3.1.5. Example Implementation

We implemented the proposed approach of the on-line public operation in the C programming language on an ATUC3A1512 32-bit microcontroller from Atmel's AVR32 family. We chose an embedded 32-bit platform basically because SHA-256, which we chose as the hash algorithm in the example implementation, is designed for 32-bit platforms. There also exist 32-bit smart card controllers [43], thus our evaluations are significant for this type of platform.

The personal computer (PC), acting as the client device in this setup, communicates with the AVR32 over a serial line. For the implementation of the serial communication, on the AVR32, we used the API for the device's Universal Asynchronous Receiver Transmitter (UART), provided by Atmel. On the side of the personal computer, we used the API to the serial port of the Linux operating system. The PC can send commands to the AVR32, which are formed by a six-byte header and optional payload data, the length of which is encoded in the last four header bytes. The first header byte is zero for all commands, and the second byte determines one of the following commands:

- set the vector to multiply
- carry out the on-line multiplication (starts an interactive protocol for the matrix transmission described below)
- get the multiplication result from the AVR32
- get the hash result from the AVR32

The AVR32 responds to these commands by sending a two-byte status code and optional data payload preceding the status code, or in the case of the on-line multiplication command, by starting an interactive protocol.

This protocol is depicted in Figure 3.2. As a precondition, the vector to multiply has to be set in the device through the corresponding command. After receiving the on-line multiplication command (which does not carry payload data), the AVR32 sets up two buffers $B1$ and $B2$, which are of an equal predefined size. It sends a two-byte acknowledgement (ACK) code to the PC as the answer to the command. Then, the PC sends the first matrix part, which is of equal size as the buffers $B1$ and $B2$. The receipt of a single byte over the UART interface of the AVR32 triggers an interrupt which is serviced by an Interrupt Service Routine (ISR), which writes the byte to the next free position in $B1$. After the first block has been received completely, the AVR32 sends another ACK code to the PC, who in turn reacts by sending the next part. At this point, the AVR32 exchanges the role of the buffers $B1$ and $B2$: the data are now received to $B2$ (which did not play any role while receiving the first part), and $B1$, containing the first matrix part, is fed into the SHA-256 computation and the matrix multiplication. Both, the hashing and the matrix multiplication are implemented as objects, which can be updated by calling routines that take arbitrary amounts of data as an argument.

For hash functions, this is the standard implementation technique. Because demanded by our approach, we adopted this technique for the matrix multiplication. In our implementation, the matrix-vector multiplication is carried out column-wise. The advantages

and disadvantages of this approach in contrast to row-wise multiplication are discussed in Section 3.1.9. The multiplication object knows the number of rows and columns of the matrix and has the source vector set. As the matrix data is fed column-wise, it keeps track of the current row and column position. It processes the current column by carrying out the logical AND (multiplication in $\mathbb{F}_2$) between the matrix column and the vector 32-bit word-wise, and computes the XOR (addition in $\mathbb{F}_2$) with a 32-bit accumulator. When a column is finished, the parity (i.e. sum of all the word's bits in $\mathbb{F}_2$) of the accumulator is written to the corresponding result bit.

The hash implementation is based on the open source implementation [44]. The C source code allows for the complete unrolling of the SHA-256 compression function through a macro definition. Activating loop unrolling resulted in a performance gain of 1.6 for the hash function computation. All the performance data given later are based on this implementation choice.

3.1.6. Non-interactive Version of the Protocol

It turned out that the interactive protocol incurs significant delay in the communication, which most probably results from the fact that our PC program is running in user space and, thus, sending and receiving data via the serial interface is delayed. If the protocol were implemented in a card terminal, which could be the case in a real world implementation of the on-line multiplication, such issues would not arise. To show the efficiency of the approach, we modified the protocol depicted in Figure 3.2: the AVR32 does not send any ACK answers beyond the very first one. Consequently, the matrix data are sent as a continuous stream after the AVR32 has sent the initial ACK. In this way, the protocol looses the feature that it works independently of the ratio of transmission speed and computation speed: in this non-interactive setting, it must be guaranteed that the hash and the multiplication computation of the processed buffer have finished before the receive buffer has been completely filled. With this approach, the performance could be improved by a factor of roughly 1.3 compared to the interactive variant of the protocol. The concrete results are discussed shortly.

3.1.7. Simulation of higher Transmission Rates

On the chosen AVR32 platform, the maximal transmission speed is given by a baud rate of 460,800. In the RS232 transmission format, each data byte is encoded in 10 bits, yielding a net transmission rate of 46,080 bytes/s. In order to demonstrate the computation speed that would be possible beyond this limitation, we implemented a means of simulating higher transmission speeds. This is achieved by creating a matrix whose rows have repetitive entries, i.e. the values of 8-bit chunks repeats r times. An example of the beginning of a row for $r = 4$ would be

```
0x1D, 0x1D, 0x1D, 0x1D, 0xA3, 0xA3, 0xA3, 0xA3, 0x22, ...
```

In this setting, on the PC side, such a repetitive matrix is generated. When the matrix is transmitted, however, each repeated element is sent only once. On the receiving side,

the repetition value r is also known and each received byte is appended to the buffer r times. In this way, we simulate a transmission rate $B_{\text{sim}} = rB_{\text{real}}$, where B_{real} is the actual UART transmission rate.

3.1.8. Experimental Results

We performed measurements for the non-interactive version described in the previous section on the AT32UC3A1 platform. Here, we used a matrix with 1000 rows and 800 columns, i.e. yielding a size of 100,000 bytes. This is approximately the size of McEliece public keys with 100 bit security [4]. In all our measurements, the CPU speed of the AVR32 was set to 33 MHz, since also today's contactless smart card platforms run at approximately this speed [38]. A receive buffer size of 1536 bytes was used.

The time for the complete on-line multiplication under these parameters is 279ms at a simulated transmission rate $B_{\text{sim}} = 368,640$ (using $r = 8$). This amounts to an effective computational throughput of 92 cycles/byte for the SHA-256 hash computation and the matrix multiplication.

We also determined the actual computational throughput on the platform for both computational tasks: it is 4.2 cycles/byte for the matrix multiplication and 55.6 for the hash computation; taken together, this amounts to 59.8 cycles/byte. Based on this, if there were no other delays in the whole process, a transmission rate of 551,839 bytes/s could be theoretically supported.

Furthermore, we measured the random error vector creation as the second part of the encryption operation for code parameters $n = 2048$ and $t = 50$ to be less than 4 ms at a CPU speed of 33MHz, the addition (XOR) of the error vector to the intermediate vector is certainly even much less complex, and thus completely negligible for the timings considered here.

The transmission speed of 386,640 bytes/s, which can be sustained in our test setup, is approximately half of that of the research implementation presented in [42], already mentioned in Section 3.1.4. Thus, our results show that even without dedicated hardware, today's embedded platforms already enable computation speeds for the hash calculation and the matrix multiplication not too far from the associated transmission rates that can be expected to be supported by contactless devices in the near future. This makes it feasible that, with adequate hardware support, the full 837.500 Byte/s rate given in [42] can be supported by the throughput of the computational tasks.

3.1.9. Column-wise vs. Row-wise Matrix-Vector Multiplication

The row-wise computation of the matrix-vector multiplication is an alternative to the column-wise approach. In this case, the computation of the result is according to $b = \sum_i G_i a_i$, where G_i is the vector represented by the i-th row of G. This means that a row G_i is added to the result if the corresponding bit a_i is one, otherwise nothing has to be done. In the normal case, where the whole matrix is available instantly, this approach has a significant advantage over the column-wise approach since, on average, half of the vector a's bits have value zero. But in the case of the on-line public opera-

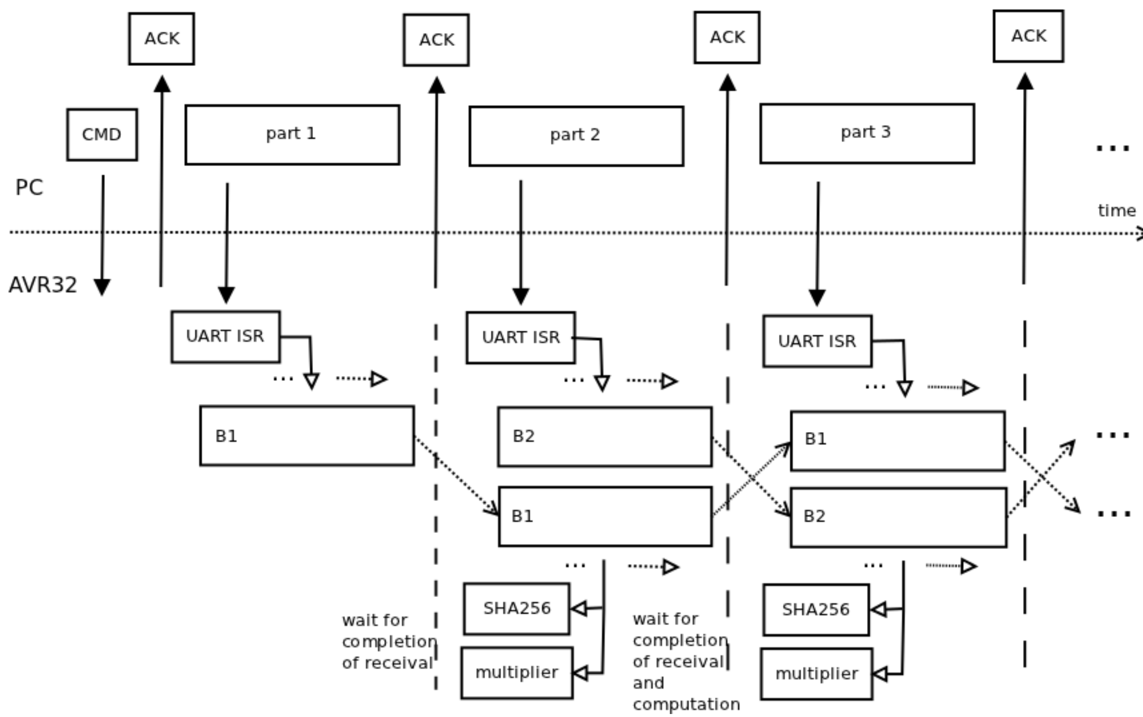


Figure 3.2.: Schematic overview of the interrupt based implementation of the on-line multiplication.

tion, this advantage disappears, since the matrix-vector multiplication's running time is determined by the transmission time alone (under the assumption of sufficient computational power of the device as analysed in Section 3.1.4). The row-wise approach would only have an advantage if the saved computational effort could be used to perform other tasks, which can be assumed to be rather unlikely or at least of minor relevance in the context of embedded devices such as smart cards.

On the other hand, the disadvantage of the row-wise multiplication lies in its potential side-channel vulnerability. Specifically, if an attacker is able to find out whether the currently transmitted row is added or ignored, for instance by analyzing the power trace [22], he can deduce the value of the secret bit a_i. Of course, countermeasures can be implemented. A certain randomization could for instance be introduced by keeping a number of received rows in a buffer and processing them in a randomized order. However, whether the questionable computational advantage of this method is worth such efforts must be decided in a concrete implementation scenario.

In any case, once the X.509 key format for a code-based scheme is defined, the choice for one of the two methods is taken. While it would still be possible, in that case, to transmit the matrix in the other orientation, in order to carry out the multiplication, the on-line hash computation only works if the correct orientation is used.

3.1.10. Code-based Signature Schemes

A number of code-based signature schemes have been proposed. In the following, we will very briefly address two of these schemes with the goal of showing that the proposed approach for the on-line public operation is applicable to both of them.

In [31], the McEliece scheme is inverted, in the sense that the signer proves his ability to decode a binary vector related to the message using a certain code. Thus, the signature verification basically consists of a matrix-vector multiplication, like for the encryption schemes described in Section 3.1.3. For security considerations concerning this scheme, please refer to [45, 46].

A signature scheme involving two binary matrices as the public key is presented in [47]. In the verification operation, both matrices have to be multiplied by a vector. Thus, the on-line public operation can be carried out by transmitting them one after another. Note, however, that the originally proposed parameters for this scheme are insecure [48].

3.2. McEliece Decryption without the Parity Check Matrix

The essential private key of the McEliece PKC is given by the Goppa Polynomial $g(Y)$ and the support Γ. However, for fast syndrome computation, all previous works about implementations of the scheme feature the parity check matrix as part of the private key [4, 16, 17, 20]. Since the size of the parity check matrix is in the same order of magnitude as that of the public key (see for instance Table 3.1), this implementation choice poses a great problem for memory-constrained devices. Consequently, in this section, we devise an optimized algorithm for the syndrome computation without the parity check matrix,

and compare its performance with the syndrome computation based on the availability of this matrix [8].

3.2.1. Optimized Algorithm for the Syndrome Computation without the Parity Check Matrix

In this section, we provide an algorithm with optimized running time for the syndrome computation without the parity check matrix. It is achieved by tailoring the EEA to the specific case of the syndrome computation.

In the McEliece scheme, the first step of the decryption operation is, according to Algorithm 2, the computation of the syndrome polynomial $S(Y)$ as

$$S(Y) \equiv \sum_{i=1}^{n} \frac{c_i}{Y \oplus \alpha_i} \mod g(Y), \tag{3.1}$$

where $g(Y)$ is the Goppa Polynomial, c_i is the i-th ciphertext bit and the α_i is the i-th support element.

From (3.1) we see that the syndrome computation without the parity check matrix is, in principle, achieved by invoking the Extended Euclidean Algorithm (EEA) with $g(Y)$ and $Y \oplus \alpha_i$ for all $i \in \{0, \ldots, n-1\}$ as the initial remainders. Here, since $g(Y)$ is irreducible, the coefficient to $Y \oplus \alpha_i$ is $(Y \oplus \alpha_i)^{-1} \mod g(Y)$. This coefficient is considered the result of the EEA, in the following. Finally, the results of all these EEA computations have to be summed up to yield the syndrome polynomial $S(Y)$. All these EEA invocations execute in a single iteration. Accordingly, in an implementation of the syndrome computation, a number of optimizations are possible. The resulting algorithm is given in Algorithm 5. There, $z[i]$ denotes the i-th ciphertext bit and B_j the coefficient to Y^j of $B(Y)$, etc. The complexity of the algorithm as a function of the ciphertext's Hamming weight w is

$$C_{\text{syndr}}(w) = 2wt(C_{\text{mult}} + C_{\text{add}}) + wC_{\text{inv}}.$$

Its average complexity (i.e. the complexity for a ciphertext with Hamming weight $n/2$, which is the expectation value for a random ciphertext), expressed in the terms of additions, multiplication and inversions in $\mathbb{F}_{2^m}$, is

$$C_{\text{syndr}} = nt(C_{\text{mult}} + C_{\text{add}}) + \frac{n}{2}C_{\text{inv}}.$$

This value is derived as follows: C_{syndr} is obviously the cost of the computation for ciphertexts with the mean Hamming weight $n/2$. It is also the average cost for equally distributed random ciphertexts. This can be seen from a simple symmetry argument making use of the linearity of $C(w)$: For each Hamming weight $w_1 = (n/2) - x$ there exists a Hamming weight $w_2 = (n/2)+x$ with the same probability of occurence $p_{(n/2)-x}$. Assuming even n, we find for the expectation value of $C_{\text{syndr}}(w)$:

$$C_{\text{syndr}} = p_{(n/2)}C(n/2) + \underbrace{p_{(n/2)-1}C((n/2)+1) + p_{(n/2)-1}C((n/2)-1)}_{2p_{(n/2)-1}C(n/2)} + \ldots = C(n/2),$$

		$n = 2048$, $t = 50$ 102 bit security		$n = 2960$, $t = 56$ 122 bit security	
		cycles	t @ 33 MHz	cycles	t @ 33 MHz
A	decryption	$2.00 \cdot 10^6$	61 ms	$3.12 \cdot 10^6$	95 ms
	syndrome computation	$0.26 \cdot 10^6$	8 ms	$0.39 \cdot 10^6$	12 ms
	private key size	155,640 byte		274,192 byte	
B	decryption	$4.42 \cdot 10^6$	134 ms	$7.39 \cdot 10^6$	224 ms
	syndrome computation	$2.65 \cdot 10^6$	80 ms	$4,71 \cdot 10^6$	143 ms
	private key size	14,840 byte		25,552 byte	

Table 3.1.: Comparison of private key sizes, cycle counts and corresponding timings taken for the McEliece decryption operation and its suboperation, the syndrome computation, with (A) and without (B) the parity check matrix.

where the probabilities sum up to 1. Assuming odd n instead, the Hamming weights to consider are $(n/2) \pm \frac{1}{2}$, $(n/2) \pm 1\frac{1}{2}$ etc., which changes nothing about the result.

3.2.2. Implementation and Performance Results

We implemented this algorithm in a McEliece PKC implementation based on the open source implementation [20] presented in [14], using the Berlekamp Trace Algorithm for the root-finding. Table 3.1 shows the timing results measured on an Atmel AT32 AP7000 CPU, a CPU similar to the AT32UC3A1. The CPU runs at 150 MHz, but we give the according running time for the typical smart card CPU speed of 33 MHz, which was already employed in Section 3.1.5. Each cycle count was obtained by carrying out the operation ten times and taking the mean of the results. The smaller parameter set features 102 bits of security [4], the larger one will be introduced in Section 3.3.3.

The respective private key sizes without the parity check matrix given in Table 3.1 are formed by the Goppa Polynomial $g(Y)$, the support Γ, a matrix for computing the square root modulo $g(Y)$, which is needed to speed up the decryption, and the logarithm and anti-logarithm tables for $\mathbb{F}_{2^m}$, each of size $\lceil \log_2 n \rceil$ elements, i.e. a total of 8,192 resp. 16,384 bytes for either parameter set (each element occupies 2 bytes). These latter tables need not necessarily be stored in the key; instead, they can be created in RAM before the decryption operation, if allowed by the memory constraints of the given platform. From this example implementation, which does not use any hardware support for the $\mathbb{F}_{2^m}$ operations or DSP instructions, we see that the decryption time approximately doubles when the parity check matrix is not stored as part of the key, but due to the general speed advantage of the McEliece scheme over RSA or Elliptic curve based schemes [14, 16], these timings are still highly competitive.

Algorithm 5 The Syndrome computation without parity check matrix

Input: the ciphertext $\vec{z} \in \mathbb{F}_2^n$, and the Goppa Polynomial $g(Y) \in \mathbb{F}2^m[Y]$ of degree t
Output: the syndrome polynomial $S(Y) \in \mathbb{F}2^m[Y]$ of degree $\leq t-1$

```
S(Y) ← 0
for i ← 0 up to n − 1 do
  if z[i] = 1 then
    B(Y) ← 0
    b ← g_t
    for j ← t − 1 down to 0 do
      B_j ← b
      b ← b · α_i ⊕ g_j
    end for
    f ← b^{-1}
    for j ← 0 up to deg (B(Y)) do
      S_j ← S_j ⊕ f · B_j
    end for
  end if
end for
```

3.3. Efficient Root-Finding during the Decryption

Though on personal computers the speed of the operations of code-based PKCs is known to be the strong side of these schemes, it is important to know their performance in pure software implementations on computationally weak platforms, in order to determine the extent to which hardware support is necessary on devices such as smart cards. Furthermore, the minimization of an algorithm's running time is always useful when it comes to keeping small the computational load on server systems.

From this point of view, the root-finding of the error locator polynomial $\sigma(Y)$ in code-based decryption deserves special attention because, as addressed already in previous work [15, 16], it is in general the most time-consuming part of the decoding algorithm. Furthermore, it is the only computational task during the decryption for which different algorithms have been proposed. Thus, in this section, after giving some preliminaries about software implementations of $\mathbb{F}_{2^m}$ operations, we present four algorithmic variants for this computational task, along with two new hybrid variants. Finally, we give a performance comparison in terms of running times on a personal computer platform [7].

3.3.1. Remarks about the $\mathbb{F}_{2^m}$ Operations

Before we start with the descriptions of the root-finding algorithms, we want to point out some details concerning the costs of the basic $\mathbb{F}_{2^m}$ operations that are involved, i.e. addition and multiplication.

While $C_{\text{gf_add}}$, the cost of an addition in $\mathbb{F}_{2^m}$ is given by a simple XOR operation, the multiplication in $\mathbb{F}_{2^m}$ is much more complex and has a number of variants. An

efficient software implementation of finite field arithmetics with characteristic 2 and small extension degrees is realized by the use of one lookup table for the logarithm of each non-zero element to the base of some primitive element, and the corresponding anti-logarithm table.

The standard multiplication, as it is for instance implemented by the "C" macro `gf_mul()` in HyMES [20], which is used throughout their code, takes arguments in the normal representation and outputs the result in normal representation. This type of multiplication, we refer to as *mul_nnn*. Its cost is two conditional branches to check whether the arguments are zero, three table lookups, one arithmetic ADD, and reduction of the result modulo the fields multiplicative order, which in turn consists of several instructions. In the general case, this multiplication is needed, as in most places in the algorithms involved in the syndrome decoding, multiplication and addition in $\mathbb{F}_{2^m}$ are intermixed, and moreover, operands having value zero cannot be excluded.

However, when operands are known to be non-zero, and multiplications are carried out subsequently, other forms of the multiplication that have results (a in the algorithm description *mul_abc*) or operands (b and c) in the logarithmic representation, are more efficient:

- *mul_lll* consists only of one arithmetic ADD (a certain number of these multiplications can be carried out before a reduction modulo the multiplicative order becomes necessary to avoid overflowing the register)
- *mul_nln* saves one conditional branch and one table lookup compared with *mul_nnn*

This rough review of the finite field arithmetic implementations in software shows that speaking of the cost of multiplication in $\mathbb{F}_{2^m}$ is not sufficient to describe actual computational costs, because there exist drastic differences in the cost with respect to how the multiplication is embedded into the algorithm.

3.3.2. Variants of Root Finding

In the following subsections, we give brief descriptions of the root-finding algorithm variants analysed in this work.

3.3.2.1. Exhaustive Evaluation with and without Division

The most straightforward implementation of the root-finding is to simply evaluate the polynomial $\sigma(Y)$ for each element of the code.

The complexity of this algorithm is given as

$$C_{\mathrm{eval-rf}} = (n-1)t(C_{\mathrm{gf_add}} + C_{\mathrm{mul_nln}})$$

Remember that n is the code length and t is the error correcting capability. Taking a look at the Horner Scheme evaluation used here, we see that when evaluating $\sigma(Y)$, we can transform $x \neq 0$ to the logarithmic representation, avoiding some unnecessary table lookups, i.e. making use of *mul_nln* .

The algorithm can be sped up by dividing the polynomial $\sigma(Y)$ by each root found, resulting in Algorithm 6. Such a division has basically the same complexity as the evaluation of the polynomial for one single element of $\mathbb{F}_{2^m}$. In the following, we will call these two variants *eval-rf* and *eval-div-rf*.

Algorithm 6 The algorithm *eval-div-rf* for finding the roots of a polynomial over $\mathbb{F}_{2^m}$

Input: the polynomial $\sigma(Y)$ over $\mathbb{F}_{2^m}$
Output: the set $\mathcal{E}$, where γ_i is a root of $\sigma(Y)$ if and only if $i \in \mathcal{E}$
1: $\mathcal{E} = \emptyset$
2: **for** $i = 0$ up to $i = n - 1$ **do**
3: **if** $\sigma(\gamma_i) = 0$ **then**
4: $\mathcal{E} \leftarrow \mathcal{E} \cup \{i\}$
5: $\sigma(Y) \leftarrow \sigma(Y)/(Y - \gamma_i)$
6: **end if**
7: **end for**
8: **return** $\mathcal{E}$

3.3.2.2. Berlekamp Trace Algorithm

Another root-finding variant is the Berlekamp Trace Algorithm [49]. We adopted the implementation of this algorithm found in the HyMES open source implementation of the McEliece scheme [14, 20] for purposes of comparison with the other variants. For completeness, we provide the description of this algorithm as originally given in [14] in Algorithm 7. The initial call to this recursive algorithm is given as $\mathrm{BTA}(\sigma(Y), 1)$, which we will refer to as *BTA-rf* for the remainder of this work. The trace function is defined as $\mathrm{Tr}(Y) = Y + Y^2 + Y^{2^2} + \ldots + Y^{2^{m-1}}$, and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ is a standard basis of $\mathbb{F}_{2^m}$.

The HyMES implementation actually performs precomputations of $\mathrm{Tr}(\beta_i Y) \mod \sigma(Y)$ with $i \in \{0, \ldots, m-1\}$, as described in [50]. In that work, the complexity of the BTA is

Algorithm 7 The recursive Berlekamp Trace Algorithm $\mathrm{BTA}(\sigma(Y), i)$.

Input: the error locator polynomial $\sigma(Y)$
Output: the set of roots of $\sigma(Y)$
1: **if** $\deg(\sigma(Y) \leq 1)$ **then**
2: **return** root of $\sigma(Y)$
3: **end if**
4: $\sigma_0(Y) \leftarrow \gcd(\sigma(Y), \mathrm{Tr}(\beta_i Y))$
5: $\sigma_1(Y) \leftarrow \gcd(\sigma(Y), 1 + \mathrm{Tr}(\beta_i Y))$
6: **return** $\mathrm{BTA}(\sigma_0(Y), i+1) \cup \mathrm{BTA}(\sigma_1(Y), i+1)$

given as $\mathcal{O}(mt^2)$. In order to make a fair comparison of the various root-finding variants in terms of performance, we optimized the existing implementation of the algorithm by applying the more cost-efficient versions of multiplication in $\mathbb{F}_{2^m}$ as discussed in Section 3.3.1, where possible. As a result, the running time was reduced by about 10%.

Furthermore, *BTA-rf* can be sped up by using specific root-finding algorithms for polynomials of low degree [50]. We only implemented the variant where the roots of polynomials of degree two are determined through the use of a lookup table of size $2n$ bytes (supporting $m = 15$ at most), presented in the referenced work. Then, in the recursion, this algorithm is invoked instead of Algorithm 7 whenever the degree of $\sigma(Y)$ is two. In the following, we refer to this algorithm by BTZ_2*-rf*.

3.3.2.3. Root Finding with linearised Polynomials

In this section, we explain a root-finding method based on decomposing a polynomial in $\mathbb{F}_{2^m}[Y]$ into linearised polynomials [51]. The idea of this approach is based on the fact that the exhaustive evaluation of a linearised polynomial can be done with much less computational cost than for general polynomials.

Definition 3.3.1. *A polynomial $L(Y)$ over $\mathbb{F}_{2^m}$ is called a linearised polynomial if $L(Y) = \sum_i L_i Y^{2^i}$, where $L_i \in \mathbb{F}_{2^m}$.*

As shown in [51], an affine polynomial of the form $A(Y) = L(Y) + \beta$ with $\beta \in \mathbb{F}_{2^m}$ can be evaluated for the value $Y = x_i$ based on the evaluation result for $Y = x_{i-1}$ as stated in the following theorem:

Theorem 3.3.1. *Let $x_i \in \mathbb{F}_{2^m}$ and $A(x)$ an affine polynomial. Then*

$$A(x_i) = A(x_{i-1}) + L(\Delta_i),\ \Delta_i = x_i - x_{i-1}. \tag{3.2}$$

Proof. Let $\{\alpha^0, \alpha^1, \ldots, \alpha^{m-1}\}$ be a standard basis of $\mathbb{F}_{2^m}$, i.e. $x_i = \sum_{k=0}^{m-1} x_{i,k}\alpha^k$ with $x_{i,k} \in \mathbb{F}_2$. The notation for an unsubscripted x is adopted naturally.

First, we prove that $L(y) = \sum_{k=0}^{m-1} y_k L(\alpha^k)$:

$$\begin{aligned}
L(y) &= \sum_j L_j y^{2^j} = \sum_j L_j \left(\sum_{k=0}^{m-1} y_k \alpha^k\right)^{2^j} = \sum_j L_j \sum_{k=0}^{m-1} (y_k\alpha^k)^{2^j} \\
&= \sum_j L_j \sum_{k=0}^{m-1} y_k \alpha^{k2^j} = \sum_{k=0}^{m-1} y_k \sum_j L_j (\alpha^k)^{2^j} \\
&= \sum_{k=0}^{m-1} y_k L(\alpha^k).
\end{aligned}$$

Using this result, we prove that $L(x_i) - L(x_{i-1}) = L(x_i - x_{i-1}) = L(\Delta_i)$:

$$\begin{aligned}
L(x_i) - L(x_{i-1}) &= \sum_{k=0}^{m-1} x_{i,k} L(\alpha^k) - \sum_{k=0}^{m-1} x_{i-1,k} L(\alpha^k) \\
&= \sum_{k=0}^{m-1} (x_{i,k} - x_{i-1,k}) L(\alpha^k) = L(x_i - x_{i-1}),
\end{aligned}$$

Now the theorem is proven:

$$A(x_i) = \beta + L(x_i) = A(x_{i-1}) - L(x_{i-1}) + L(x_i) = A(x_{i-1}) + L(x_i - x_{i-1}).$$

□

In order to take advantage from this, we perform the evaluations of the polynomial in Gray-code ordering, i.e. for all i we have that x_i and x_{i+1} differ only in one single bit in their binary string respresentation. Then, there are only m different values $L(\Delta_i)$ that have to be precomputed.

To have a benefit in the evaluation of general polynomials, they have to be decomposed in a way that as many affine polynomials as possible appear in the decomposition. Such a generic decomposition of a polynomial $f(Y) = \sum_{i=0}^{t} f_i Y^i$, also given in [51], is

$$f(Y) = f_3 Y^3 + \sum_{i=0}^{\lceil (t-4)/5 \rceil} Y^{5i} A_i(Y), \tag{3.3}$$

where

$$A_i(Y) = f_{5i} + \sum_{j=0}^{3} f_{5i+2^j} Y^{2^j}. \tag{3.4}$$

The evaluation of each $A_i(x_i)$ is done efficiently according to (3.2). To this end, the exhaustive evaluation of (3.3), i.e. evaluation of the polynomial for all elements of $\mathbb{F}_{2^m}$, is done with the x_i being in Gray-Code ordering. Specifically, we use the Gray Code generated by $x_i = (\mathrm{i} >> 1) \oplus i$, where ">>" denotes logical right shift. The actual computational cost for the root-finding with linearised polynomials includes the sum of the precomputations, i.e. the computation of the $A_i(Y)$. This cost is given in [51], it is however negligible for secure code parameters. The dominating cost is that of computing $f(Y)$ for all n code elements:

$$\begin{aligned} C_{\mathrm{dcmp-rf}} = (n-1)(&C_{\mathrm{gf_log}} + C_{\mathrm{squ_ll}} + 2C_{\mathrm{mul_lll}} + C_{\mathrm{mul_nll}} + \\ &\lceil (t+1)/5 \rceil \left(2C_{\mathrm{gf_add}} + C_{\mathrm{mul_lll}} + C_{\mathrm{mul_nln}}\right)), \end{aligned}$$

where $C_{\mathrm{gf_log}}$ refers to the cost of converting a $\mathbb{F}_{2^m}$ element from normal to logarithmic representation and $C_{\mathrm{squ_ll}}$ is the cost of squaring in the logarithmic representation.

3.3.2.4. New Hybrid Variants

We also implemented two new hybrid variants. We label the first *dcmp-div-rf(a,b)*, where a and b are parameters of the algorithm. It is given simply by restarting the whole *dcmp-rf* everytime after, through divisions by found roots, the degree of $\sigma(Y)$ has been reduced by at least $5a$ to $5k+4$ for some positive integer k. Furthermore, once in this process $\deg(\sigma(Y)) = b$ is reached, no more divisions are performed, and standard *dcmp-rf* is used henceforth.

A further variation of this is given through *dcmp-div-BTZ*$_2$*-rf(a,b)*. It is equal to *dcmp-div-rf(a,b)* until $\deg(\sigma(Y)) = b$. Then, when $\sigma(Y)$ has degree b, *BTZ*$_2$*-rf* is invoked to find the remaining roots.

3.3.3. Performance of the Root-finding Variants

In this section, we give a comparison of the performance of the root-finding algorithms given in Section 3.3.2.

The code was compiled with GCC version 4.5.2 with the optimization options

```
-finline-functions -O3 -fomit-frame-pointer -march=i686 -mtune=i686
```

and run on a Intel(R) Core(TM)2 Duo CPU U7600 CPU.

In the following, we give results for two parameter sets based on the propositions given in [52] for 128 and 256 bit security, which are based on code parameter choices aiming at the minimization of the public key size, which is known to be the most problematic feature of code-based cryptosystems. The only deviation of our parameter choices is the number of errors added during encryption: in [52], List Decoding [53], which allows for the correction of more than t errors, is assumed during decryption. For the smaller parameter set, they choose $t+1$ errors and for the larger $t+2$ errors. The reduction of security of the smaller parameter set in our implementation using only t errors, however, can easily be bounded by understanding that an attacker can get, from a ciphertext with $t+1$ errors to t errors by guessing one error position correctly, the success probability of which is $(t+1)/n = 0.02$. Accordingly, the security of the scheme with t errors cannot be smaller than $128 - \log_2(1/0.02) > 122$ bits. A corresponding calculation for the larger parameter set gives a lower bound of 244 bits.

It is noteworthy that these parameter sets, optimized for minimal public key size for a given security level, use codes with $n < 2^m$, and that this has different effects for our four candidate algorithms. Both *eval-rf* and *eval-div-rf* are faster for $n < 2^m$ in contrast to $n = 2^m$, however, for the latter, the speedup is less than for the former, as in *eval-div-rf* the roots found at the end of the support cause less effort. *dcmp-rf* also benefits from $n < 2^m$, since also then the support can be built from a Gray Code. *BTA-rf*, however, has the same running time no matter whether $n < 2^m$ or $n = 2^m$.

Tab. 3.2 gives the results for the mentioned parameters. We clearly see that *BTZ_2-rf* and *dcmp-rf* are almost equally fast and that *dcmp-div-rf*, the parameters of which were experimentally optimized for the given code parameters, has even better performance. For the smaller parameter set, *dcmp-div-BTZ_2-rf* has a small edge on *dcmp-div-rf*, for the larger code parameter set no parameters of *dcmp-div-BTZ_2-rf* allowing an improvement over *dcmp-div-rf* were found.

Besides the execution time, memory demands are certainly of great importance, at least for the application on resource constrained platforms. In Section 5.1, the memory consumption of the McEliece decryption on an embedded platform for the various root-finding variants will be given.

However, it must be pointed out, that for parameter choices using large values of n while trying to minimize t, *BTA-rf* clearly wins against *dcmp-rf*. In Tab. 3.3, we give the root-finding running times for these two algorithms for some parameter sets taken from the results given in [14]. Note that the security levels for all these parameter sets are lower than 128 bit; furthermore, their concrete values given in [14] are deprecated by [52]. The drawback of such a parameter choice with small t is a large public key size.

parameters	security level	root-finding algorithm	cycle count / 10^5 cycles
$n = 2960$, $t = 56$	122 bit	*eval-rf*	21.16
		eval-div-rf	16.26
		BTA-rf	8.89
		BTZ_2*-rf*	6.33
		dcmp-rf	6.45
		*dcmp-div-rf(*1,19*)*	5.42
		dcmp-div-BTZ_2*-rf(*1,19*)*	5.12
$n = 6624$, $t = 115$	244 bit	*eval-rf*	141.86
		eval-div-rf	71.48
		BTA-rf	32.59
		BTZ_2*-rf*	26.10
		dcmp-rf	25.55
		*dcmp-div-rf(*3,19*)*	18.38

Table 3.2.: Comparison of the average root-finding algorithm performance on an x86 Intel(R) Core(TM)2 Duo CPU U7600 for code parameters as suggested in [52]. All given values are the average of 50 decryptions.

parameters	root-finding algorithm	running time / 10^5 cycles
$n = 2048$, $t = 32$	*BTA-rf*	3.16
	dcmp-rf	3.21
$n = 4096$, $t = 21$	*BTA-rf*	1.47
	dcmp-rf	5.34
$n = 8192$, $t = 18$	*BTA-rf*	1.28
	dcmp-rf	10.10

Table 3.3.: Comparison of the root-finding algorithm performance on an x86 Intel Intel(R) Core(TM)2 Duo CPU U7600 for code parameters with large n and small t.

3.4. Comparison of the McEliece and Niederreiter PKCs in Terms of Efficiency

In this section, we compare the McEliece and the Niederreiter scheme, where we consider results from the previous sections as well as recent work by other authors.

3.4.1. Public Key Size and Encryption Speed

For public keys $G_p \in \mathbb{F}_2^{n \times k}$ and $H_p \in \mathbb{F}_2^{mt \times n}$ that are not brought to reduced row echelon form, the Niederreiter public key is smaller, but for the usually much more reasonable choice of the reduced public keys, both public keys are in $\mathbb{F}_2^{mt \times k}$.

The Niederreiter scheme also features smaller encryption timings. In the matrix-vector multiplication, both the matrix and the vector have the same dimensions in both schemes. But in the Niederreiter scheme, a vector of the low Hamming weight given by t is employed, whereas that of the McEliece scheme has a Hamming weight of $n/2$ on average [15], which is much larger than t for usual code-parameter choices. In the same work, it is shown that the additional constant-weight-word encoding that is needed for the encryption in the Niederreiter PKC has only a small impact on encryption time. In that work, a speed advantage of factor 112 is claimed for the Niederreiter PKC over the McEliece PKC.

3.4.2. Private Key Size and Decryption Speed

In case of the choice of a McEliece public key in reduced row echelon form, the private key does not need to contain the scrambling matrix T because the message is reproduced (though certainly distorted by the error vector) in the first k positions of the ciphertext. For the Niederreiter system, an analogous approach is not possible. Thus, in terms of memory efficiency, the McEliece PKC has the advantage that it allows for a much smaller private key size than the Niederreiter PKC: if the parity check matrix is excluded from the McEliece private key as discussed in Section 3.2, then this key is smaller than a Niederreiter private key by exactly the size of the scrambling matrix $T \in \mathbb{F}_2^{mt \times mt}$.

In [15], the Niederreiter decryption uses a matrix T that is generated from a pseudo random number generator (PRNG), reducing the memory demands of this matrix down to only a small seed for the PRNG. However, the public key cannot then be chosen to be in reduced row echelon form, which is a major efficiency drawback.

In [15], the decryption time of the Niederreiter PKC is about three times lower than that of the McEliece PKC. The main reason for this is that in the Niederreiter decryption, the syndrome computation does not have to be carried out. However, they obviously compare both a Niederreiter and a McEliece implementation not using public keys in reduced row echelon form, otherwise the speed advantage of the Niederreiter PKC must be assumed to be lower: for the reduced public keys, in the McEliece PKC, the multiplication by the scrambling matrix T does not have to be carried out at all; thus, in this case the only speedup of the Niederreiter decryption stems from the fact that it features a matrix multiplication $T^{-1} \in \mathbb{F}_2^{mt \times mt}$ by vector in $\mathbb{F}_2^{mt}$ in contrast to

n, t, wt $(\vec{e})$	security level	plaintext size in bits		ciphertext size in bits	
		McEliece	Niederreiter	McEliece	Niederreiter
2048, 27, 27	80	1751	202	2048 (1.17)	297 (1.47)
2048, 50, 50	102	1498	333	2048 (1.37)	550 (1.65)
2960, 56, 57	128	2288	401	2960 (1.29)	672 (1.68)
6624, 115, 117	256	5129	842	6624 (1.29)	1495 (1.78)

Table 3.4.: Comparison of plaintext and ciphertext sizes of the McEliece and Niederreiter scheme. wt $(\vec{e})$ denotes the number of errors used during encryption. In the column with the ciphertext sizes, the message expansion, i.e the ratio of ciphertext and plaintext size, is given in brackets.

$H \in \mathbb{F}_2^{mt \times n}$ being multiplied by $\vec{z} \in \mathbb{F}_2^n$ in the McEliece decryption. Thus the ratio of the cost of the former multiplication to that of the latter is mt/n.

3.4.3. Message and Ciphertext Sizes

In the McEliece PKC, we have messages $\vec{m} \in \mathbb{F}_2^k$; in the Niederreiter PKC, the message size depends on the algorithm used for the constant-weight-word encoding. In Table 3.4, we give the message and ciphertext sizes of both cryptosystems for a number of parameter sets taken from [15, 52, 4], where the two highest parameter sets assume list decoding [53], and thus allow for the use of more than t errors during encryption. For the determination of the Niederreiter plaintext size, the encoder used in [14] was assumed. It yields a plaintext bit size of $l \geq \lfloor \log_2 \binom{n}{t} \rfloor - 1$.

We consider hybrid encryption, i.e. the encryption of the payload data through a symmetric encryption scheme and the encryption of the symmetric key by the public key scheme, as the most important application of public key schemes. It is obvious that, under this assumption, the Niederreiter PKC has a considerable advantage in the ciphertext size, while the plaintext size is always large enough for the encryption of symmetric keys that have a size appropriate for the respective public key security level.

4. Side Channel Security

In this chapter, we investigate side channel and fault attack vulnerabilities of code-based cryptosystems. In Section 4.1, we demonstrate the threat of message-aimed attacks against the code-based decryption and propose appropriate countermeasures. Afterwards, in Section 4.2, we devise two key-aimed attacks against code-based cryptosystems. For both types of attack, we give practical implementations and propose countermeasures. The last type of attack against code-based cryptosystems that we investigate in Section 4.3 are fault attacks. In Section 4.4, we show that all these attacks that we present for the McEliece PKC are transferable to the Niederreiter PKC. Finally, in Section 4.5, we investigate the relations of message-aimed timing and fault attacks and countermeasures between the McEliece and RSA schemes.

4.1. Message-aimed Side Channel Attacks against the Decryption Operation

In this section, we develop message-aimed side channel attacks against code-based cryptosystems, i.e. attacks that aim at recovering the message to a given ciphertext by gathering side channel information about the decryption operation. The observation that all of these attacks base on is as follows. From (2.1), it is apparent that the following connection between the number of errors w in a ciphertext and the degree of the error locator polynomial $\sigma(Y)$ is given: if $w \leq t$, then $\deg(\sigma(Y)) = w$. Otherwise, with high probability, if $w > t$, then $\deg(\sigma(Y)) = t$ as experimental results show, and never larger. The probability for lower degrees, in this case, is given approximately through the probability of the respective number of coefficients among σ_t, σ_{t-1}, etc. becoming zero when each is chosen randomly and equally distributed from $\mathbb{F}_{2^m}$.

This observation leads to two timing vulnerabilities in the syndrome decoding: the first, presented in Section 4.1.1, is given in the root-finding step [1]; the second, given in Section 4.1.2, in the syndrome decoding EEA [2]. For the latter, a related power analysis attack [6] is described in Section 4.1.3. Finally, in Section 4.1.4, we present a further timing vulnerability for a specific choice of the root-finding algorithm [5].

4.1.1. Timing Vulnerabilities in the Root-Finding based on the Degree of the Error Locator Polynomial

The first message-aimed attack we develop exploits a trait of the root-finding algorithm. The most straightforward root-finding variant is the exhaustive evaluation of the error locator polynomial $\sigma(Y)$ for each support element (refer to Section 2.1 for the role of

the support elements), given by *eval-rf* presented in Section 3.3.2.1. In the following, we assume this algorithmic choice. There, the evaluation is done with the Horner scheme. The running time of this operation depends linearly on the degree of $\sigma(Y)$. This gives us the timing effect that will be exploited in the attack: for larger degrees of $\sigma(Y)$, the time taken by the decryption operation is larger than for smaller degrees of this polynomial. Since for each of the n support elements t multiplications and additions in $\mathbb{F}_{2^m}$ are carried out, a considerable timing difference results for the evaluation of $\sigma(Y)$ of different degrees. According to the explanations in Section 4.1, the degree of $\sigma(Y)$ is closely related to the weight w of the error vector.

The general idea of the attack that exploits this timing behaviour is as follows: the attacker holds a ciphertext he wishes to decrypt. He inputs manipulated versions of the ciphertext, which differ from the original only in a single bit, into the decryption device and measures the decryption time. From the timing, he deduces whether, at this bit position, the corresponding bit in the error vector had value one or zero.

The attack is specified in Algorithm 8. There, sparse_vec (i) denotes the vector with zeros as entries, except for the i-th position having value 1, and the first position being indexed by 0. The starting point is that the attacker wishes to decrypt the ciphertext z. He subsequently creates n manipulated versions z'_i of the ciphertext z by flipping a single bit in each. The position where he performs the bit flip is i. He inputs this ciphertext into the decryption device N times and takes the average of the resulting timings. Then, he finds the guesses for the error positions in the original ciphertext as the t indexes i, which are associated with the t lowest timings.

The attack works on the base of the following reasons: If in a manipulation of the ciphertext he chooses an error position, i.e. $e_i = 1$ in the error vector used during encryption, the bit flip will reduce the error weight w by one, causing the degree of $\sigma(Y)$ to be $t-1$. Otherwise, the degree of $\sigma(Y)$ will be with high probability equal to t. By virtue of the above analysed timing effects concerning the degree of $\sigma(Y)$, he can, in this way, determine for every position i whether $e_i = 1$ or $e_i = 0$. This allows him to find the error vector $\vec{e}$ used during encryption. Subsequently, he can recover the message $\vec{m}$ by solving $\vec{z} = \vec{m}G \oplus \vec{e}$.

Algorithm 8 Timing Attack against the evaluation of $\sigma_{\vec{e}}(X)$

Input: ciphertext $\vec{z}$, and the parameter t of the McEliece PKC.
Output: a guess $\vec{e}'$ of the error vector $\vec{e}$ used by Alice to encrypt $\vec{z}$.
1: **for** $i = 0$ to $n-1$ **do**
2: Compute $\vec{z}_i = z \oplus$ sparse_vec (i).
3: Take the time u_i as the mean of N measured decryption times where $\vec{z}_i$ is used as the input to the decryption device.
4: **end for**
5: For the t smallest timings u_i put the corresponding indexes i into the set M.
6: **return** the vector $\vec{e}'$ with entries $e'_i = 1$ when $i \in M$ and all other entries as zeros.

The attack was carried out against the Java McEliece implementation in the Flexi-

Hamming distance of e' and e	success probability of the attack
0	48%
2	77%
4	96%
6	99%
8	99%
10	100%
12	100%
14	100%
16	100%
18	100%
20	100%
22	100%
24	100%

Table 4.1.: Experimental results of the attack according to Algorithm 8 against the FlexiProvider [54] Java implementation without countermeasures. The left column shows the upper bound of the number of wrongly determined error positions in the reconstructed error vector $\vec{e}'$ in contrast to the actual error vector $\vec{e}$. The right column gives the probability for this upper bound to be met for a single attack.

Provider [54]. There, the decryption operation was called from the attack program and the time taken by the operation was determined by the time functions provided by the Java Virtual Machine. Table 4.1 shows the results of the attack in terms of success probabilities. Here, the value of the attack parameter $N = 2$ was used. The first entry gives the probability 48% for the attack to completely determine the correct error vector. Thus, the main result is that the attack successfully finds the correct error vector with probability about one half.

A countermeasure against this vulnerability could simply be given through artificially raising the degree of $\sigma(Y)$ before the root-finding. However, there are further sources of timing differences occurring at earlier stages of the decryption procedure that can be exploited for an analogous attack, this will be subject of the following section. As the countermeasure to protect against these further vulnerabilities also removes the above explained vulnerability in the root-finding, there is no need for the realization of this manipulation of $\sigma(Y)$.

4.1.2. Timing Vulnerability of the Key Equation solving EEA and Countermeasures

In this section, we show that even in the absence of the timing vulnerability in the root-finding algorithm presented in the previous section, there is a remaining vulnerability in a previous part of the decryption algorithm yielding smaller timing differences.

This vulnerability allows the same attack as the one presented in Section 4.1.1, because apart from the quantative differences, the timing effect is the same as that explained in Section 4.1.1. Specifically, the vulnerability is located in the key equation solving EEA, given in Algorithm 9. After the theoretical analysis of the vulnerability, we explain countermeasures and experimental results.

4.1.2.1. Identification of the Vulnerability

Algorithm 9 The Extended Euclidean Algorithm (EEA($r_{-1}(Y)$, $r_0(Y)$, d))

Input: the polynomials $r_{-1}(Y)$ and $r_0(Y)$, with $\deg(r_0(Y)) < \deg(r_{-1}(Y))$

Output: two polynomials $r_N(Y)$, $b_N(Y)$ satisfying $r_N(Y) = b_N(Y)r_0(Y) \mod r_{-1}(Y)$ and $\deg(r_0(Y)) \leq \lfloor \deg(r_{-1})/2 \rfloor$

```
1: b_{-1} ← 0
2: b_0 ← 1
3: i ← 0
4: while deg(r_i(Y))) > d do
5:    i ← i + 1
6:    (q_i(Y), r_i(Y)) ← r_{i-2}(Y)/r_{i-1}(Y) // polynomial division with quotient q_i(Y) and
      remainder r_i(Y)
7:    b_i(Y) ← b_{i-2}(Y) + q_i(Y)b_{i-1}(Y)
8: end while
9: N ← i
10: return (r_N(Y), b_N(Y))
```

The timing vulnerability that we demonstrate in this section is based on the connection between the Hamming weight of the error vector and the number of iterations of the key equation solving EEA, which is invoked in the McEliece decryption from the Patterson Algorithm, i.e. Algorithm 1, in Step 4. The number of iterations in turn is naturally correlated with the algorithm's execution time, which builds the basis of the attack.

The key equation solving EEA is invoked with the polynomials $g(Y)$ and $\tau(Y)$ and $d = \lfloor \frac{t}{2} \rfloor$ as parameters. According to the break condition specified by d and the connection $\prod_{i=0}^{w-1}(\alpha_{E_i} \oplus Y) = \sigma(Y) = a(Y)^2 \oplus Yb(Y)^2$, with $\{E_i | i \in \{1, \ldots, w\}\}$ being the indexes of the bits having value one in the error vector, we have

$$\deg(a(Y)) \leq \left\lfloor \frac{w}{2} \right\rfloor \text{ and} \tag{4.1}$$

$$\deg(b(Y)) \leq \left\lfloor \frac{w-1}{2} \right\rfloor , \tag{4.2}$$

if $w \leq t$.

We now analyse how the value of w influences the number of iterations N executed in the key equation solving EEA, which, as already mentioned, certainly influences the timing of this operation. To this end, we first assume that in each iteration the degree of the

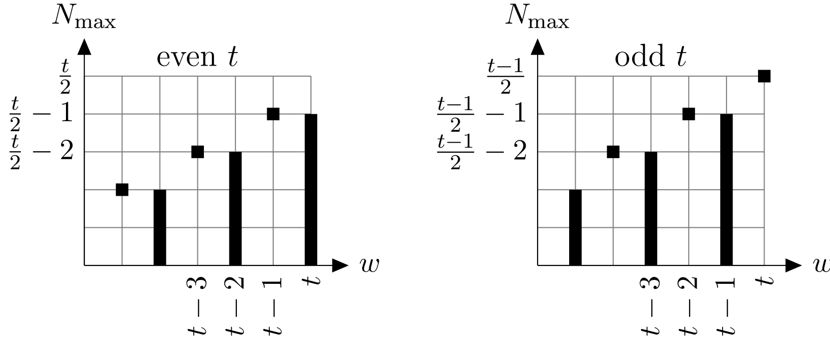


Figure 4.1.: Possible EEA iteration counts as a function of the error vector weight under the assumption that all quotient polynomials $q_i(Y)$ have degree one.

quotient $q_i(Y)$ is one, which happens with a probability of about $P_{N_{\max}} = (1-2^{-m})^{N_{\max}}$, where $N_{\max}$ is the number of iterations carried out in the case that all quotients have degree one. This probability is derived from the assumption that the coefficients of the remainders are random and equally distributed chosen from $\mathbb{F}_{2^m}$. Then, in each iteration, the probability for the reduction of the degree of the remainder by one is $(1-2^{-m})$. For McEliece parameters $m = 11$ and $t = 50$, for instance, we have $P_{N_{\max}} = 98.83\%$. Accordingly, we find

$$\deg\left(b(Y)\right) = \sum_{i=1}^{N_{\max}} 1 = N_{\max} = \left\lfloor \frac{w-1}{2} \right\rfloor . \tag{4.3}$$

This means that, for odd t, reducing the number of errors w from t to $t-1$ through a bit flip on the ciphertext by the attacker, just as described in Section 4.1.1, leads to $N_{\max,t-1} = \frac{t-3}{2}$ in contrast to $N_{\max,t} = \frac{t-1}{2}$ for the original ciphertext.

For an even value of t, the error weight has to be reduced by at least two to produce a reduced value of $N_{\max}$. The connection between the error weight and $N_{\max}$ for odd and even values of t is given in Figure 4.1. The reason for the difference between these two cases is that for $N_{\max} = \deg\left(b(Y)\right)$, odd t and $w = t$, it is $b(Y)$ that yields the leading coefficient of $\sigma(Y)$. Thus, a reduction of the error weight to $w = t-1$ implies that the degree of $b(Y)$ is reduced by at least one, since now $a(Y)$ must determine the leading coefficient. For even t and $w = t$, it is $a(Y)$ that provides the leading coefficient of $\sigma(Y)$, and reducing w to $t-1$ causes $b(Y)$ to take on this role, but according to $\sigma(Y) = a(Y)^2 \oplus Yb(Y)^2$, its degree will not be reduced.

We now analyse the problem of the attacker to create ciphertexts with an error weight of $t-2$ in the course of the attack. Clearly, the probability to produce ciphertexts of an error weight $w = t-x$ from an original ciphertext with t errors through random bit flips becomes smaller with growing x. Table 4.2 gives some example values showing that for the exemplary parameter set the attacks against McEliece implementation with even t are still feasible.

w	Probability	Example: $t = 50$ and $n = 2048$
$t-1$	$\frac{t}{n}$	2.44%
$t-2$	$\frac{t}{n}\frac{t-1}{n-1}$	0.06%
$t-3$	$\frac{t}{n}\frac{t-1}{n-1}\frac{t-2}{n-2}$	0.0026%

Table 4.2.: Probability to create manipulated ciphertexts with a given w.

4.1.2.2. Timing Countermeasure

The countermeasure to protect against this vulnerability is realized by detecting and preventing a premature abortion of the EEA, which can only happen if $w < t$. For even values of t, this is achieved by ensuring that $\deg(a(Y)) = d_{\text{break}}$. If its degree is lower, i.e. $\deg(r_i(Y)) < t$ and Algorithm 9 would terminate, then $r_i(Y)$ is manipulated such that $\deg(r_i(Y)) = \deg(r_{i-1}(Y)) - 1$, which would be the most probable progression of the algorithm in the case $w = t$.

In the case of odd t, it has to be ensured that $N_{\max} = \deg(b(Y)) = d_{\text{break}}$, as then the attacker gets no information about the error weight. Again, if the degree of the current remainder $r_i(Y)$ is lower than d_{break}, i.e. the termination condition for Algorithm 9 is fulfilled, but at the same time $\deg(b_i(Y)) < d_{\text{break}}$, then the current $r_i(Y)$ is manipulated in the same way as described for even t. The EEA with these countermeasures is given in Algorithm 10.

The reason for the validity of the approach for odd t is given by the following observations: $\deg(r_i(Y)) \leq d_{\text{break}}$ is the condition for the last iteration. If in the last iteration $\deg(b_i(Y)) < d_{\text{break}}$, we would have $\deg(\sigma(Y)) < t$, in this case the degree of the remainder is manipulated as in the case of even values of t. The reason that this enforces $N_{\max}$ iterations (ignoring any potentially previous random skipping of iterations) is based on the connection between $\deg(r_i(Y))$ and $\deg(b_i(Y))$ that is enforced by the EEA. Any change of the degree of the remainders $\deg(r_i(Y)) = \deg(r_{i-1}(Y)) - x$ leads to $\deg(q_{i+1}(Y)) = x$, and consequently to $\deg(b_{i+1}(Y)) = \deg(b_i(Y)) + x$. Thus, only the last iteration can cause a decrease of the degree of $a(Y) = r_N(Y)$ without a corresponding increase of the degree of $b(Y) = b_N(Y)$. Accordingly, if $\deg(b_i(Y)) < d_{\text{break}}$, then $\deg(r_{i-1}(Y)) > d_{\text{break}}$, which shows that the countermeasure proposed for odd t works, since then successive decrements of $\deg(r_i(Y))$ lead to $\deg(b_N(Y)) = d_{\text{break}}$.

4.1.2.3. Implementation and Verification of the Countermeasure

The attack and the effectiveness of the countermeasures were experimentally verified on an FPGA implementation. Specifically, the McEliece decryption for code parameters $n = 2048$ and $t = 50$ was implemented on a Virtex-5 Xilinx FPGA. To assess the running time dependency on the error weight w, the following test ciphertexts were created: from an original ciphertext with $w = 50$ the ciphertexts z_1 with $w = 49$ and z_2 with $w = 48$ were created by flipping the corresponding number of error positions. The decryption timings that were found for these three ciphertexts with and without the

Algorithm 10 EEA with countermeasures against message-aimed attacks

Input: $\tau(Y)$, $g(Y)$
Output: $a(Y)$ and $b(Y)$, with $b(Y)\tau(Y) = a(Y) \mod (g(Y))$ and $\deg(a(Y)) \leq d_{\text{break}}$

1: $r_{-1}(Y) = g(Y)$
2: $r_0(Y) = \tau(Y)$
3: $b_{-1}(Y) = 0$
4: $b_0(Y) = 1$
5: $i = 0$
6: **while** $\deg(r_i(Y)) > d_{\text{break}}$ **do**
7: $\quad i = i + 1$
8: $\quad q_i(Y) = r_{i-2}(Y)/r_{i-1}(Y)$
9: $\quad r_i(Y) = r_{i-2}(Y) \mod r_{i-1}(Y)$
10: $\quad b_i(Y) = b_{i-2}(Y) + q_i(Y) \cdot b_{i-1}(Y)$
11: $\quad$ **if** t even **then**
12: $\quad\quad$ **if** $\deg(r_i(Y)) < d_{\text{break}}$ **then**
13: $\quad\quad\quad$ Manipulate $r_i(Y)$, so that $\deg(r_i(Y)) = \deg(r_{i-1}(Y)) - 1$
14: $\quad\quad$ **end if**
15: $\quad$ **else**
16: $\quad\quad$ **if** $\deg(r_i(Y)) \leq d_{\text{break}}$ AND $\deg(b_i) < d_{\text{break}}$ **then**
17: $\quad\quad\quad$ Manipulate $r_i(Y)$, so that $\deg(r_i(Y)) = \deg(r_{i-1}(Y)) - 1$
18: $\quad\quad$ **end if**
19: $\quad$ **end if**
20: **end while**
21: $a(Y) = r_i(Y)$
22: $b(Y) = b_i(Y)$
23: **return** $a(Y)$ and $b(Y)$

$\vec{z}$	w	cycles w/o countermeasure	cycles w/ countermeasure
$\vec{z}$	t	189,138	189,138
$\vec{z}_1$	$t-1$	189,138	189,138
$\vec{z}_2$	$t-2$	188,836	189,138

Table 4.3.: Timings for the decryption on the FPGA implementation with code parameters $n = 2048$, $t = 50$ and ciphertexts with different error weights.

countermeasure described in Section 4.1.2.2 are given in Table 4.3. Because of t being even, the error weight w must be reduced by two in order to achieve a decrease of the timing in the unprotected version. The results show the vulnerability of the unprotected version with respect to the timing attack developed in Section 4.1.1. The protected version has constant timings for all three ciphertexts. Accordingly, the vulnerability is not present in this version.

4.1.3. A related Simple Power Analysis Attack against the Key Equation Solving EEA

Any timing attack based on the distinction of different control flows can be extended to a simple power analysis attack if it is possible to distinguish the control flows in the power trace. This means that it is not sufficient for a countermeasure to simply ensure unambiguous overall timings of the total cryptographic operation if power analysis attacks shall be thwarted as well. In the following, a simple power analysis attack against an FPGA implementation is developed based on the vulnerability explained in Section 4.1.2.1. After explaining the measurement setup, we give the experimental results of the attack, and finally devise appropriate countermeasures, the effect of which is again demonstrated in an experiment.

4.1.3.1. Measurement Setup

For the simple power analysis attack, the relevant part of the McEliece decryption, i.e. the EEA, was implemented on a SASEBO-G board [55], which was specifically developed for the purposes of performing power analysis attacks against implementations on the Virtex-II P30 FPGA that is installed on this board. The restriction of the implementation to only the relevant parts is due to the fact that the complete McEliece decryption with the parameters $n = 2048$ and $t = 50$ could not be realized on the limited resources of the board's FPGA. The FPGA's clock rate was set to 50 MHz.

The power measurement is performed by measuring the voltage drop along a 1Ω resistor connected to the FPGA's power supply with a digital sampling oscilloscope.

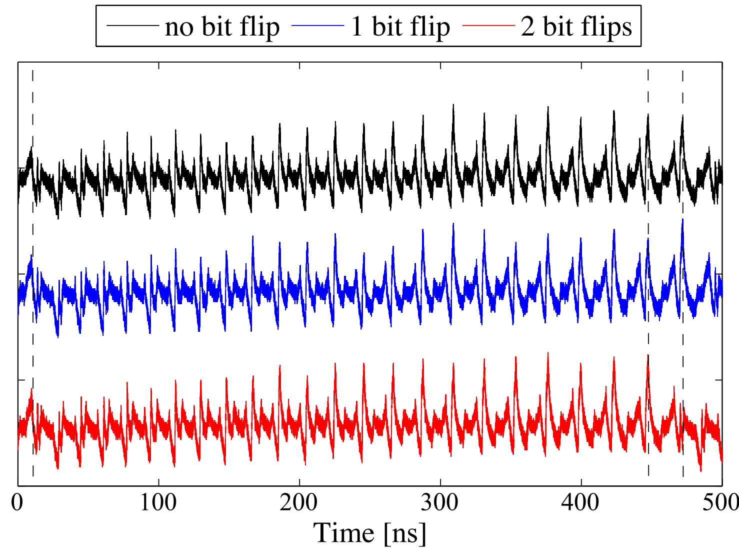


Figure 4.2.: Power Traces for the XGCD Implementation without Countermeasure

4.1.3.2. Attacks against the insecure Implementation

The attack is executed by inputting different polynomials $\tau(Y)$ into the EEA. This is done with the help of the Matlab-based control software. Specifically, three different polynomials $\tau(Y)$ corresponding to decryptions of ciphertexts with three different Hamming weights: $w = t$, $w = t - 1$ and $w = t - 2$ are set as input data to the EEA, as well as the corresponding Goppa polynomial $g(Y)$. The execution of the EEA with these input polynomial $\tau(Y)$ corresponds to the ciphertext inputs during the attack described in Section 4.1.2. Then, the control software starts the EEA execution as well as the recording of the power trace via the oscilloscope.

The experimental results are shown in Figure 4.2. The results are in line with the theoretical analysis of Section 4.1.2.1 and the results of the timing attack given in Section 4.1.2.3: for $w = t$ and $w = t-1$ the number of iterations, which is obviously equal to the number of peaks in the trace before the drop of the height of the peaks, is 24, while it is smaller by one for $w = t-2$. Since an attacker is able to distinguish these different error weights, he is able to conduct a message-aimed attack in the same manner as described in Section 4.1.2.1.

4.1.3.3. Countermeasure

The fundamental approach of the countermeasure is equal to the one protecting against the underlying timing vulnerability, given in Algorithm 10, but to achieve security against power analysis attacks, it is important to ensure that all operations are executed unconditionally.

As in the timing attack presented in Section 4.1.2.1, the analyzed implementation features an even value of t. From a high level point of view, the countermeasure, as in the case of the timing attack countermeasure, works as follows: When it is found that $\deg(r_i(Y)) < d_{\text{break}}$, which, in this case, indicates an improper ciphertext, all coefficients of $r_i(Y)$ are set to non-zero values so that $\deg(r'_i(Y)) = \deg(r_{i-1}(Y)) - 1$ and the proceeding of the algorithm is ensured.

The detailed implementation of the countermeasure is shown in Figure 4.3. The unprotected implementation is given by the part with the white background including the dashed line indicating the direct feedback of $r_i(Y)$ into the next iteration as $r_{i-1}(Y)$. In each iteration, $r_{i-1}(Y)$ and $r_{i-2}(Y)$ are divided by the GOPF DIV unit, which outputs the remainder $r_i(Y)$ as well as the quotient $q_i(Y)$, which is further processed in the update of the value of $b_i(Y)$.

In the protected implementation however, which additionally features the area with the blue background, the direct feedback of $r_i(Y)$ denoted by the dashed line is not realized. Instead, the degree of $r_i(Y)$ is determined, and, in case of being smaller than d_{break}, an improper ciphertext is detected, and $r_i(Y)$ is modified to $r'_i(Y)$ by setting all coefficients to predetermined values as described above. This is done in the register in the blue countermeasure area of Figure 4.3, which otherwise contains $r_i(Y)$, i.e. if no improper ciphertext is detected. In the case that the attack is detected in the last iteration (i.e. $\deg(r_i(Y)) = d_{\text{break}}$), $r'_i(Y)$ is output directly.

To verify the countermeasure's effectiveness, the power measurements were repeated with the same input data. The result is depicted in Figure 4.4. The difference in the number of peaks allowing to deduce the number of iterations is not given anymore.

4.1.4. Vulnerability in Root-Finding with exhaustive Evaluation and Division

In this section, we present a new timing vulnerability that is given for a special choice of the root-finding algorithm. In Section 4.1.1, it was explained that in the case of $\text{wt}(\vec{e}) < t$, the error locator polynomial $\sigma(Y)$ will have a degree smaller than t, which might be revealed through the timing in an unsecured implementation of the McEliece decryption, where timing differences occur in both the Patterson Algorithm and the root-finding algorithm. We have also explained that it is known how to achieve security against these threats.

In this section, however, we want to point out another potential vulnerability in the root-finding algorithm concerning the message-aimed attacks against the McEliece cryptosystem. Specifically, the problem arises when this algorithm involves factoring the error locator polynomial. An example for this is *eval-div-rf* given by Algorithm 6 in Section 3.3.2.1. There, an exhaustive search for the roots in $\mathbb{F}_{2^m}$ is improved in terms of running time by dividing the polynomial by each root that was found. We show that with this algorithmic choice significant timing differences exist between the cases $w = t$ and $w > t$. To this end, we analyse the behaviour of the degree of the error locator polynomial for these two cases.

As already discussed in Section 4.1, for $w < t$, this degree will be equal to w. On the other hand, as experimental results show, the error locator polynomial output by the

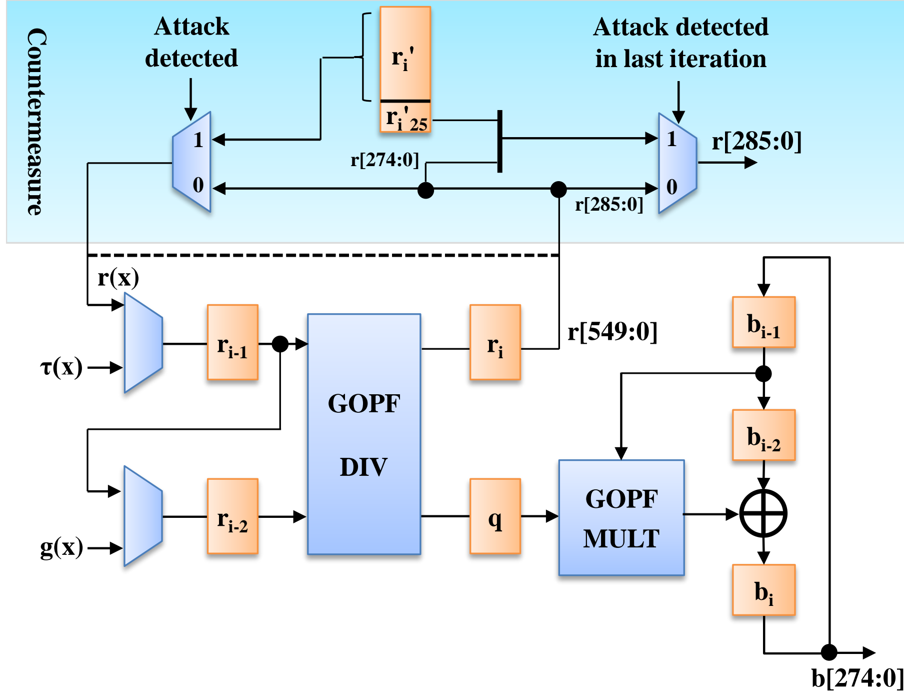


Figure 4.3.: Datapath of the EEA Implementation with Countermeasure

Patterson Algorithm in the case that more than t errors occurred, has much fewer than t roots (typically less than five in our tests with parameters $n = 2048$, $t = 50$). This extends the running time of the root-finding algorithm in the case the attacker flipped a bit at a non-error position (i.e. $w = t + 1$), since much fewer divisions occur then, and, accordingly, the average degree of the error locator polynomial for the n evaluations is considerably higher.

We performed experiments with a proprietary implementation of the McEliece decryption written in the "C" programing language without any of the countermeasures against the message-aimed attacks from Sections 4.1.1 and 4.1.2 on a personal computer. We used the code parameters $n = 2048$ and $t = 50$. The result shown in Figure 4.5 depicts the mean running time of the decryption operation as a function of the weight of the error vector $\vec{e}$. For this figure, 100,000 measurements of the decryption of the same ciphertext were used for each error weight on the horizontal axis. The standard deviation of the respective data sets is given as error bars. The results where obtained on an Intel Core Duo T7300 running Linux, the code was compiled with GCC 4.2.4. For the domain $\mathrm{wt}(\vec{e}) \leq t$, we see the influence of the previously known vulnerabilities mentioned in Sections 4.1.1 and 4.1.2. The effects of the problems introduced by Algorithm 6 manifest themselves as an increased running time when $\mathrm{wt}(\vec{e}) > t = 50$. Clearly, an attacker is able to distinguish whether a bit flip introduced by him on the ciphertext was an error position or not based on the timing.

Note that the case of $w < t$ is covered by the countermeasures proposed in Section

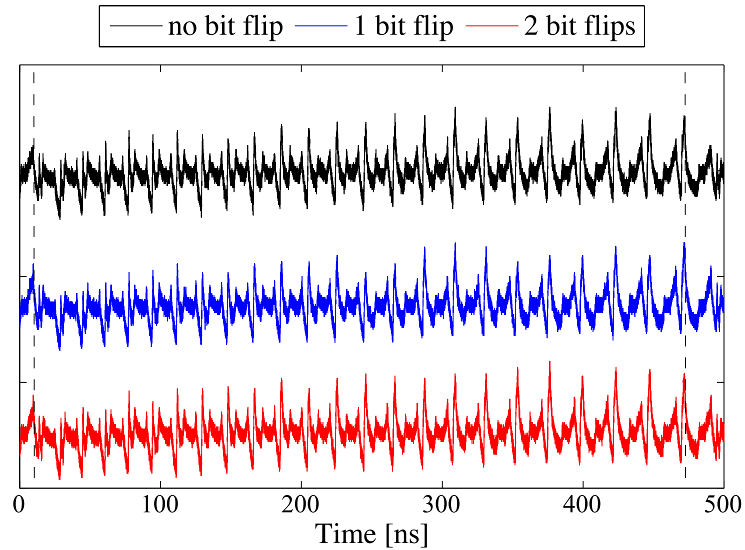


Figure 4.4.: Power Traces for the EEA Implementation with Countermeasure

4.1.2.2, i.e. it is rendered indistinguishable from the case $w \geq t$. In the presence of these countermeasures, the decryption of a ciphertext with $\mathrm{wt}(e) < t$ also results in $\sigma(Y)$ with degree t and very few roots because of the pseudorandom manipulation of the remainder in the EEA. But it is important to be aware that even if $w < t$ and $w > t$ are not distinguishable based on the timings, but $w = t$ can be distinguished from $w \neq t$, an attack is still possible: by flipping two bits in a ciphertext and trying to find those cases where $w = t$, the attacker will learn whenever he flipped one non-error and one error position.

As a consequence, in a secure implementation of the McEliece decryption, in addition to the countermeasures regarding the Patterson Algorithm, it is required to avoid the introduction of dependencies of the root-finding algorithm's running time on the error weight. This can for instance be achieved by simply leaving out the division in Step 5 of Algorithm 6.

4.2. Side Channel Attacks against the secret Support

In the following, we investigate timing vulnerabilities that allow attacks against the secret key. The first attack, given in Section 4.2.1, exploits vulnerabilities in the two invocations of the EEA, i.e. the syndrome inversion in Algorithm 1, Step 2 and the solving of the key equation in Algorithm 1, Step 4 [9, 3]. The second type of vulnerability, presented in Section 4.2.2, is found in certain root-finding algorithm variants [7]. In both cases, the aim of the attack is to recover the secret support Γ. This is enough to recover the other part of the secret key, the Goppa Polynomial $g(Y)$, as well [18].

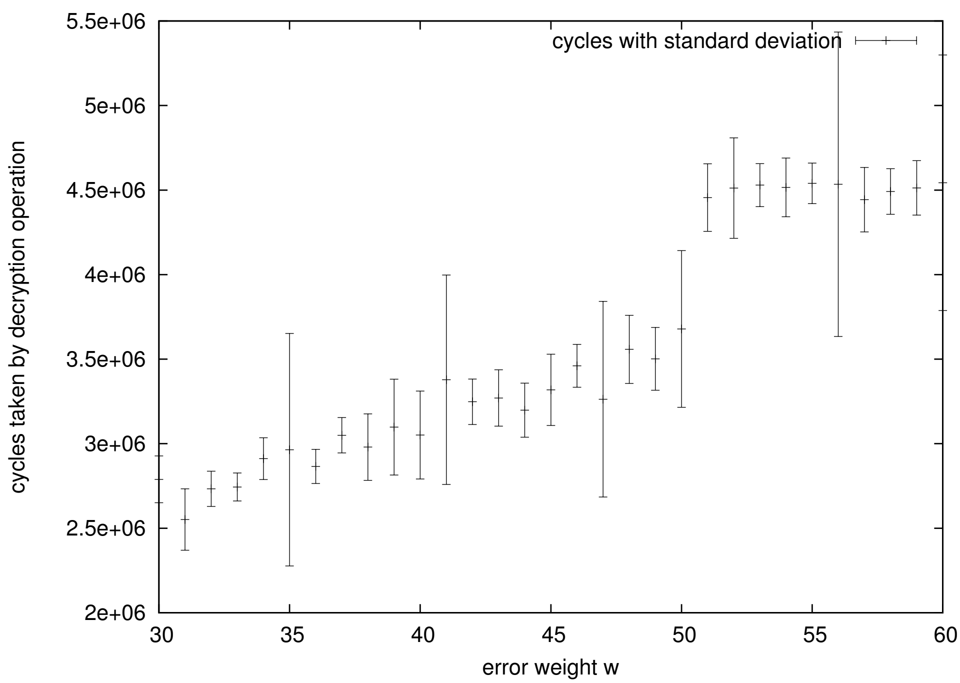


Figure 4.5.: Timings of the decryption operation for different error weights for an entirely unprotected McEliece implementation with parameters $n = 2048$ and $t = 50$. The error bars indicate the standard deviation.

4.2.1. Timing Attacks against the EEA

In this section, we develop a timing attack that targets the recovery of the secret support. The attack builds on three different vulnerabilities, which we analyse in the following sections. To this end, in the next section, we first explore certain general properties of the syndrome inversion that is part of the code-based decryption with the Patterson Algorithm. The syndrome inversion is achieved by the use of the EEA. Then, in Sections 4.2.1.2 and 4.2.1.3, we explore how these general properties lead to exploitable timing differences when the attacker prepares ciphertexts with fixed error weights $w = 4$ and $w = 6$, respectively. Specifically, through the leakage, the attacker learns certain equations about the elements of the secret support. While in these analyses we only considered the syndrome inversion, which is the first application of the EEA during the code-based decryption, Section 4.2.1.4 shows that the second application of the EEA, i.e. for the solving of the key equation, contributes to the increase of the found timing differences. Afterwards, in Section 4.2.1.5, we show a third vulnerability that reveals the zero element of the support. Consequently, Section 4.2.1.6 introduces an attack that exploits all three vulnerabilities. Section 4.2.1.7 presents experimental results that show the practicality of the attack.

To complete the analysis, in Sections 4.2.1.8, 4.2.1.9, we consider the effect of countermeasures against other timing attacks on the practicality of the new attack and discuss possible extensions of the attack. Finally, Section 4.2.1.10 addresses the problem of countermeasures against this attack.

For simplicity, we confine the analysis to the case $n = 2^m$.

4.2.1.1. Properties of the Syndrome Inversion

In this section, we determine a basic property of the syndrome inversion EEA that will be used in the subsequent sections to derive concrete timing vulnerabilities. To this end, we turn to the form of syndrome polynomial.

The syndrome polynomial is defined as

$$S(Y) \equiv \sum_{i=1}^{w} \frac{1}{Y \oplus \epsilon_i} \equiv \frac{\Omega(Y)}{\sigma(Y)} \mod g(Y) \tag{4.4}$$

Here, w is the Hamming weight of the error vector $\vec{e}$ and the $\{\epsilon_i | i \in \{1, \dots w\}\}$ denote the support elements associated with the indexes of those bits in the error vector having value one in arbitrary ordering. For instance, if the bits found at the index j and k in the error vector have value one, then $\epsilon_1 = \alpha_j$, $\epsilon_2 = \alpha_k$ and so on. The identification of the error locator polynomial $\sigma(Y)$ in the denominator is simply a result of the form of the common denominator of all sum terms. In the McEliece PKC Decryption, during the error correction, Algorithm 1, Step 3, $S^{-1}(Y)$ is computed by invoking Algorithm 9 as $\mathrm{EEA}(g(Y), S(Y), 0)$. But it is known that in case of $w \leq t/2$, instead, it is possible to find $\sigma(Y)$ already at this stage with the EEA:

Theorem 4.2.1. *Assume a Goppa Code defined by $g(Y)$ and Γ. For the Hamming weight w of the error vector associated with $S(Y)$ let $w \leq t/2$. When Algorithm 9 is invoked as*

EEA$(g(Y), S(Y), \lfloor t/2 \rfloor)$, *i.e. with* $r_{-1}(Y) = g(Y)$ *and* $r_0(Y) = S(Y)$ *and breaking once* $\deg(r_i(Y)) \leq (t/2)$ *then it returns* $\delta\sigma(Y) = b_M(Y)$ *and furthermore* $\delta\Omega(Y) = r_M(Y)$. *Here,* $\delta \in \mathbb{F}_{2^m}$ *and* M *is the number of iterations performed by the EEA .*

Proof. For the proof, see [56] or [33], Chapter 12, §9. □

Given this form of the $S(Y)$, we can make a statement about the maximally possible number of iterations in the EEA used to compute $S^{-1}(Y) \equiv \sigma(Y)/\Omega(Y) \mod g(Y)$. As already mentioned, the actual invocation of the syndrome inversion is EEA$(g(Y), S(Y), 0)$. But Theorem 4.2.1 shows that, in this case, we could stop at $\deg(r_i(Y)) \leq t/2$ to find $\sigma(Y)$. This means that there is one iteration in the EEA, where $r_i(Y) = \delta\Omega(Y)$ and $b_i(Y) = \delta\sigma(Y)$, in case of $w \leq t/2$.

Theorem 4.2.2. *Assume a Goppa Code defined by* $g(Y)$ *and* Γ. *When Algorithm 9 is invoked as* EEA$(g(Y), S(Y), 0)$ *with* $S(Y) \equiv \frac{\Omega(Y)}{\sigma(Y)} \mod g(Y)$, *and the error vector* $\vec{e}$ *corresponding to* $S(Y)$ *satisfies* $\mathrm{wt}(\vec{e}) \leq (\deg(g(Y))/2)$, *then for the number of iterations in Algorithm 9 we find:*

$$M \leq M_{\max} = \deg(\Omega(Y)) + \deg(\sigma(Y))$$

Proof. Consider the iteration where $r_j(Y) = \delta\Omega(Y)$ and $b_j(Y) = \delta\sigma(Y)$. Since, according to Algorithm 9, the degree of $b_j(Y)$, starting from zero increases at least by one in each iteration, we find $j \leq \deg(\sigma(Y))$. From the iteration j on, the degree of $r_j(Y) = \delta\Omega(Y)$ is decreased by at least one in each subsequent iteration down to $\deg(r_M(Y)) = 0$, i.e. $M - j \leq \deg(\Omega(Y))$, giving $M = M - j + j \leq \deg(\Omega(Y)) + \deg(\sigma(Y))$. □

Because, in the following, we are only interested in the derivation of equations of the form $\sigma_i = 0$ for a specific value of i, we will ignore the constant δ from here on.

4.2.1.2. Linear Equations from $w = 4$ Error Vectors

In this section, we demonstrate a connection between the number of iterations in the syndrome inversion EEA and the form of one of the coefficients of the error locator polynomial $\sigma(Y)$ for the specific case of ciphertexts with error weight $w = 4$. The number of iterations of this EEA invocation, in turn, serves as source of timing differences. Specifically, we will show that an attacker is able to learn linear equations about the support elements.

To this end, we first investigate the effect of the results from Section 4.2.1.1 for the case that ciphertexts created with error vectors of Hamming weight four are input to the decryption operation. In the case of $w = 4$, the syndrome polynomial is of the form:

$$S(Y) \equiv \frac{\Omega(Y)}{\sigma(Y)} \equiv \sum_{i=1}^{4} \frac{1}{Y \oplus \epsilon_i} \equiv \frac{\sigma_3 Y^2 \oplus \sigma_1}{Y^4 \oplus \sigma_3 Y^3 \oplus \sigma_2 Y^2 \oplus \sigma_1 Y \oplus \sigma_0} \mod g(Y), \quad (4.5)$$

where $\epsilon_i \in \mathbb{F}_{2^m}, i \in 1, \ldots, 4$ denote the four elements of the support associated with the error positions. Furthermore, in the right-hand side of (4.5), which is found by bringing

all four sum terms to their common denominator, we have

$$\sigma_3 = \epsilon_1 \oplus \epsilon_2 \oplus \epsilon_3 \oplus \epsilon_4,$$

$$\sigma_2 = \epsilon_2\epsilon_3\epsilon_4 \oplus \epsilon_1\epsilon_3\epsilon_4 \oplus \epsilon_1\epsilon_2\epsilon_4 \oplus \epsilon_1\epsilon_2\epsilon_3,$$

$$\sigma_1 = \epsilon_1\epsilon_2 \oplus \epsilon_1\epsilon_3 \oplus \epsilon_1\epsilon_4 \oplus \epsilon_2\epsilon_3 \oplus \epsilon_2\epsilon_4 \oplus \epsilon_3\epsilon_4,$$

$$\sigma_0 = \epsilon_1\epsilon_2\epsilon_3\epsilon_4.$$

With the aim of finding a timing vulnerability revealing certain coefficients of $\sigma(Y)$, and, thus, information about the secret support, we now analyze the connection between the number of iterations on the one hand and the degree of $\Omega(Y)$ on the other. Regarding $\Omega(Y)$ for the case $w = 4$, we find that the coefficient to the highest power of Y is given by $\sigma_3 = \epsilon_1 \oplus \epsilon_2 \oplus \epsilon_3 \oplus \epsilon_4$. If $\sigma_3 = 0$, then the degree of $\Omega(Y)$ is zero, otherwise it is two. According to Theorem 4.2.2, this means that in the case of $\sigma_3 = 0$, the maximal number of iterations in the inversion is four, in contrast to six in the general case. Under the assumption that always the maximal number of iterations occur, and under the additional assumption that the duration of a single iteration of the EEA is sufficiently large, the following vulnerability is given: Assume the attacker inputs random ciphertexts with $w = 4$. He can now determine those decryptions where $\sigma_3 = 0$ from the timing, as they coincide with a smaller M, i.e. fewer iterations of the EEA. Given that he knows which error positions he chose in preparation of the ciphertext, he now knows the equation $\sigma_3 = \epsilon_1 \oplus \epsilon_2 \oplus \epsilon_3 \oplus \epsilon_4 = \alpha_{j_1} \oplus \alpha_{j_2} \oplus \alpha_{j_3} \oplus \alpha_{j_4} = 0$ describing the secret support.

We now give arguments as to why the results are still valid when we drop the assumption that always the maximal number of iterations is executed. In the majority of the cases $M_{\max}$ iterations occur, i.e. six when $\deg(\Omega(Y)) = 2$ and four when $\deg(\Omega(Y)) = 0$. In this case, all the quotients $q_i(Y)$ have degree one. But, with probability about $1/n$ in each iteration, a larger degree of the quotient polynomial $q_i(Y)$ occurs (Section 4.1.2 already explored this behaviour of the EEA), accordingly, we have $M < M_{\max}$. With the aim of assessing the reliability of the differences in the running time allowing to identify the case $\deg(\Omega(Y)) = 0$, we examine whether $M < M_{\max}$ might lead to timings for $\deg(\Omega(Y)) = 2$ as low as for $\deg(\Omega(Y)) = 0$, which would complicate the timing attacks as then a reliable distinction of $\sigma_3 = 0$ and $\sigma_3 \neq 0$ would not be possible.

To this end, we will prove three theorems that hold only for the case of ciphertexts built with $w = 4$ error vectors. Taken together, they show that the by far most complex polynomial multiplication within the EEA execution is only executed when $\sigma_3 \neq 0$. In this course, we refer to the iteration where $r_j(Y) = \delta\Omega(Y)$ and $b_j(Y) = \delta\sigma(Y)$, discussed in Section 4.2.1.1, as the *intermediate* iteration and use the index j to identify it. Table 4.4 is given as an aid. It shows the degrees of the polynomials in each iteration of the EEA for the case $\sigma_3 \neq 0$ and the assumption that the maximal number of iterations is carried out.

First, we show that the following theorem holds:

Theorem 4.2.3. *In the case of $\sigma_3 = 0$, the intermediate iteration is the last iteration of the syndrome inversion EEA. Otherwise, if $\sigma_3 \neq 0$, at least one iteration is performed after the intermediate iteration.*

Proof. The first statement is immediately obvious, since in this case $\deg(\Omega(Y)) = \deg(r_j(Y)) = 0$, and thus the EEA is completed. The second one is true because in this case $\deg(\Omega(Y)) = \deg(r_j(Y)) = 2$, and the break condition of the EEA is not yet satisfied. □

Next, we show a result about the complexity of the multiplications:

Theorem 4.2.4. *Concerning the polynomial multiplication $b_{i-1}(Y)q_i(Y)$ that is carried out in each iteration of the EEA, for all iterations from the first up to and including the intermediate iteration, the maximally possible degree of the factors is 4.*

Proof. Note that $b_{i-1}(Y)$ grows at least by one in each iteration from 0 to the value 4 in the intermediate iteration. Thus, this factor can have at most degree four. On the other hand, $q_i(Y)$ cannot be larger than 3 in the iterations before, and including, the intermediate iteration: otherwise, $\deg(b_j(Y)) = \sum_{i=1}^{j} q_i(Y) > 4$, which would mean that the intermediate iteration was skipped and thus is impossible. □

Finally, we show a result about the high complexity of the multiplication in the iteration after the intermediate iteration:

Theorem 4.2.5. *The polynomial multiplication in the iteration after the intermediate iteration has one factor $q_{j+1}(Y)$ of degree $t - 6$ and the other factor $b_j(Y)$ of degree 4.*

Proof. The degree of $b_j(Y)$ must be four in the intermediate iteration because $\deg(\sigma(Y)) = 4$. It remains to prove the statement about $q_{j+1}(Y)$. To this end, we first show that $\deg(r_{j-1}(Y)) = t - 4$:

$$4 = \deg(b_j(Y)) = \sum_{i=1}^{j} \deg(q_i) = \deg(r_{-1}(Y)) - \deg(r_{j-1}(Y)) = t - \deg(r_{j-1}(Y)) \,. \tag{4.6}$$

Furthermore, it holds that $\deg(q_{j+1}(Y)) = \deg(r_{j-1}(Y)) - \deg(r_j(Y)) = (t-4) - \deg(r_j(Y))$. Now, since there is an iteration after the intermediate iteration, we must have the case $\sigma_3 \neq 0$; consequently, the only possible value for $\deg(r_j(Y))$ is 2: According to Theorem 4.2.1, $\deg(r_j(Y)) = \deg(\Omega(Y))$, which must be 2 for $\sigma_3 \neq 0$ according to (4.5). □

Taken together, Theorems 4.2.3, 4.2.4 and 4.2.5 show that the iteration with the complex multiplication is carried out if and only if $\sigma_3 \neq 0$. For realistic code parameters, $t-6$ is far greater than 4. This means that the complex polynomial multiplication is far more costly than any other prior polynomial multiplication in the same EEA invocation. Note that in a straightforward implementation, which is used in the attack scenario given later, the cost of the polynomial multiplication is linear in the degree of either factor.

i	$\deg(q_i(Y))$	$\deg(b_i(Y))$	$\deg(r_i(Y))$
1	1	1	$t-2$
2	1	2	$t-3$
3	1	3	$t-4$
4	1	**4**	**2**
5	$t-6$	$t-2$	1
6	1	$t-1$	0

Table 4.4.: Overview of the iterations in the syndrome inversion EEA for Hamming weight four error vectors for the case $\sigma_3 \neq 0$, i.e. $\deg(\Omega(Y)) = 2$, and the assumption that $M_{\max}$, the maximal number of iterations is carried out. The intermediate iteration is shown in bold.

This means that we can be confident to distinguish the cases $\sigma_3 = 0$ and $\sigma_3 \neq 0$ from the timing even if not the maximal number of iterations $M_{\max}$ is executed.

We do not show in detail that this cost, and thus the timing of the complex multiplication, is also large compared to the polynomial division, the other part of each EEA iteration. However, from Algorithm 11 given later in Section 4.2.1.5, it is apparent that the number of iterations in the division, and thus its running time grows with the difference in the degrees of the input polynomials. Again, it is the iteration after the intermediate iteration that features the most complex division, since there we divide the polynomials $r_{j-1}(Y)$ with degree $t-4$, and $r_j(Y)$ with degree 2, while for all prior divisions the input polynomials have a difference in the degree of approximately 1.

4.2.1.3. Cubic Equations from $w = 6$ Error Vectors

The vulnerability found for $w = 4$ error vectors can be generalized to any even value of w. For our attack, we also employ the case $w = 6$. There, we find that the syndrome polynomial according to (4.4) is of the form

$$S(Y) \equiv \frac{\Omega(Y)}{\sigma(Y)} \equiv \frac{\sigma_5 Y^4 \oplus \sigma_3 Y^2 \oplus \sigma_1}{Y^6 \oplus \sigma_5 Y^5 \oplus \sigma_4 Y^4 \oplus \sigma_3 Y^3 \oplus \sigma_2 Y^2 \oplus \sigma_1 Y + \sigma_0} \mod g(Y), \quad (4.7)$$

where

$$\sigma_3 = \sum_{j=3}^{6} \sum_{k=2}^{j-1} \sum_{l=1}^{k-1} \epsilon_j \epsilon_k \epsilon_l, \quad (4.8)$$

$$\sigma_5 = \sum_{i=1}^{6} \epsilon_i. \quad (4.9)$$

The vulnerability here is that an attacker can detect $\sigma_3 = 0$ and simultaneously $\sigma_5 = 0$ from the timing, which is lower in this case. Now, we introduce the assumption that $\sigma_5 = 0$ for all ciphertexts with $w = 6$. This is because in the attack that is presented in

Section 4.2.1.6, this condition is satisfied (it is achieved by using prior knowledge found through the other vulnerabilities for the creation of the $w = 6$ ciphertexts).

Again, the main reason for this timing effect is that in this case $\deg(\Omega(Y)) = 0$, and thus, according to Theorem 4.2.2 two iterations fewer are executed in the EEA than in the general case (where $\sigma_5 = 0$ and thus $\deg(\Omega(Y)) = 2$).

In the following, we show that it is again the most complex iteration, i.e. the one featuring the by far most complex polynomial multiplication of the syndrome inversion EEA that is skipped if $\deg(\Omega(Y)) = 0$. To this end, we adopt Theorems 4.2.3, 4.2.4 and 4.2.5 to the case of $w = 6$ and the additional above stated assumption that always $\sigma_5 = 0$.

Theorem 4.2.6. *In the case of $\sigma_3 = 0$ (and simultaneously $\sigma_5 = 0$), the intermediate iteration is the last iteration of the syndrome inversion EEA. Otherwise, if $\sigma_3 \neq 0$, at least one iteration is performed after the intermediate iteration.*

Proof. The result simply follows from adopting the proof of Theorem 4.2.3. □

Theorem 4.2.7. *Concerning the polynomial multiplication $b_{i-1}(Y)q_i(Y)$ that is carried out in each iteration, for all iterations from the first up to and including the intermediate iteration, the maximal possible degree of the factors is 6.*

Proof. The result simply follows from adopting the proof of Theorem 4.2.4. □

Theorem 4.2.8. *The polynomial multiplication in the iteration after the intermediate iteration has one factor $q_{j+1}(Y)$ of degree $t - 8$, and the other factor $b_j(Y)$ of degree 6.*

Proof. For the degree of $b_j(Y)$, the proof is analogous to that of Theorem 4.2.5. For the degree of $q_{j+1}(Y)$, we give the differences of the values to those used in Theorem 4.2.5: We first see that $\deg(r_{j-1}(Y)) = t - 6$ and that the only possible value of $\deg(r_j(Y))$ is 2 (since $\sigma_5 = 0$). Thus, we find that $\deg(q_{j+1}(Y)) = t - 8$. □

Since $t - 8$ is far greater than 6 for reasonable code parameters, we see that also for the $w = 6$ error vectors $\sigma_3 = 0$ reliably leads to low timings, since then the most complex polynomial multiplication is skipped. Thus, from detecting $\deg(\Omega(Y)) = 0$, the attacker can learn the equations $\sigma_3 = \sum_{j=3}^{6} \sum_{k=2}^{j-1} \sum_{l=1}^{k-1} \epsilon_j \epsilon_k \epsilon_l = 0$.

4.2.1.4. Enlargement of the Timing Differences by the Key Equation Solving EEA

From the analysis of Sections 4.2.1.2 and 4.2.1.3, we now infer how the case $\deg(\Omega(Y)) = 0$ influences the execution of the other EEA invocation during the code-based decryption, the key equation solving EEA in Algorithm 1, Step 4. We will find that the key equation solving EEA is subject to effects that even enlarge the timing differences resulting from the vulnerabilities found in the two previous sections.

From (4.5) and (4.7), we find that the coefficients of $\Omega(Y)$ are a subset of the coefficients of the polynomial of $\sigma(Y)$. Specifically, they are the coefficients of $\sigma(Y)$ to odd powers of Y. According to Algorithm 1, Step 5, it is $b(Y)$, determined by the key equation solving EEA, which provides the contribution of odd powers of Y to $\sigma(Y)$. Thus,

if $\deg(\Omega(Y)) = 0$, we must have $\deg(b(Y)) = 0$ as well. Considering the determination of $b(Y)$ via the key equation solving EEA, Algorithm 9, we find that we must have $N = 0$, with N being the number of iterations in this EEA execution, since $b_0 = 1$ is the only way to satisfy this restriction on the degree of $b(Y)$. On the other hand, since for $w = 4$ we have $\deg(\Omega(Y)) = 2$ as the only alternative, we have $N = 1$. For $w = 6$ and $\deg(\Omega(Y)) > 0$, we have $N \geq 1$.

The experimental results from Section 4.2.1.7 confirm that, taken together, the timing differences emerging in both EEA applications, i.e. the syndrome inversion and the key equation solving, actually allow for reliable distinction of $\deg(\Omega(Y))$ being zero or non-zero, and thus the attacker is able to learn equations of the form $\sigma_3 = \sum_{i=1}^{4} \epsilon_i = 0$ resp. $\sigma_3 = \sum_{j=3}^{6} \sum_{k=2}^{j-1} \sum_{l=1}^{k-1} \epsilon_j \epsilon_k \epsilon_l = 0$. Remember that through the choice of the error vector during encryption, he chooses the indexes j_i with $i = 1, \ldots, w$ of the support elements $\alpha_{j_i} = \epsilon_i$ according to the definition of the ϵ_i notation for the support elements.

4.2.1.5. The Zero Element of the Support from $w = 1$ Error Vectors

We now show a timing vulnerability that allows for the determination of the position of the zero element in the secret support by inputting error vectors of Hamming weight one into the decryption operation.

For $w = 1$, the whole control flow in Patterson's Algorithm is very simple and unambiguous on a high level:

$$S(Y) \equiv \frac{1}{Y \oplus \epsilon_1} \mod g(Y),$$

$$S^{-1}(Y) = Y \oplus \epsilon_1,$$

$$\tau(Y) = \sqrt{\epsilon_1}$$

$$a(Y) = \tau(Y)$$

$$b(Y) = 1$$

$$\sigma(Y) = Y \oplus \epsilon_1$$

The polynomial inversion is, according to Theorem 4.2.2, performed in exactly one iteration. But there is an ambiguous control flow within the polynomial division given in Algorithm 11 that is executed within this EEA iteration. For $w = 1$, the only division that is carried out is $g(Y)/S(Y)$. We find $q_1(Y) = Y$ because there is no alternative to $\deg(S(Y)) = t-1$. In Algorithm 11, $s_{i,j}$ denotes the coefficient to Y^j in $s_i(Y)$. If $\epsilon_1 = 0$, then the division has to stop at this point. Otherwise, a second iteration is performed giving $q_2(Y) = Y \oplus \epsilon_1$ with $\epsilon_1 \neq 0$. Thus, if the timing difference resulting from the different number of iterations in the division is detectable, the index of z of the secret support element $\alpha_z = 0$ can be found.

Algorithm 11 Polynomial Division poly_div$(n(Y), d(Y))$

Input: the polynomials $n(Y), d(Y)$ with $\deg(n(Y)) \geq \deg(d(Y))$

Output: two polynomials $s(Y)$, $q(Y)$ with $q(Y)d(Y) + s(Y) = n(Y)$ and $\deg(s(Y)) < \deg(d(Y))$

1: $s_{-1}(Y) \leftarrow n(Y)$
2: $s_0(Y) \leftarrow d(Y)$
3: $q_0(Y) \leftarrow 0$
4: $i \leftarrow 0$
5: **while** $\deg(s_i(Y)) \geq \deg(d(Y))$ **do**
6: $\quad i \leftarrow i + 1$
7: $\quad a_i \leftarrow s_{i-2,\deg(s_{i-2}(Y))} / s_{i-1,\deg(s_{i-1}(Y))}$
8: $\quad f_i \leftarrow \deg(s_{i-2}(Y)) - \deg(s_{i-1}(Y))$
9: $\quad q_i(Y) = q_{i-1} + a_i Y^{f_i}$
10: $\quad s_i \leftarrow s_{i-2}(Y) - a_i s_{r-1}(Y) Y^{f_i}$
11: **end while**
12: **return** $(q_i(Y), s_i(Y))$

4.2.1.6. Combining the "$w = 1$", "$w = 4$", and "$w = 6$" Vulnerabilities to a practical Attack

In this section, we explain the construction of a practical attack based on the vulnerabilities shown in Sections 4.2.1.2, 4.2.1.3 and 4.2.1.5. The attack is partitioned into three steps: in Step 1, a linear equation system is built from the information gained through the "$w = 1$" and "$w = 4$" vulnerabilities. In the second step, cubic equations are collected in a specific way. The third and last step is the solving of the resulting equation system. For each step, we first explain the theoretical background and then give the attack description for the respective step. For the third step, we also provide an example for better understanding.

Step 1 – Collecting linear Equations

Theory We recapitulate again the information the attacker gains from the "$w = 4$" ciphertexts, which was derived in Section 4.2.1.2. According to the results of that section, he can determine from the timing of the decryption operation whether $\sigma_3 = 0$ or $\sigma_3 \neq 0$. Specifically, the former case is indicated by a lower timing. The equation he learns in this case is $\sigma_3 = \epsilon_1 \oplus \epsilon_2 \oplus \epsilon_3 \oplus \epsilon_4$. In order to see how this actually contains information about the secret support Γ, we convert the notation:

$$\sigma_3 = \alpha_{i_1} \oplus \alpha_{i_2} \oplus \alpha_{i_3} \oplus \alpha_{i_4},$$

where the i_j with $j \in \{1, \ldots, 4\}$ are the indexes in the error vector $\vec{e}$ that were set to one by the attacker and the α_i are the support elements.

Note that the highest possible rank for a homogeneous linear equation system describing a permutation of $\mathbb{F}_{2^m}$ is $n - m$, since there must be m linearly independent basis

elements.

Attack Description By repeatedly measuring the decryption time for random "$w = 4$" ciphertexts on the decryption device, a rank $n - m - 1$ linear equation system is built. Our experimental results showed that this is the maximal rank that can be achieved in this manner.

Afterwards, the index of the zero element, α_z is determined through the "$w = 1$" vulnerability. In the majority of the cases, this information increases the rank of the equation system to $n - m$. In the rare cases, where the rank remains at $n - m - 1$, the attack's on-line and off-line complexity is increased by a factor of n.

In the following, we assume that we have an equation system of rank $n - m$. Accordingly, by bringing the linear equation system into reduced row echelon form, we find that the elements associated with the m rightmost columns must be a basis $\{\beta_i\}$:

α_0	α_1	$\ldots$	α_i	$\ldots$	α_{n-m-3}	α_{n-m-2}	β_0	$\ldots$	β_{m-1}
1	0	$\ldots$	0	$\ldots$	0	0	X	$\ldots$	X
$\vdots$									
0	0	$\ldots$	1	$\ldots$	0	0	X	$\ldots$	X
$\vdots$									
0	0	$\ldots$	0	$\ldots$	0	1	X	$\ldots$	X

Step 2 – Collecting cubic Equations

Theory At this point, for each element α_i, we know the corresponding B_i with $\alpha_i = \sum_{j \in B_i} \beta_j$, i.e. its representation in the chosen basis. If the values of all basis elements β_i were known, then the values of all α_i would be set as well, and the support was recovered. Accordingly, the next step in the attack is to collect cubic equations according to (4.8) in a way that allows for efficient guessing or solving for the values of the β_i. How this is achieved is explained in the following. We start by showing how the cubic equations are equations about the basis elements and what conditions the ϵ_i, i.e. the error vectors the attacker uses for the creation of the ciphertexts, must satisfy.

First of all, we understand that each cubic equation that the attacker finds through the vulnerability from Section 4.2.1.3 is generally an equation about the basis elements β_i: this is simply achieved by replacing each α_i by its representation as the sum of basis elements: $\alpha_i = \sum_{j \in B_i} \beta_j$ with B_j known from the linear equation system.

Now we see that is already possible to ensure

$$0 = \sigma_5 = \sum_{i=1}^{6} \epsilon_i = \sum_{i=1}^{6} \alpha_{j_i} = \sum_{i=1}^{6} \sum_{k \in B_{j_i}} \beta_k$$

when constructing the ciphertexts: Without knowing the concrete values of the basis elements β_i, we simply have to take care that all the basis elements in the sum in the rightmost term are cancelled out. For this, the knowledge of the B_i is sufficient.

label	equations
C_1	$\beta_{s_1}(\beta_{g_1}, \beta_{g_2}, \beta_{g_3})$
C_2	$\beta_{s_2}(\beta_{s_1}, \beta_{g_1}, \beta_{g_2}, \beta_{g_3})$
C_3	$\beta_{s_3}(\beta_{s_2}, \beta_{s_1}, \beta_{g_1}, \beta_{g_2}, \beta_{g_3})$
$\vdots$	$\vdots$
C_{m-3}	$\beta_{s_{m-3}}(\beta_{s_{m-2}}, \beta_{s_{m-1}}, \dots, \beta_{g_3})$

Table 4.5.: Form of the equations about the basis elements that are collected with the help of the $w = 6$ vulnerability.

As a result, ciphertexts that satisfy $\sum_{i=1}^{6} \epsilon_i = 0$ have a much larger probability for $\deg(\Omega(Y)) = 0$ than ciphertexts built with random $w = 6$ error vectors: for the latter, this probability is in the domain of $1/n^2$, since two coefficients σ_3 and σ_5 must be simultaneously zero. But since for our candidate ciphertexts $\sigma_5 = 0$ is already ensured, their probability for $\deg(\Omega(Y)) = 0$ is in the domain of $1/n$. This reduces the on-line effort of the attack enormously. This probability estimation is based on the assumption that all coefficients of the error locator polynomial become zero with probability $1/n$.

Next, we show how it is possible to collect cubic equations of the form (4.8), which can be solved for one basis element efficiently. We label this basis element β_{s_i}. The simplest type of equation that we can achieve are quadratic equations: because of the restriction $0 = \sigma_5 = \sum_{i=1}^{6} \epsilon_i = 0$, the basis element β_{s_i} must be contained in at least two of the ϵ_i. If exactly two of the ϵ_i contain β_{s_i}, we get a quadratic equation for β_{s_i}. Basic algebra shows that if we label those two ϵ_j that contain β_{s_i} as ϵ_1 and ϵ_2, then from (4.8) we find the following equation:

$$a\beta_{s_i}^2 + b\beta_{s_i} + c = 0, \tag{4.10}$$

with $a = \sum_{j=3}^{6} \epsilon_i$, $b = (\epsilon_1 + \epsilon_2)a$, $c = (\epsilon_1 - \beta_{s_i})(\epsilon_2 - \beta_{s_i})a + (\epsilon_1 + \epsilon_2)\sum_{j=4}^{6}\sum_{k=3}^{j-1} \epsilon_j\epsilon_k + \sum_{j=5}^{6}\sum_{k=4}^{j-1}\sum_{l=3}^{k-1} \epsilon_j\epsilon_k\epsilon_l$ (for clarity, in these formulas we provide "+" and "−" even though both amount to "$\oplus$"). Such a quadratic equation has two solutions for β_{s_i}. This is the type of equation that is used in the attack.

In order to be able to efficiently solve the system of these equations, we want to minimize the brute force effort. We achieve this by building $m - 3$ sets of equations over the β_i of the form given in Table 4.5. This table has to be read as follows: The first set of equations C_1 contains equations for β_{s_1}, which is one arbitrarily chosen basis element. The type of equation represented by $\beta_{s_1}(\beta_{g_1}, \beta_{g_2}, \beta_{g_3})$ is of the form of (4.10) and all the involved ϵ_i are built only from the basis elements $\{\beta_{s_1}, \beta_{g_1}, \beta_{g_2}, \beta_{g_3}\}$. It is not entirely correct to write β_{s_1} as function of the other basis elements, since (4.10) has two solutions. However, we use this notation for simplicity. The next set C_2 contains equations that determine a new basis element β_{s_2} different from the four elements that appear in the equations in C_1. Here, β_{s_2} is determined by the four basis elements that appear in C_1.

The solving of this equation system is the topic of the next step. However, in order to

motivate the choice of these equations, we point out the basic idea behind it: A guess for the first three elements $\beta_{g_1}, \beta_{g_2}, \beta_{g_3}$, using C_1, leads to the determination of the possible values for β_{s_1}. Using C_2, we find the possible values for β_{s_2} and so on. As we shall see, in this process, some of the solutions to the quadratic equations will have to be classified as invalid. Thus, certain "paths" of solutions die off as different "levels" C_i are processed. As a result, each guess for the values of the elements $\beta_{g_1}, \beta_{g_2}, \beta_{g_3}$ leads to a (potentially empty) set of solutions for all basis elements and thus to a guess for the whole support.

At this point, two remarks are appropriate: The reason why we start with an equation where β_{s_1} is dependent on exactly three other basis elements (and not fewer) is given in Appendix A.1. The size of the C_i, which we refer to as c_i, i.e. the number of equations the specific set contains, will be addressed in the explanation of the next step of the attack. For now, it is sufficient to know that the c_i can be chosen to be different from one another and that they are parameters of the attack.

We now summarize again all the above mentioned conditions that apply to the ϵ_i, and thus the error positions that the attacker choses when preparing the ciphertexts with $w = 6$.

1. $\epsilon_i \in \text{span}(\{\beta_{s_1}, \beta_{g_1}, \beta_{g_2}, \beta_{g_3}\})$,
2. $\sum_{i=1}^{6} \epsilon_i = 0$,
3. exactly two of the ϵ_i contain β_{s_1}.

Attack Description Thus, the second step of the attack is carried out as follows: The attacker creates ciphertexts that meet these three conditions, and inputs them into the decryption device. From the timing of this operation, he identifies those ciphertexts for which $\sigma_3 = 0$. Whenever this happens, he registers an according equation of form of (4.10). In this process, he first collects a specified number of equations that involve four basis elements; this will yield the set of equations C_1. Afterwards, he adds a fourth element to the list of available basis elements and collects a specified number of equations C_2. He continues this process, until all $m - 3$ sets C_i sets are completed.

Step 3 – Solving

Theory In this step, the solving or guessing is performed. This means that the equations collected in the previous step about the basis elements β_i are solved. Note that the information in the linear equation system built in Step 1 has already been used to make the conversion from the ϵ_i to the β_i in (4.10) in Step 2, and thus is not further useful.

Note that during the guessing of the values of the basis elements, since we are looking for linearly independent $\mathbb{F}_{2^m}$ elements, all new guesses that are linearly dependent on basis elements that have already been guessed, are discarded.

Attack Description The solving is performed by enumerating all the possible combinations of the values of β_{g_1}, β_{g_2} and β_{g_3}. Here, and for the subsequent guesses, we enforce that

$$\beta_{g_i} \notin \mathrm{span}(\{\beta_{g_1}, \ldots, \beta_{g_{i-1}}\}), \tag{4.11}$$

by discarding inappropriate guesses or solutions. Here, we imply the convention $\beta_{s_i} = \beta_{g_{i+3}}$.

For each such combination of values for $\beta_{g_1}, \beta_{g_2}, \beta_{g_3}$, the roots of each equation in C_1 are potential candidates for the value of β_{s_1}. However, additionally to the restriction from (4.11), those roots that are found only for a subset of C_1 are discarded.

The remaining roots are iterated over to find the possible solutions for β_{s_2} by solving the equations in C_2, which in turn are used to compute the possible values of β_{s_3}, etc. Whenever in such a chain of guesses a solution for all β_{s_i} is found, a guess for the whole support $\Gamma = (\alpha_i | \alpha_i = \sum_{j \in B_i} \beta_j)$ is implied, which has to be checked by a means of key recovery, as described in [18]. The attack is finished when such a key recovery is successful.

The value of a choice $c_i = |C_i| > 1$ lies in the following: The more equations are used for the determination of the set of possible solutions of one specific β_{s_i}, the more likely it is that none of them is passed on to the subsequent evaluation of C_{i+1}. Thus, the larger the c_i are chosen, the higher is the on-line effort of the attack (more cubic equations have to be collected), but the off-line effort is reduced as the number of possible solutions for each β_{s_i} is potentially decreased. By virtue of this effect, the employment of large values for the c_i allows to reduce the total number of the necessary key recoveries.

Example Figure 4.6 gives a sample run for very small values of the parameters c_1 and c_2. We see that the equations in C_1 are evaluated for the values x, y, z of the basis elements $\beta_{g_1}, \beta_{g_2}, \beta_{g_3}$. All three equations in C_1 find the same two solutions a and b for β_{s_1}. Furthermore, they are both found to be linearly independent from x, y, z. Thus, both of these values are valid candidates and must now be used to evaluate the equations in C_2. For the solution $\beta_{s_1} = a$, the two equations in C_2 yield only one value that is in the intersection of their solutions, this is $\beta_{s_2} = d$. Furthermore, this value is linearly independent of the all the previously determined basis elements a, x, y, z. Thus, it is a valid candidate and must be now evaluated for C_3.

For the solution $\beta_{s_1} = b$, the equations of C_2 again yield two values f and h for β_{s_2}, which are in the intersection of their solutions. However, it turns out that d is linearly dependent of the previously determined basis elements and thus needs not be processed further. However, d is again a valid solution and used for the evaluation of the equations in C_3.

4.2.1.7. Experimental Results

We successfully conducted the attack with the following measurement setup on an Intel Core 2 Duo x86 platform: from the attack program, the decryption function was called with the attack ciphertexts as input, and the decryption time was measured with the CPU's cycle counter.

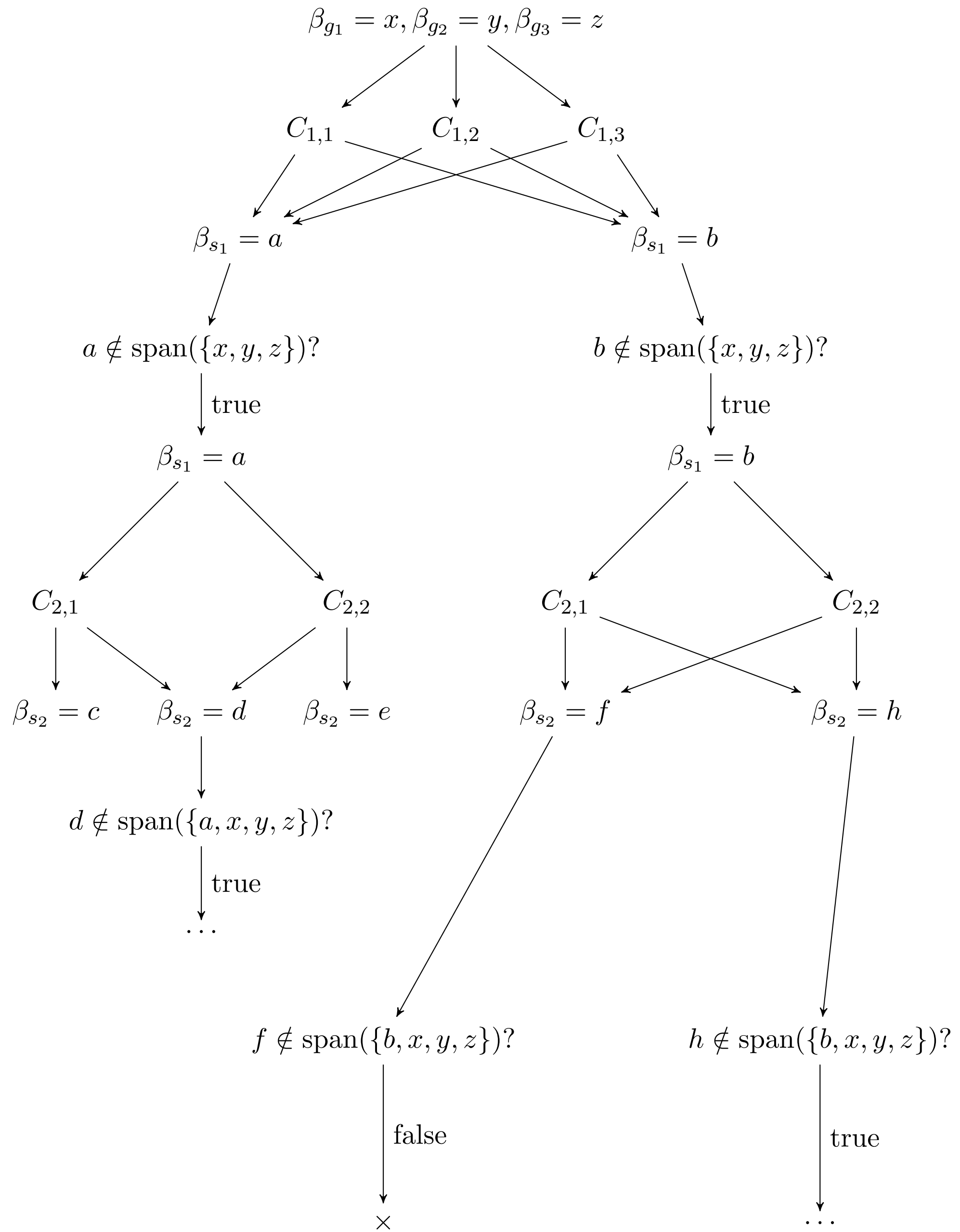


Figure 4.6.: Sample run of the solving of the equations about the basis elements for $c_1 = 3$ and $c_2 = 2$. Here, the equations contained in C_i are enumerated as $C_{i,1}, C_{i,2}, \dots$.

Because the cycle counts measured for a deterministic operation of the duration of a code-based decryption vary considerably on such CPUs, a specific strategy has to be used to identify positives, which refers to $\epsilon_1 = 0$ for $w = 1$ and $\deg(\Omega(Y)) = 0$ for $w = 4$ and $w = 6$, i.e. those cases that yield an equation for the attacker. The timing behaviour of a modern x86 CPUs like the Core2 Duo can be approximated through the following model: for repeated executions of a fixed sequence of instructions, there is a hypothetical constant cycle count (the minimal execution time for the instruction sequence), which is increased by a random delay. This delay varies in each execution and ranges from zero to a certain maximal value. Because in all three different attack types the positives, from the algorithmic point of view, are executed faster than the negatives, the following classification strategy can be used: Prior to the attack, a training phase is carried out where the minimal cycle counts for positives are determined as well as the minimal cycle counts for negatives (using a different secret key than during the attack). Then, the threshold below which an operation is classified as a positive during the attack is set as the mean of these two values. We refer to the distance between the minimal cycle counts for positives and the minimal cycle counts for negatives as the cycles gap. Clearly, the larger this gap the larger is the probability for finding positives. Furthermore, the above approximate model for the cycle counts on the employed CPU is lacking other effects that could be observed in our experiments: during the execution of the attack, the previously determined maximal and minimal cycle counts for the two classes of operations seem to be subject to an “upwards drift”, i.e. they tend to successively increase over time but sometimes also drop again approximately to the initial levels after some time.

Table 4.6 summarizes the results for single attack runs with different code parameters. The rows labeled “cycles gap ... ” indicate the above discussed gaps. We found that gaps of a couple of hundreds cycles that are characteristic for the $w = 1$ vulnerability tend to cause problems in the detection of positives, i.e. in some runs, due to the mentioned drift of the cycle counts, the zero support element could not be determined, while the considerably larger gaps for the $w = 4$ and $w = 6$ vulnerabilities allow for reliable detection of positives.

The rows labeled “number of queries ... ” show the number of decryption operations that had to be executed with ciphertexts created with error vectors of the respective weight in the course of a single run of the attack.

“number of final verifications” is the number of the guesses for the complete support that are output by the attack. We did not implement an actual verification, but simply compared the guess for the support with the correct support Γ. As already mentioned, in [18], the procedure that had to be used in a real life attack is described. It involves only some linear algebra operations on the public key and the invocation of an EEA and would not perceptibly increase the time for solving, given the small numbers of such final verifications occurring in the attacks.

The time for the solving step is given in the last row. From the theory, one expects an increase of the solving time by a factor of about eight for each increase of m by one. The reason is that the number of initial guesses, i.e. the number of combinations of values that can be chosen for β_{g_1}, β_{g_2} and β_{g_3} is roughly n^3, and all $\mathbb{F}_{2^m}$ operations, including the solving of the quadratic equations [50] (4.10), are done with the help of

	$m = 9$, $t = 33$	$m = 10$, $t = 40$
cycles gap $w = 1$	≈ 400	≈ 600
cycles gap $w = 4$	$\approx 13,000$	$\approx 19,000$
cycles gap $w = 6$	$\approx 17,000$	$\approx 23,000$
number of queries for $w = 1$ (Step 1)	3,575,494	11,782,695
number of queries for $w = 4$ (Step 1)	1,517,253	2,869,424
number of queries for $w = 6$ (Step 2)	374,927	1,837,125
number of final verifications (Step 3)	$\approx 8,000$	$\approx 2,000$
running time for solving on 1 GHz x86 CPU (Step 3)	3h	28h

Table 4.6.: Experimental results for single runs of the attack. Refer to the text for explanations.

lookup tables, and thus are executed in constant time.

The number of equations collected per β_{s_i} were chosen as $c_1 = 1$, $c_2 = 2$, $c_3 = 4$, i.e. chosen as the double of the previous count, up to a maximal value of 16, i.e. $c_i = 16$ for $i \geq 5$. This choice was found to provide good solving times as well as a small number of final verifications. However, we did not systematically search an optimal set of parameters c_i.

As previously mentioned, in the rare cases where the knowledge about the zero-element of the support does not increase the rank of the equation system, Steps 2 and 3 would have to be repeated about up to n times; for these cases stronger hardware would be needed to keep the solving time in reasonable margins.

These results show first of all the practical relevance of the vulnerability concerning local attacks, i.e. where the attacker has direct access to the decryption routines without any delay. However, the sizes of the cycle gaps for the "w=4" and "w=6" are so large that even network attacks seem possible: in [57], timing gaps between 10.000 and 20.000 cycles are exploited for a distinguishing attack over a network connection. Concerning the amount of queries that is in the millions for our attack, it must be noted that in other attacks considered in the literature far greater numbers are used: in [58] (not a timing attack but purely a mathematical one), $2^{30} \approx 10^9$ queries are needed for an attack against RC4 in TLS.

4.2.1.8. Effect of Countermeasures against other Attacks

If the countermeasure proposed in Section 4.1.2.2 is incorporated in an implementation, then the attack becomes even easier. This is because the countermeasure from Section 4.1.2.2 tremendously bloats the timing difference that is exploited in the attack. Specifically, it will not alter the case of $N = 0$ on one hand, but will enforce the maximum number of iterations (that would occur for $w = t$), instead of $N = 1$, on the other hand. This is a direct consequence of the fact that this countermeasure resides in a way inside the EEA loop, and will be skipped if that loop is skipped.

4.2.1.9. Possible Extensions of the Attack

In this section, we give two possible extensions or improvements of the attack, the first being useful if noisy timings are used and the second showing how a power analysis attack could be used to retrieve the same information.

"n-3" Scans for the Linear Equations In attacks in other scenarios, for instance when the attack is executed as a remote timing attack (over a network), the attacker may be confronted with the problem of receiving only noisy measurements. This means for our attack that the decision whether $N = 0$ or $N = 1$, based on a timing measurement might be very difficult or impossible for a single timing. But for the retrieval of the linear equations, it is possible for the attacker to conduct measurements on certain sets containing $n - 3$ error vectors, where he knows that exactly one of the vectors in the set causes $N = 0$. This is due to the following observation.

For every error vector of Hamming weight $w = 4$, we have $\sigma_3(a, b, c, d) = \gamma_a + \gamma_b + \gamma_c + \gamma_d$ where a, b, c, d are all different values from $\{0, \ldots, n-1\}$. Now, take w.l.o.g., the sum of three of these elements, $\gamma_a + \gamma_b + \gamma_c$. We now postulate that there always exists some γ_d different from the former three elements, such that $\sigma_3(a, b, c, d) = 0$. We prove this postulate by showing that assuming the opposite statement leads to a contradiction. Specifically, the non-existence of such a γ_d implies that the additive inverse of $\gamma_a + \gamma_b + \gamma_c$ is one out of the set $\{\gamma_a, \gamma_b, \gamma_c\}$. Since in a finite field of characteristic 2, each element is its own additive inverse, this in turn would imply $\gamma_a + \gamma_b + \gamma_c = \gamma_a$, where the choice of γ_a is w.l.o.g. But then we would have $\gamma_b = \gamma_c$, which is the contradiction we sought. This also implies that the probability for $N = 0$ for a ciphertext with a random error pattern with $w = 4$ is $1/(n-3)$.

Thus, the attacker can proceed as follows: he fixes an error pattern of Hamming weight $w = 3$ and produces the $n - 3$ vectors, which can be reached by adding one more error position and uses them to create $n - 3$ ciphertexts and measures the corresponding decryption times. He can now use a maximum-likelihood strategy to determine the one timing which corresponds to $N = 0$ iterations in the EEA. He can selectively repeat measurements to increase the certainty of his decision if the amount of noise suggests this.

Starting Point for a Power Analysis Attack. We now show how a potential power analysis vulnerability emerges for the $w = 4$ ciphertexts, which allows to gain the same information as from the timing vulnerability. From the invocation of Algorithm 9 from Step 4 of Algorithm 1 and the considerations given in Section 4.2.1.2, it is clear that in the case of $N = 0$ we have $\alpha(Y) = \tau(Y)$, and thus, $\deg(\tau(Y)) = 2$ in Step 3 of Algorithm 1. On the other hand, if $N = 1$, then we know that $\deg(\tau(Y)) = t - 1$, because $\deg(\beta(Y)) = \deg(q_1(Y)) = 1$ is the only possibility according to Algorithm 9.

This means, that in Step 3 of Algorithm 1, for $N = 0$, a large number of coefficients turns out to be zero when the square root is computed. Since the Hamming weight of registers or the number of the changed bits in a register are usual targets of a power

analysis attack [59], one could expect that an unsecured implementation makes it possible to distinguish between the cases $N = 0$ (i.e. $\sigma_3 = 0$) and $N = 1$ (i.e. $\sigma_3 \neq 0$).

4.2.1.10. The Problem of Countermeasures

An effective countermeasure to protect against this attack has not been created. We address this open problem in Section 6.2.

4.2.2. Timing Attacks against Root-Finding Algorithms

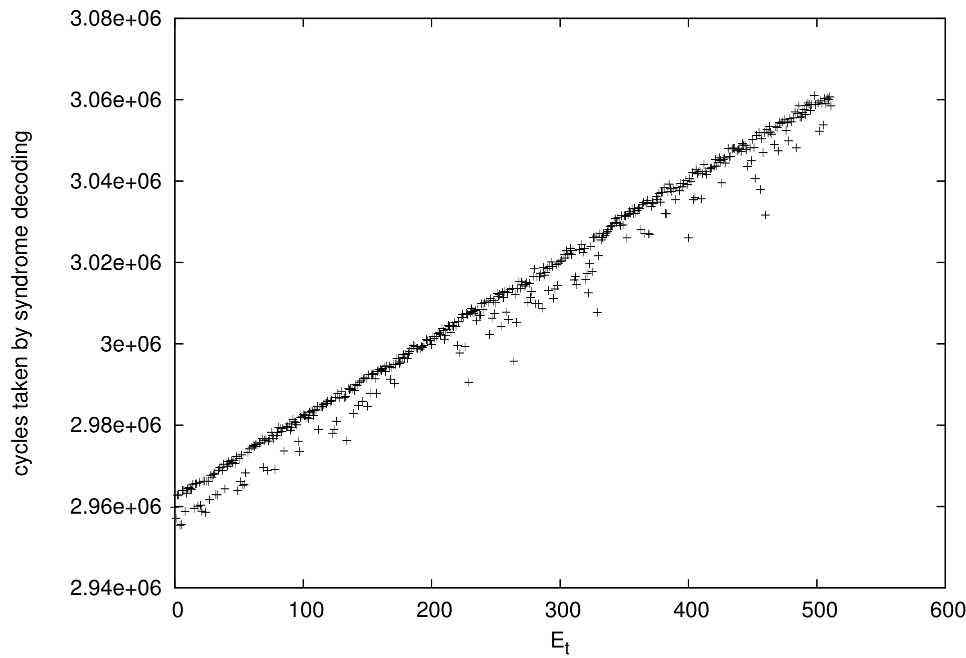
We now show that in the root-finding algorithm vulnerabilities can arise, which allow attacks against the secret support of code-based PKCs. This is the case when the running time of the root-finding algorithm depends on the values of the roots found. To understand that this is a vulnerability, one has to consider that an attacker can create ciphertexts with $\vec{e}$ known to him. Then, according to (2.1), any information about the roots is information about the support Γ.

4.2.2.1. Vulnerability of eval-div-rf

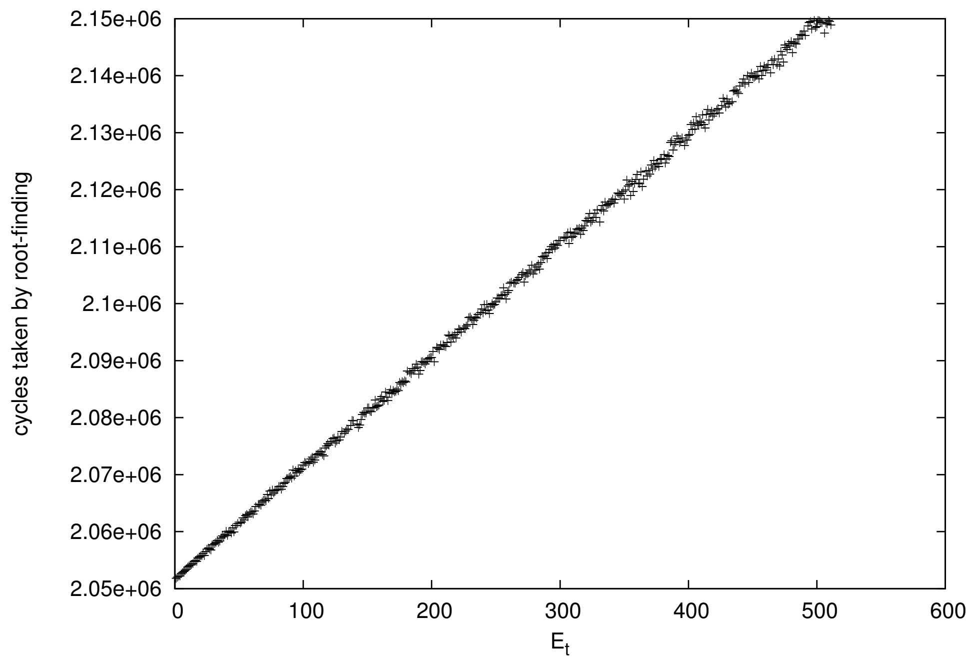
One possible vulnerability arises if in *eval-div-rf*, presented in Section 3.3.2.1, the evaluation of $\sigma(Y)$ is done with Y being substituted in lexicographical order. In this case, the found roots are later mapped to the corresponding E_i, i.e. the indexes of the bits having value one in the error vector, by using a table for Γ^{-1}. The timing effects of this implementation choice are shown in Fig. 4.7(a) and 4.7(b). These figures depict running times on an Atmel ATMega1284P AVR platform of the syndrome decoding with *eval-div-rf* for $n - (t - 1)$ error vectors created in the following way: a random error pattern of weight $t - 1$ was fixed, and the position of the last error, E_t, was varied over the remaining free positions, resulting in error vectors with Hamming weight t. On the x-axis, E_t is shown. Henceforth, we will refer to this type of plot as “support scan”. The result is a relatively clear linear ascent, which is not surprising when considering the *eval-div-rf* algorithm: Starting evaluation at $Y = 0$, the earlier a root is found, the more beneficial in terms of running time is the reduction of the degree of $\sigma(Y)$ by one through the subsequent division. Thus, the task for an attacker amounts to bringing the measured timings into an ascending ordering, giving him Γ. Obviously, there is some distortion of this ordering in Fig. 4.7(a), which stems from other operations of variable duration in the syndrome decoding. We leave it open whether in this manner the support Γ becomes known to the attacker in its entirety, however, a large amount of information about Γ becomes available.

This vulnerability can be avoided by performing the evaluation of $\sigma(Y)$ with Y being substituted in the order $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$. Note, however, that the vulnerable version is slightly faster, since there only t table lookups in Γ^{-1} for the found roots are done, whereas in the secure version n such lookups in Γ are necessary. Thus, the described problem is realistic.

We also wish to point out that an attack exploiting this vulnerability, in contrast to other previously published timing attacks [2, 3], cannot be detected: The ciphertext



(a) Timing for the whole syndrome decoding with *eval-div-rf*.



(b) Timing for the root-finding with *eval-div-rf*.

Figure 4.7.: Running times of *eval-div-rf* for $n - (t - 1)$ ciphertexts, where $t - 1$ error positions are fixed and the $t - th$ position varies, with code parameters $n = 512$ and $t = 33$.

carries t errors and will pass the CCA2 integrity test. This is important, because the other attacks, which cannot be carried out in a clandestine manner in this sense, can be thwarted by countermeasures that detect the irregularity of the ciphertext, and for instance add an enormous delay or enforce constant running time, if possible, on the respective platform. In the presence of the threat of power analysis attacks, however, such countermeasures would in most cases not suffice as adding delays after the actual computation will most likely be detectable in the power trace.

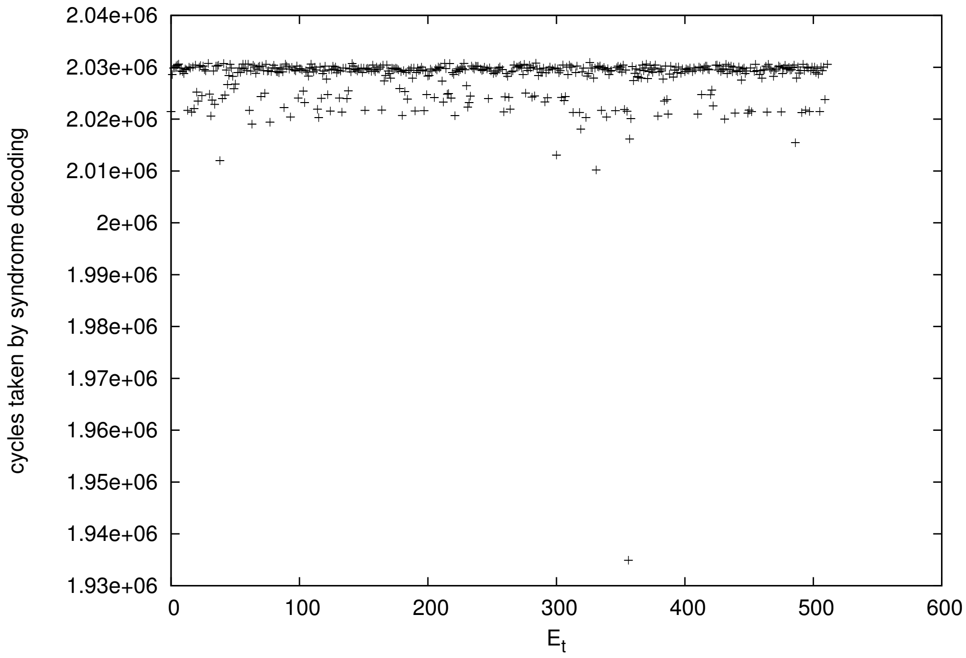
This vulnerability extends naturally to the root-finding variant *dcmp-div-rf*. However, since in this algorithm the error locator polynomial of reduced degree is only used after its degree has been reduced by five, the information an attacker can collect about the secret support is limited there.

4.2.2.2. Vulnerability of dcmp-rf

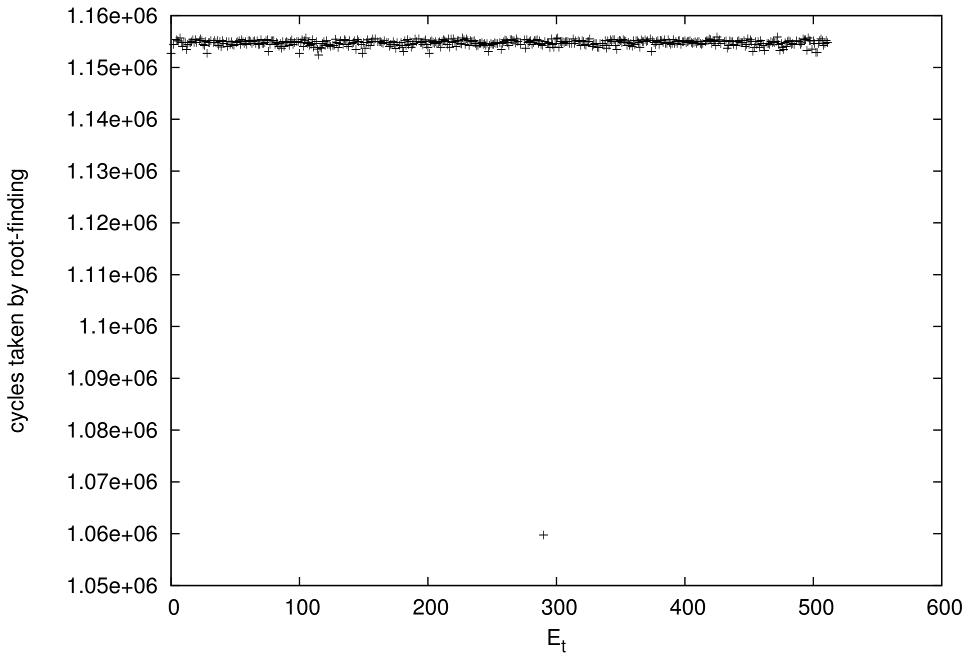
In *dcmp-rf*, there exists one minor potential vulnerability: this is the multiplication by σ_3 in (3.3). If $\sigma_3 = 0$, then this multiplication is faster (no table lookups and reduction modulo the multiplicative order is performed, refer to Section 3.3.1) than in the general case. Another implementation choice is to precompute the logarithm of σ_3 for a performance gain, then in the case of $\sigma_3 = 0$, this multiplication has to be skipped. However, the timing vulnerability is the same in both cases. The manifestation of this vulnerability on the AVR platform is shown in Figure 4.8. Note that the countermeasure from Section 4.1.2.2 prevents this vulnerability from being used for the low weight attacks from Section 4.2.1, where it would be useful to further reduce the timings for $\sigma_3 = 0$. This countermeasure ensures a fake error locator polynomial $\sigma(Y)$ of degree t; it is triggered always for the ciphertexts in the low weight attacks. Thus, the use of this vulnerability in these attacks is only a realistic assumption for an implementation without the countermeasures from Section 4.1.2.2. Accordingly, we turn our attention to the case of $w = t$, where this countermeasure is not triggered. In that case, the information gained is, according to (2.1):

$$0 = \sigma_3 = \alpha_{E_1}\alpha_{E_2}\dots\alpha_{E_{w-3}} \oplus \alpha_{E_1}\alpha_{E_2}\dots\alpha_{E_{w-4}}\alpha_{E_w} \oplus \dots,$$

i.e. the sum of products of all possible combinations of $w-3$ different support elements associated with the respective error positions, where w is the error vector's Hamming weight, usually $w = t$. It is certainly not trivial to exploit this information; however, in combination with other vulnerabilities, it might be useful to provide a means of verifying guesses for Γ. The countermeasure to protect against this vulnerability is trivial and comes at a low computational cost: it is realized by assigning the precomputed value of the logarithm of σ_3 a dummy value during the initialization phase of *dcmp-div-rf* if $\sigma_3 = 0$, and carrying out the multiplication $Y^3\sigma_3$ with both operands in logarithmic representation regardless of the value of σ_3. Afterwards, a logical AND operation is performed on the result with a mask having all bits set in case of $\sigma_3 \neq 0$ and having value 0 otherwise.



(a) Running times of the syndrome decoding without countermeasures to hide $\sigma_3 = 0$ in *dcmp-rf*. The outlier with smallest running time shows such a case.



(b) Running times of the root-finding with *dcmp-rf* without countermeasures to hide $\sigma_3 = 0$. The outlier with smallest running time shows such a case.

Figure 4.8.: Running times of *dcmp-rf* for $n - (t - 1)$ ciphertexts, where $t - 1$ error positions are fixed and the t-th position E_t varies.

4.3. Fault Attacks

In this section, we point out software fault attack vulnerabilities in the decryption operation of an existing open source implementation of the McEliece PKC [7].

A hardware fault attack [60] is given when an attacker can gain information about internal values of computations through faults that he provokes by influencing the hardware of the device. In a software fault attack, which is the type of attack we are considering here, the attacker gains knowledge about internal values evaluating error replies of the device as reaction to invalid input data.

The software fault attack vulnerabilities presented in this section are directly related to the message-aimed timing vulnerabilities from Section 4.1. They differ from these only in one aspect: the distinction of two different control flows is not achieved through a timing analysis, but by evaluating error messages. This means that also the corresponding fault attacks are naturally built from their timing attack counterparts.

We analyzed the HyMES open source implementation [20] with respect to such fault attack vulnerabilities. We found two such vulnerabilities in the McEliece decryption, which we present in the following sections.

4.3.1. Fault Attack Vulnerability revealing the Degree of the Error Locator Polynomial

The first fault vulnerability enables fault attacks based on the timing attacks presented in Section 4.1.2.1. All code relevant to the syndrome decoding in HyMES is found in the file `decrypt.c`; all line numbers given in the following refer to this file. In line 270, when $\deg(\sigma(Y)) \neq t$, decryption is aborted with an error. This is only a problem if the countermeasures proposed in Section 4.1.2.2 are not implemented (as it is the case in the HyMES implementation). In this case, it allows message-aimed fault-attacks of the type explained there: it is highly likely that $w > t$ leads to $\deg(\sigma(Y)) = t$, and always that $w < t$ leads to $\deg(\sigma(Y)) = w$. In the timing attack, it is deduced from the timing whether $\deg(\sigma(Y)) < t$; in the corresponding fault attack an explicit error message reveals this condition. If the countermeasure from Section 4.1.2.2 is implemented, then this fault attack vulnerability is removed.

4.3.2. Fault Attack Vulnerability revealing Information about the Number of Roots of the Error Locator Polynomial

In this section, we show that in the HyMES implementation there exists a vulnerability that enables the timing attacks from Section 4.1.4 as fault attacks.

In line 276 of the file `decrypt.c`, if the root-finding did not return t roots, decryption is also aborted with an error. This allows message-aimed fault attacks with the two-bit-flip attack described earlier in Section 4.1.4: there, it is deduced from the timing whether t roots or only a fraction of this are found during the decryption of a ciphertext. With the fault attack vulnerability, in the case of t roots, the decryption is processed without errors, and in case of less roots an error is returned. Accordingly, this enables fault

attacks derived from the timing attack described in Section 4.1.4. Note that, according to the results of that section, the number of roots remains an indicator for the value of w even in the presence of the countermeasure given in Section 4.1.2.1. Accordingly, such a check for the number of roots must not be present in a secure implementation.

4.4. Transferability of the Vulnerabilities and Countermeasures to the Niederreiter PKC

In the following, we show that the attacks presented in Section 4.1, 4.2 and 4.3, which were presented as attacks against the McEliece PKC are equally applicable to the Niederreiter PKC. The attacks from Section 4.1 and 4.3 rely on the attacker's ability to change a single error position in the McEliece ciphertext. The equivalent procedure for the Niederreiter PKC is the addition of a row of the public parity check matrix: given a ciphertext $\vec{z} = H\vec{e}^t$, the ciphertext $\vec{z}' = \vec{z} \oplus H\vec{v}^t = H(\vec{e} \oplus \vec{v})^t$ with $\vec{v}$ having exactly a single non-zero entry is a syndrome corresponding to an error vector carrying either $t-1$ or $t+1$ errors; this is just the type of manipulation of ciphertexts that is needed in the message-aimed attacks.

The transferability of the key-aimed attacks from Section 4.2 is also obvious, as the attacker can create ciphertexts with error patterns of his choice just like in the case of the McEliece PKC.

4.5. Relation of the Side Channel Vulnerabilities of Code-based PKCs to those of other Cryptosystems

In the following, we analyze the relation of certain aspects of side channel issues in code-based PKCs to those in other PKCs [5]. First, in Section 4.5.1, we put the message-aimed side channel attacks introduced in Section 4.1 into a broader context. This allows us to derive a methodology for the analysis of public key cryptosystems with homomorphic properties with respect to message-aimed side channel attacks.

Then, in Section 4.5.2, we explain the situation of code-based PKCs with respect to generic side channel countermeasures. We find that such a countermeasure can only be realized with an increase in the security parameters.

4.5.1. Message-aimed Side Channel Attacks against Cryptosystems with homomorphic Properties

We now put the message-aimed attacks given in Section 4.1 into a more general context; specifically, we will show that these attacks share a number of features with the message-aimed attacks against the RSA cryptosystem. As the common features of the attacks against these two PKCs we find that both are carried out as timing attacks based on (adaptively) chosen ciphertext attacks, where the timings reveal information about certain properties of the message.

The usefulness of the knowledge of the properties of the manipulated plaintexts is given by the so-called homomorphic properties of the cryptosystems in question. A homomorphic property is present when an equation of the type $\mathcal{E}(a) \bullet \mathcal{E}(b) = \mathcal{E}(a \odot b)$ is given, where $\mathcal{E}(x)$ denotes the encryption operation, and "$\bullet$" and "$\odot$" indicate arbitrary group operations. A homomorphic property is a special form of malleability [61]. This latter property is given when any relations between the two ciphertexts can be produced that cannot necessarily be expressed in a closed equation.

Homomorphic properties are sometimes desired, for instance in the context of electronic voting schemes [62]. Here, however, we are interested in exploitable properties rather than useful ones. This allows us to work also with homomorphic properties that can be characterized as faint or restricted and would never qualify for useful applications. As we will see in the following, this is true for the McEliece cryptosystem.

We start with a review of the message-aimed side channel attacks against the RSA cryptosystem in Section 4.5.1.1, then we show the homomorphic properties of both PKCs in Section 4.5.1.2. In Section 4.5.1.3, we make a comparison between the vulnerabilities underlying the message-aimed attacks against these cryptosystems. In Section 4.5.1.4, we conclusively derive a methodology for the analysis of PKCs with homomorphic properties.

4.5.1.1. Manger's Attack against RSA-OAEP

First, let us briefly review the famous RSA cryptosystem. The RSA private key consists of the private exponent d, the public key is given by the modulus n and the public exponent e. RSA encryption is performed by computing the ciphertext $c = m^e \mod n$, which is decrypted as $m = c^d \mod n$. The knowledge of the prime factors p, q with $pq = n$ is what enables the holder of the secret key to determine the correct private exponent d.

In [63], James Manger proposes a message-aimed attack against RSA-OAEP [64]. RSA-OAEP is a conversion that makes the cryptosystem secure against adaptively chosen ciphertext attacks. During the decryption operation, first, the RSA message representative is computed by the RSA decryption operation, then the OAEP decoding is applied to the encoded message representative. Manger's attack is based on the fact that during the OAEP decoding, two different checks are performed. The first one is directly applied to the encoded RSA message: it is demanded that the message features at least one octet less than the encoded public modulus. If this condition is violated, an error shall be thrown. In the remainder of Section 4.5, we will speak of a supernumerary octet in this case. Then, at a later stage in the OAEP decoding, the actual integrity check is carried out, and if the final comparison of two hash values fails, a second error condition is triggered. This latter error condition will always be triggered when a manipulated ciphertext is decrypted. Manger points out that these two different error conditions could be identified by an attacker if they either produce different error messages or the associated timings can be distinguished. In contrast to older specifications [64], the current version of the PKCS#1 specification [65] specifies the algorithm in such a way that implementers will less likely introduce such vulnerabilities.

Manger's attack makes use of this distinction of error conditions in the following way: The attacker creates manipulated versions of the ciphertext he wishes to decrypt. Specifically, given the ciphertext c_0, he chooses an integer f and computes

$$c'_0 = f^e c_0 \mod n \tag{4.12}$$

where e and n are the public exponent and modulus of the key that was used to encrypt c_0. The attack builds on the ability of the attacker to let the decryption device decrypt the manipulated cipher text c'_0 and learn whether $m'_0 = fm_0 = c'^d_0 \mod n$ has a supernumerary octet in the sense described above. Let B be the smallest value of a message m that features a supernumerary octet, then the information learned by the attacker is whether $fm_0 \mod n \geq B$ is true or false. Repeating this with f chosen based on previous outcomes, he can by and by narrow down the number of possible values of m_0. This is done with a specific strategy described in [63]. The details of this attack are not necessary to understand the remainder of this section; the only important thing is the source of the information gain.

In [66], it is discussed that other sources for timing differences based on the existence of such a supernumerary octet can be found in the integer to octet string conversion that precedes the OAEP decoding operation. This is due to the fact that these conversion routines generally iterate over the octets of the encoded integer. The implementation of this function in the Botan Library [67] is given in Listing 1 as an example.

```
335  void BigInt::binary_encode(byte output[]) const
336   {
337   const u32bit sig_bytes = bytes();
338   for(u32bit j = 0; j != sig_bytes; ++j)
339      output[sig_bytes-j-1] = byte_at(j);
340   }
```

Listing 1: The implementation of the integer encoding in Botan-1.9.7 located in the file `math/bigint/bigint.cpp`.

Furthermore, in the same work it is shown that already the last operations in the modular multi-precision integer arithmetic within the RSA decryption routine might introduce small timing differences based on the number of significant octets.

Concerning such small timing differences, it is not always clear whether or under which circumstances these are exploitable through a timing attack, since other sources of timing differences in the decryption algorithm may introduce considerable noise into the measurements. However, through simple or differential power analysis [22], the running times of the individual operations may be observed in an isolated way, if these operations can be distinguished in the power trace. In the above example, this means that if the duration of the integer to octet string conversion can be concluded from the power trace, an attack is possible. This idea can be applied to the other timing-related vulnerabilities discussed in the following and will be assumed throughout the remainder of Section 4.5 without explicitly mentioning it.

4.5.1.2. Homomorphic Properties of RSA and the McEliece Cryptosystem

In this section, we show that both the message-aimed attacks against the RSA and McEliece cryptosystems are based on their homomorphic properties. While the homomorphic properties of RSA are well known and their exploitation in the attack by Manger is quite obvious, for the McEliece cryptosystem this may be worth pointing out in some more detail.

The homomorphic property of the RSA cryptosystem is given by

$$\mathcal{E}(m_1)\mathcal{E}(m_2) = \mathcal{E}(m_1 m_2 \mod n).$$

From (4.12), we see that the message-aimed attacks against RSA discussed previously make use of this property.

As a matter of fact, the McEliece cryptosystem does not really have plain homomorphic properties. It can easily be modified to have somewhat restricted homomorphic properties: if a t-error correcting code is used, but only error vectors e with $\mathrm{wt}\,(e) = \lfloor t/r \rfloor$ are added to the codewords during the encryption operation, then we have the property

$$\mathcal{E}(\vec{v}_1) \oplus \mathcal{E}(\vec{v}_2) = \mathcal{E}(\vec{v}_1 \oplus \vec{v}_2)$$

$$\text{for up to } r \text{ additions } \mathcal{E}(\vec{v}_1) \oplus \mathcal{E}(\vec{v}_2) \tag{4.13}$$

because $\vec{v}_1 G_p \oplus \vec{e}_1 \oplus \vec{v}_2 G_p \oplus \vec{e}_2 = (\vec{v}_1 \oplus \vec{v}_2) G_p \oplus (\vec{e}_1 \oplus \vec{e}_2)$ with $\mathrm{wt}\,(\vec{e}_1 \oplus \vec{e}_2) \leq t$.

But for our purpose, it is more useful to view the error vector $\vec{e}$ as message. This view is justified because knowing $\vec{e}$ allows anyone to recover $\vec{v}$ with acceptable effort[1]. Then, we obviously have the same homomorphic property as in Equation (4.13). In this view, the introduction of single bit flip errors in the ciphertext, as in the attacks outlined in Section 4.1, can be interpreted as the addition of another ciphertext, i.e. an error vector of Hamming weight one, according to Equation (4.13).

The reason that we can also attack a McEliece cryptosystem where $\mathrm{wt}\,(\vec{e}) = t$ is that bit flips are homomorphic operations with a certain probability, i.e. when the overall number of errors is not increased. From this perspective, it suffices if the attacker is able to distinguish whether his manipulations are homomorphic operations or not.

4.5.1.3. Comparison of Message-aimed Side Channel Attacks against RSA and McEliece

We now show the analogies between the messaged-aimed attacks against the RSA and McEliece cryptosystems. The goal is to gain insights into the principles that enable this type of side channel attack against public key cryptosystems.

With this motivation, we first turn our attention to the timing attack vulnerabilities of the RSA and McEliece cryptosystems we already reviewed and introduced, in the previous sections. We then take into account fault attacks and finally show how so-called reaction attacks are related to the former two types of attack.

[1] Also, the McEliece cryptosystem could in fact be altered to actually encode the message in the error vector only. In this case however, the Niederreiter [12] scheme would be more preferable.

Timing Attacks Table 4.7 shows both the RSA and the McEliece decryption processes, where we distinguish between the public key decryption routine, the encoding of the message representative, and the CCA2 check, and indicate the subset of these operations for each cryptosystem that exposes potential timing vulnerabilities. Both cryptosystems share the property that, already during the public key decryption operation itself, there are potential sources of timing differences based on those properties of the message controlled by the attacker.

As we have seen in Section 4.5.1.1, for RSA, the first potential vulnerability is the final operation in $\mathbb{Z}_n$. Timing related vulnerabilities (with respect to message-aimed attacks) that reach back further into the RSA decryption operation are not known.

In the McEliece PKC, there are two parts of the decryption operation which reveal the number of errors through e.g. the timing. These are, as previously discussed, the key equation solving and the root finding (Steps 4 and 6 of Algorithm 1). Compared to the corresponding vulnerabilities in the RSA cryptosystem, those of the McEliece cryptosystem reach back considerably further into the decryption algorithm.

The next interesting step is the encoding of the message representative. Here, according to [66], the encoding routine's running time depends on the number of octets needed to represent the RSA message. In the case of McEliece, such a vulnerability is rather inconceivable in a reasonable implementation. The reason for this difference is that for the encoding of an element of $\mathbb{Z}_n$, cryptographic libraries usually use their standard multi-precision integer encoding routines[2], while an intuitive encoding routine for binary vectors will not strip leading zero octets. If, however, this latter assumption is violated, then also the encoding of the message representative would be vulnerable in the case of the McEliece cryptosystem.

The next step is the integrity check of the employed CCA2 conversion. Assuming OAEP encoding for RSA, this operation is vulnerable to timing attacks according to [63] if the implementation does not feature appropriate countermeasures. For the McEliece cryptosystem, the conversions proposed in [36, 68] all employ a function "Conv^{-1}", which converts the error vector to a multi-precision integer during the decryption operation. We did not review any concrete propositions for the implementation of this function, but obviously its running time may not depend on the Hamming weight of the error vector. Apart from this open problem, the mentioned conversions do not introduce any obvious potential side channel problems.

Fault Attacks Also referring to Table 4.7, we discuss which of the steps might be vulnerable to fault attacks that reveal a property of the message. For the RSA cryptosystem, the OAEP decoding is basically the only operation that is potentially vulnerable to fault attacks. This is because it is not conceivable that an implementer would implement checks for the number of significant number of octets needed to represent the message at earlier stages of the decryption operation.

In the case of McEliece, however, fault attack vulnerabilities may be present, as we

[2]As a matter of fact, cryptographic libraries usually do not even feature specific objects for the representation of $\mathbb{Z}_n$ elements, but use their multi-precision integers.

General	RSA-OAEP	McEliece
Decryption	...	...
		key equation solving EEA
	Final $\mathbb{Z}_n$ Operation	Root Finding
Message Encoding	Encoding in $\mathbb{Z}_n$	Encoding in $\mathbb{F}_2^n$
CCA2 Check	OAEP Check	appropriate CCA2 Check

Table 4.7.: Comparison of the sources of critical timing differences in the RSA and McEliece cryptosystems. Fields with a gray background indicate potentially vulnerable algorithms.

have seen in Section 4.3. Concerning the CCA2-conversions proposed in [36, 68, 69], none of them explicitly performs any checks on the number of errors during the decryption. But there is one important caveat: the function "Conv^{-1}", already addressed in the above section on timing attacks, is defined as a mapping from elements of $\mathbb{F}_2^n$ with Hamming weight t to integers in the range from 0 to $\binom{n}{t} - 1$. In order to disable fault attacks, this function may not refuse input vectors with a Hamming weight lower than t.

Reaction Attacks For both the RSA and the McEliece cryptosystem, when no CCA2 conversion is applied, message-aimed adaptively chosen ciphertext attacks are known: Bleichenbacher's attack [70] concerning RSA and [71] concerning McEliece. While Bleichenbacher's attack must be viewed as a fault attack, the attack against McEliece given in [71] does not fall into this category. The attack is basically the same as the one described in Section 4.1. But since no conversion is applied at all, it suffices that the attacker can learn whether the decryption of a manipulated ciphertext led to the same plaintext as the original ciphertext, which is the case when $\text{wt}(e) \leq t$. This type of attack, which relies on the observation of semantic reactions of the decrypting party, is called reaction attack.

4.5.1.4. Methodology for the Analysis of public Key Cryptosystems with homomorphic Properties

From the previous sections, we can derive the following simple methodology for the analysis of public key cryptosystems with homomorphic properties: It must be analyzed how any properties of the decrypted plaintext affect both the computations of the core public key decryption routine, and the further computations that take the plaintext as input. If such effects on the computation are observable through a side channel (for instance conditional branching based on this property), then it is highly likely that an

attack can be built. Specifically, one must find a way to control that property of the plaintext through homomorphic manipulations of the ciphertext.

Furthermore, it is helpful to take known reaction or fault attacks into consideration, since they likely might reappear as side channel attacks, as we have seen for the RSA and McEliece PKC.

4.5.2. Blinding Countermeasures for Code-based Cryptosystems

Concerning side channel countermeasures, there is a fundamental difference between RSA and code-based PKCs: while for the former, so-called blinding countermeasures are possible, for the latter these are not possible without altering the algorithm specification, as we shall see in this section. A blinding countermeasure is a random transformation of the operation input, which is undone after the decryption (i.e. the exponentiation in case of RSA). For purposes of illustration, Algorithm 12 shows RSA with base blinding as a valid countermeasure against power analysis and timing attacks [21].

The problem for code based PKCs is that basically no transformation on the ciphertext is known that commutes with the decryption operation. Referring again to the homomorphic properties of both types of PKC, we see that the base blinding exploits the homomorphic property of RSA. Thus, the equivalent transformation for a code-based PKC is given by the addition of further error positions, which, certainly, is a homomorphic operation only if the total number of errors does not exceed t. Consequently, to enable a blinding countermeasure in a code-based PKC, one would have to change the algorithm specification: during the encryption, only $t-1$ errors are added, and prior to the standard decryption operation another "bit flip error" is applied, the position of which should be the same for repeated decryptions of a certain ciphertext, but otherwise appear as random, and thus should be pseudorandomly derived from the ciphertext and a constant secret value (for instance a hash value of the secret key). This approach would guarantee a pervasive alteration of the decryption operation, however it demands an increase of security parameters to compensate for the lower error weight used during encryption.

Algorithm 12 RSA decryption with base blinding side channel countermeasure

Input: RSA ciphertext c, modulus n, public exponent e and private exponent d
Output: RSA plaintext m

$r \leftarrow$ random number
$c' \leftarrow r^e c \mod n$
$m' \leftarrow c'^d \mod n$
$m \leftarrow m' r^{-1} \mod n$

5. Embedded Implementations of the McEliece PKC

In this chapter, we present two embedded implementations of the McEliece PKC along with various performance data.

5.1. A Flexible Platform independent Implementation of the McEliece PKC

In this section, we describe a platform independent implementation of the McEliece PKC providing considerable flexibility in the choice of algorithms used in the decryption.

5.1.1. Description of the Implementation

The implementation was created based on the HyMES open source implementation [14, 20] of the McEliece scheme. The HyMES implementation can be considered the state-of-the-art open source implementation of this PKC, putting an emphasis on computation speed. It features the Berlekamp Trace Algorithm (*BTA-rf*) as the root-finding algorithm, uses a precomputed square root matrix for the computation of the square root in $\mathbb{F}_{2^m}[Y]\backslash g(Y)$ in Algorithm 1, Step 3 and lookup tables for the $\mathbb{F}_{2^m}$ operations. It deviates from the original McEliece specification by encoding information also in the error vector. In [14], only code parameter sets with $n = 2^m$ are analysed; we found, however, that HyMES also supports lengths $n < 2^m$.

We removed from the HyMES implementation the encoding of information in the error vector, as for the most widespread application of a PKC, which is the encryption of symmetric keys, the message size of the original McEliece scheme is sufficient (see Section 3.4.3) and the constant weight word encoding poses thus an unnecessary burden in terms of code size. Further changes made to this implementation are the removal of a number of fault attack vulnerabilities (see Section 4.3), the optimization of the existing single root finding variant *BTA-rf*, the addition of further root finding variants (see Section 3.3), and the implementation of the decryption without the parity check matrix according to Section 3.2. Furthermore, the timing attack countermeasures described in Section 4.1.2.2, and 4.2.2.2 were implemented.

5.1.2. Performance Results

Table 5.1 shows the performance measurement results of the McEliece decryption of our McEliece implementation on an embedded Atmel AT32 AP7000 32-bit CPU mounted

param.	r.f. algo.	time/ms			memory usage / bytes			
		w/H	w/o H	only r.f.	code	private key & $\mathbb{F}_{2^m}$ tables	stack	heap
$n =$ 1632, $t = 33$, 74 bit security	*eval-rf*	47	88	38	23,527	91,468 | 13,132	584	425
	eval-div-rf	30	71	21	23,527	91,468 | 13,132	584	425
	BTA-rf	36	77	27	28,745	91,468 | 13,132	1,744	3,975
	BTZ_2-rf	31	72	22	30,449	95,568 | 17,232	1,588	3,975
	dcmp-rf	34	75	25	29,230	91,468 | 13,132	584	425
	dcmp-div-rf(1,19)	29	70	20	29,230	91,468 | 13,132	636	485
$n =$ 2960, $t = 56$, 122 bit security	*eval-rf*	142	284	113	23,527	275,114 | 26,474	720	662
	eval-div-rf	90	232	61	23,527	275,114 | 26,474	720	662
	BTA-rf	100	242	71	28,745	275,114 | 26,474	1,984	9,266
	BTZ_2-rf	82	224	53	30,449	283,310 | 34,670	1,876	9,266
	dcmp-rf	95	237	66	29,230	275,114 | 26,474	724	662
	dcmp-div-rf(1,19)	77	219	48	29,230	275,114 | 26,474	776	820
$n =$ 6624, $t = 115$, 244 bit security	*eval-rf*	1,430	2,681	1,269	23,527	1,306,534 | 61,622	1,076	1,268
	eval-div-rf	799	2,050	638	23,527	1,306,534 | 61,622	1,076	1,268
	BTA-rf	500	1,751	339	28,745	1,306,534 | 61,622	2,444	32,542
	BTZ_2-rf	433	1,684	272	30,449	1,322,922 | 77,610	2,344	32,542
	dcmp-rf	576	1,827	415	29,230	1,306,534 | 61,622	1,076	1,268
	dcmp-div-rf(3,19)	422	1,673	261	29,230	1,306,534 | 61,622	1,128	1,668

Table 5.1.: Performance results of the McEliece decryption on the AVR32 AP7000 CPU for three different code-parameter sets. The column labels have to be interpreted as follows:
"param.": the code parameters including the security level
"r.f. algo.": the root-finding algorithm that is used during the decryption
"time / ms": time of the respective operation in milliseconds
"w/H": decryption with the use of the parity check matrix
"w/o H": decryption without the parity check matrix
"only r.f." root-finding operation during the decryption
"memory usage / bytes": the usage of various types of memory
"code": the program code
"private key & $\mathbb{F}_{2^m}$ tables": the private key size including precomputations with and without the parity check matrix
"stack", "heap": the usage of stack and heap memory (RAM) during the program execution
Note that the provided code size encompasses encryption and decryption, but not key generation. The timings are given for a CPU frequency of 30 MHz.

on an ICnova AP7000 Base board running an embedded Linux. The source code was compiled with GCC version `4.2.2-atmel.1.1.3.avr32linux.1` and compiler options

```
-O3 -march=ap -mcpu=ap7000 -fomit-frame-pointer -finline-functions
```

The timings were measured directly on the AP7000 running at 150 MHz. The timings given in the table, however, are scaled down to a hypothetical CPU frequency of 30 MHz, as this is a typical smart card frequency. For all performance data, 50 runs were executed: for the timings, the mean was taken, for the stack usage, the maximum was taken (this is only relevant for *BTA-rf* and BTZ_2*-rf*, the other root-finding variants lead to deterministic stack usage of the decryption). The heap usage is constant for a given parameter set for any specific algorithmic choice.

The three code parameter sets used in the performance measurement are proposed in [52] for minimal public key sizes at a given security level. However, as already pointed out in Section 3.3.3, there the addition of more than t errors is assumed, which is not supported by our implementation. This implies a reduction of security for which we give the respective lower bounds based on considerations given in Section 3.3.3.

In the second column of Table 5.1, we give the respective root-finding variant from Section 3.3.2 employed during the decryption. In the next two columns, the running times of the complete decryption is given with and without the parity check matrix. The following column shows the running time of the root-finding alone. Comparing the results of the two larger parameter sets with the ones given in Table 3.2 on an Intel Core2 Duo U7600 CPU, we find considerable differences concerning the improvements brought by the *BTA-rf* and *dcmp-rf* variants and the hybrid methods derived from them over the primitive *eval-rf* and *eval-div-rf*: while for $n = 2960$, $t = 56$, on the Intel CPU *BTA-rf* is about twice as fast as *eval-div-rf*, it is even slower than the latter on the AP7000. Similarly, while three times faster than *eval-div-rf* on the Intel CPU, on the AP7000 *dcmp-rf* is slower than *eval-div-rf*. However, for the largest parameter set, on the AP7000, *BTA-rf* is again considerably faster than *eval-div-rf*, and, in contrast to the results for the Intel CPU, it is also faster than *dcmp-rf*. The actual reasons for this platform dependency of the performance ranking were not investigated; it is however clear that both CPUs are fundamentally different, for instance the low-end AP7000 features much lower clock rates (potentially reducing the cost of memory loads and stores) and much less instruction level parallelism than the high end Intel processor.

The remaining columns provide memory usage measurements: The code size, which is independent of the code-parameter choice, encompasses encryption and decryption, but not key generation. Furthermore, for the variants employing division, the code size is equal to the respective variant without division, since also in that case the function implementing polynomial division was compiled as it only yields a minimal increase of code size. The same applies to the code for computation of the syndrome with or without the parity check matrix. The removal of unused code was only applied to larger functional blocks that considerably affect the code size.

In the following column, the private key size in memory including precomputations with and without the parity check matrix are given. On the evaluated platform, the private key is held in RAM on the heap memory; however, since for instance on a smart

card, it would be held in non-volatile memory, we separated it from the runtime heap usage. *BTZ_2-rf* is the only root-finding variant that needs more precomputations than the other variants: as mentioned in Section 3.3.2.2, it makes use of a table with a size of $2n$ bytes. Note that the private key sizes given here incur management data, and thus are slightly larger than those from Table 3.1.

What follows are peak stack and peak heap usage by the program during decryption, excluding the input data (i.e. the ciphertext). Concerning the stack usage, however, the given numbers are only approximate due to the way the measurement was performed: as there is no memory profiler available for the AP7000 Linux platform, on any function call (which is performed with an error handling C preprocessor macro basically anywhere in the code anyway), the stack position was measured and the overall maximum was tracked. This leaves leaf functions uncovered, meaning that the actual stack usage is always larger than the measured one. But since there is no excessive stack usage in any of the single functions, the inaccuracy can be estimated to be a few hundred bytes at most. The heap usage only summarizes the maximum allocated memory, ignoring heap management data.

Comparing the stack and heap usage of the different algorithms, we find that the recursive structure of *BTA-rf* causes a considerably higher stack usage than the other algorithms. The excessive heap usage, which must be ascribed to the precomputations of this algorithm, is even more striking, and shows that it is not very suitable for the use on typical smart card platforms, which usually feature less than 20kB of RAM.

In summary, we find that decryption using *dcmp-rf* provides promising results for all parameter sets. For the lower two parameter sets, the decryption without the parity check matrix is less than 300 ms, and thus acceptable for many purposes in the smart card application context. This statement is made under the assumption that, in this context, the primary constraint is that the decryption time should not perceptibly increase the whole process of the private operation, which usually includes the PIN entry through the user.

For the 244 bit parameters, we find that the same algorithmic choice causes a decryption time of almost two seconds. Since for these parameters, the parity check has a size of more than one megabyte, it is inconceivable to use this precomputation to speed up the computation on resource-constrained devices. Thus, this parameter choice calls for hardware support to speed up at least the most time consuming part, which is the syndrome computation, here.[1]

Note that these timing results of our pure software implementation are very good compared with the state-of-the-art smart cards implementing RSA decryption using cryptographic coprocessors: The Infineon Technologies AG smart card controller family supports RSA decryption with a modulus of 1024 bit in 136 ms [39]. However, according to [72], for 256 bit security, which is close to that of our largest parameter set, one would have to use a modulus of 15,360 bit. The linear increase of the computation time due to the increased exponent size (by a factor of 15) alone already yields a time of more than two seconds on that platform. However, the larger complexity of the

[1] This is obvious from the large difference between the timings with and without the parity check matrix.

code parameters	encryption time @ 30 MHz / ms	public key size / bits
$n = 1632$, $t = 33$	12.4	460,647
$n = 2960$, $t = 56$	34.4	1,537,536
$n = 6624$, $t = 115$	154.5	7,667,855

Table 5.2.: Timings for the McEliece encryption on the AVR32 AP7000 CPU.

multiplication operations during the exponentiation would incur another tremendous increase in computation time – given that one had a cryptographic coprocessor that is able to handle operands of this size. Thus, our McEliece timing results for pure software implementations already prove the clear superiority in terms of decryption time of the scheme over RSA for high security parameter choices.

The stack and heap usage are both acceptable for decryption with *dcmp-rf*, even for the large code parameters.

For the sake of completeness, we also give the running time of the encryption operation in Table 5.2. However, it features no significant change compared to the original HyMES implementation. The only difference is that the error vector is not derived from the message, but created randomly. This implementation stores the public key in RAM; it does not make use of the on-line public operation introduced in Section 3.1. The memory usage of the encryption is rather irrelevant since beyond the error vector and the ciphertext, no other intermediate values are involved. Thus, it will always be smaller than that of the encryption.

5.2. A Smart Card Implementation of the McEliece PKC

Another implementation that was created in the course of this work is an implementation on an actual smart card [4]. As the predecessor of the implementation given in Section 5.1, it provides no algorithmic alternatives in the decryption operation.

5.2.1. Description of the Implementation

The implementation employs the $\mathbb{F}_{2^m}$ of the HyMES open source implementation [20], but it is independent otherwise. Concerning algorithm choices, however, there are a few differences to the implementation presented in Section 5.1. First, the optimization with respect to finite field representations (Section 3.3.1) were not applied to the smart card implementation. It features only a single root-finding variant, which is *eval-div-rf*, is restricted to code lengths $n = 2^m$ and performs the decryption with the parity check matrix.

In contrast to the flexible implementation, it features a CCA2 conversion, namely the one presented in [69]. The encryption and decryption algorithms of the resulting CCA2-secure scheme are given in Algorithms 13 and 14.

The message and ciphertext sizes of this scheme are given in Table 5.3. The security level is given with respect to the attack given in [52].

Algorithm 13 McEliece - CCA2 secure encryption

Input: message $\vec{m} \in \mathbb{F}_2^l$, public key $\mathbf{G}^{\text{pub}}$
Output: ciphertext $\vec{z} \in \mathbb{F}_2^{n+2l}$
$\vec{u}_1 \leftarrow$ random $(k-l)$-bit string.
$\vec{u}_2 \leftarrow$ random l-bit string.
$(\vec{z}_1, \vec{e}) \leftarrow \mathcal{E}_{\mathbf{G}^{\text{pub}}}(\vec{u}_1 \| H(m \| \vec{u}_2))$

$$z \leftarrow \left(\vec{z}_1 \;\|\; \underbrace{(H(\vec{u}_1) \oplus m)}_{\vec{z}_2} \;\|\; \underbrace{(\vec{u}_2 \oplus H(e))}_{\vec{z}_3} \right)$$

Algorithm 14 McEliece - CCA2 secure decryption

Input: ciphertext $\vec{z} = (\vec{z}_1, \vec{z}_2, \vec{z}_3) \in \mathbb{F}_2^{n+2l}$, secret key $(P, g(X))$
Output: decrypted message $\vec{m} \in \mathbb{F}_2^l$
$(\vec{w}, \vec{e}) \leftarrow \mathcal{D}_{(P,g(X))}(\vec{z}_1)$
$\vec{r} \leftarrow$ the first $k-l$ bits of $\vec{w}$
$\vec{h} \leftarrow$ the bits at $k-l+1, \cdots, k$ of $\vec{w}$.
$\vec{m} \leftarrow \vec{z}_2 \oplus H(\vec{r})$
if $\vec{h} = H(\vec{m} \| (H(\vec{e}) \oplus \vec{z}_3))$ **then**
  return $\vec{m}$
else
  return error
end if

n,t	**security bits**	**message size in byte**	**ciphertext size in byte**
1024,40	62	32	$2 \cdot 32 + 128 = 192$
2048,50	102	32	$2 \cdot 32 + 256 = 320$

Table 5.3.: Security parameter sets for the CCA2-secure McEliece PKC

platform	parameter set	operation	time	time without I/O
PC	$n = 1024$, $t = 40$	decryption	0.8 ms	-
SLE76	“	decryption	0.98s	0.69s
PC	$n = 2048$, $t = 50$	decryption	1.6ms	-
SLE76	“	decryption	1.52s	1.06s

Table 5.4.: Timings for the decryption operation of the McEliece PKC on a personal computer and the SLE76 smart card

As hardware platform, we used an SLE76CF5120P controller out of the SLE76-family [38] by Infineon Technologies AG. It features a 16 bit CPU based on the 80251 architecture. It has a clock rate of 33 MHz and provides 12 kByte of RAM. It es equipped with 504 kByte of non-volatile memory (NVM, i.e. flash memory). It also features a unified data and code cache of 1 kByte.

5.2.2. Performance Results

In Table 5.4, we give timings for the two parameter sets. For comparability, we also give timings for the same operations on a PC. The computer is an Intel Core Duo T7300 2GHz running Linux with kernel version 2.6.24. The application uses the same source code as the smart card implementation, compiled with GCC-4.1.3, optimization level `O2`.

The column labeled “time” lists the overall timing including the data transmission to and from the smart card. In the rightmost column, we provide the time that is used by the mere computation on the card, excluding the transmission times. The gross bit rate of the transmission is 9600 bit/s. Please note that the SLE76 hardware platform generally supports much faster transmission rates than this.

We do not give any encryption timings, as the mere encryption timing is irrelevant on a smart card platform, since in that case the approach given in Section 3.1, the on-line public operation would have to be applied, as is discussed at length in the referenced section. The timing results are less impressive than those from the implementation presented in Section 5.1. The main two reasons for this is the far less advanced algorithmic choice in this implementation and the fact that the platform features only a 16 bit CPU. However, for the parameters providing 102 bit security, which are of practical relevance, we have decryption times in the order of one second, which can be considered sufficient for certain applications. In this context, we wish to point out that a major improvement of the decryption time should result from the replacement of 32 bit pointers used throughout the code by 16 bit pointers. This is because the 16 bit CPU can handle the smaller pointers much faster. But since, at least for the larger parameter set, the private

resource	space in 10^3 byte
RAM ($m = 2048$,$t = 50$)	4.4
RAM ($n = 1024$,$t = 40$)	3.4
NVM code	45
NVM private key (n=1024,t=40)	60
NVM private key (n=2048,t=50)	151
NVM $\mathbb{F}_{2^{10}}$ lookup tables	4
NVM $\mathbb{F}_{2^{11}}$ lookup tables	8

Table 5.5.: Resource demands of the McEliece PKC smart card implementation with accuracy of 100 byte for RAM and 1000 byte for NVM

key size exceeds the 16 bit addressable area, this could only be achieved with the usage of the Memory Management Unit (MMU) available on the SLE76 platform.

In Table 5.5, we give the resource demands for the decryption operation, i.e. the RAM and non-volatile memory (NVM) space needed by the implementation. Again, we distinguish the two parameter sets. The demanded RAM size is made up of a fixed stack size of 1024 bytes and the peak amount of allocated heap memory. The main contribution to the private key size stems from the parity check matrix H, which makes up about $143,000$ bytes in case of $m = 11$, $t = 50$ and about $53,000$ bytes for $m = 10$, $t = 40$. This corresponds to portions of 95% and 88%, respectively. Please note that in addition to the raw matrix data, the given sizes also include certain management data overhead.

6. Open Problems

In this chapter, we address research problems that remain open in this thesis. Their solutions will become relevant at the point where concrete implementation choices of code-based PKCs are evaluated for the suitability in real world applications.

6.1. Potential Cache-Timing Vulnerabilities in Code-Based Decryption Operations

Cache-timing attacks against AES [23] are well known. In these attacks, so-called cache collisions in the lookup tables, which make AES software implementation fast, are exploited: when a table entry is accessed, and is found in the same cache line as a previously accessed entry, then this entry will be loaded faster, as it is already contained in the cache.

Since the $\mathbb{F}_{2^m}$ operations are implemented in terms of lookup tables (refer to Section 3.3.1), the danger of cache-timing attacks is potentially given for code-based PKCs as well. However, so far no concrete attacks of this type have been proposed against these schemes. The main reason for this can be seen in the fact that unlike in the case of AES, at least in an implementation using the parity check matrix, neither are the operands of the $\mathbb{F}_{2^m}$ operations controllable through the input nor does the plaintext contain immediate results of these operations. Also, a multitude of other timing differences, for instance the variable number of iterations in the EEA applications and furthermore their variable complexity must be assumed to considerably veil the targeted cache-timing effects of a hypothetical attack. Thus, it remains an open question, whether cache-timing attacks against the finite field arithmetics are possible or not in code-based PKCs.

The same question applies to the vulnerability of the lookups into the table for support Γ and Γ^{-1}, respectively. While, in principle, shorter timings can be assumed if more lookups in the same cache line are performed, also here, other timing effects in the code-based decryption operation greatly veil this potential vulnerability.

6.2. Countermeasures Against the Low-Weight Error Vector Attacks

The question of countermeasures against the attack presented in Section 4.2.1 has not been explicitly addressed in this work, but three possibilities are suggested here: The first is a generic countermeasure against timing attacks that works if a platform provides the program with exact control of timing delays in the algorithm. Given this control,

upon detection of a number of roots that is smaller than $t/2$, the program can simply enforce a previously determined worst case running time for these cases, thus providing no information to the attacker. One disadvantage of this approach is its insufficiency against power analysis attacks, because the distinction of an artificial delay from the actual decryption operation from the power trace seems highly likely, and thus this type of attack is not mitigated by the countermeasure. Another problem is that the worst case running time (be it in terms of cycles or time) is highly platform dependent and must be determined at run time rather than compilation time in a universal implementation. If this value shall be available for the next program invocation, the implementation of the cryptographic algorithm must have access to the device's non-volatile memory (e.g. the hard disk), complicating the integration of the implementation.

The second would be similar to the countermeasures given in Section 4.1.2.2 against the attack presented in Section 4.1.2.1, where "premature" abortion of the key equation solving EEA is prevented by enforcing the "missing" iterations. This is, however, a delicate undertaking, as even the smallest timing differences have to be prohibited, and thus the complexity of the individual iterations must be accounted for (consider for instance the "$w = 1$ attacks" from Section 4.2.1.5).

The third option would be to use the blinding-like countermeasure for code-based PKCs proposed in Section 4.5.2. This would be the simplest option providing security against key-aimed timing attacks, and in principle also against related power analysis attacks. However, another open question is that of the compatibility of the blinding-like countermeasure with the countermeasure from Section 4.1.2.2, which defeats the message-aimed attacks in the key equation solving EEA: this latter countermeasure relies on a proper ciphertext featuring exactly t errors, which is not guaranteed anymore in case the blinding-like countermeasure proposed in Section 4.5.2 is applied.

6.3. Side Channel Security of BTA-rf

We applied an analogous analysis to that of Section 4.1.1 to *BTA-rf* as realized in the HyMES [20] open source implementation of the McEliece scheme on the ATMega1284P platform. The timing results of the scan across different error weights given in Figure 6.1 show that the mean of the running times for $w = t$ is below most of the minimal values of sets for $w \neq t$, clearly indicating a vulnerability. Obviously, the recursive algorithm behaves differently when $\sigma(Y)$ has considerably fewer than t roots (the countermeasures from Section 4.1.2.2 lead to this also for $w < t$). Thus, message-aimed timing attacks similar to those from Section 4.1.2 are possible. The actual reasons for this behaviour as well as the question of countermeasures remain open.

Figure 6.2 shows the plots of the dependencies of the running time of the root-finding with *BTA-rf* on the position of a single error bit. The plots were created in the same way as those given in Figure 4.8 in Section 4.2.2.2. We see some "clouding" effect in the running times, which is also apparent for timings of the whole syndrome decoding, as shown in Fig. 6.2(a). It is obvious that these running times are neither constant nor random. There seems to be a tendency to build "clouds", by which we mean that it

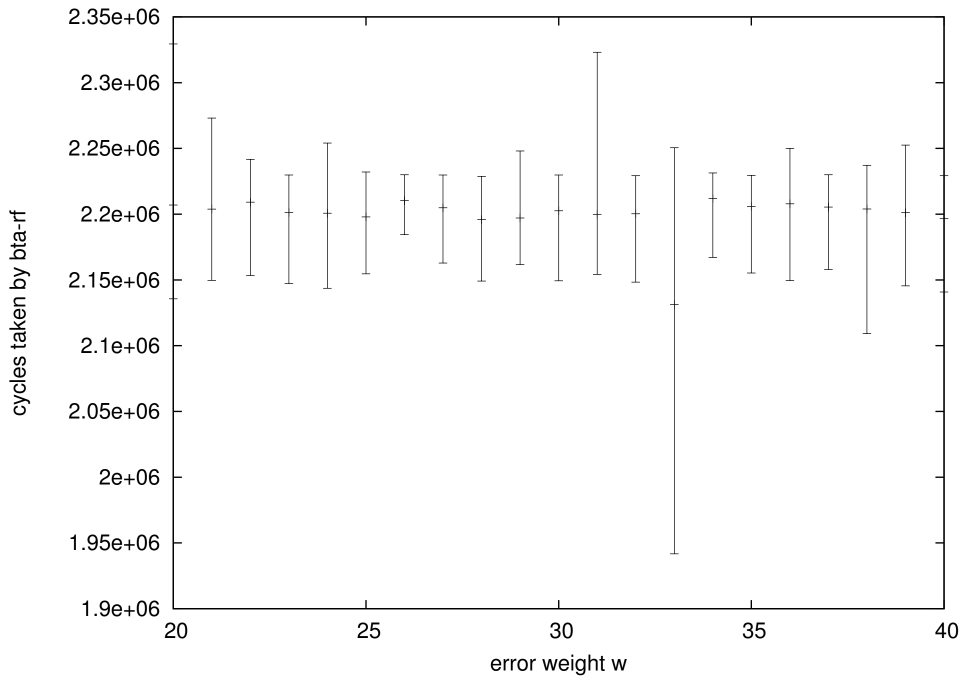


Figure 6.1.: Cycle counts taken on an ATMega1284P for the *BTA-rf* algorithm with parameters $n = 512$ and $t = 33$. The error bars denote minimal and maximal values, the center mark represents the mean. For each value of w, 30 different syndromes were decoded.

seems like an attacker should be able to build hypotheses that if for two different values of E_1 and E_2 the timings are close to each other, then also α_{E_1} and α_{E_2} have close values in their lexicographical interpretation as numbers.

Note for instance the values of E_t below 100 in 6.2(a), which have consequently lower timings than $3.04 \cdot 10^6$. Though such a dramatic effect was not obvious in all support scans we conducted, it corroborates the notion of "clouding" effects in the timings for *BTA-rf*. Thus, we strongly suggest that the running time properties of the *BTA-rf* be subject to thorough analysis before considering its use in real world implementations of code-based schemes.

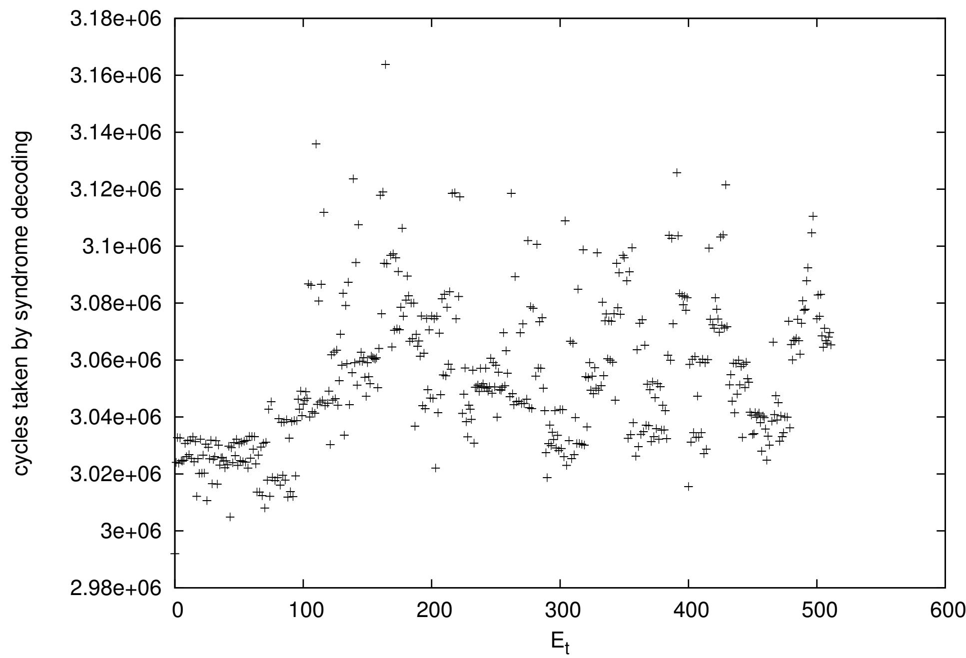
6.4. Side-Channel Secure Implementation of dcmp-div-rf

Since on both evaluated platforms, the x86, an Intel Core2 Duo CPU, and the Atmel AVR32 AP7000, *dcmp-div-rf* turns out to be the fastest, it would be interesting to have a side channel secure implementation of this algorithm, i.e. remove the vulnerabilities presented in Sections 4.1.1 and 4.2.2.1 for *eval-div-rf*, which extend to *dcmp-div-rf*.

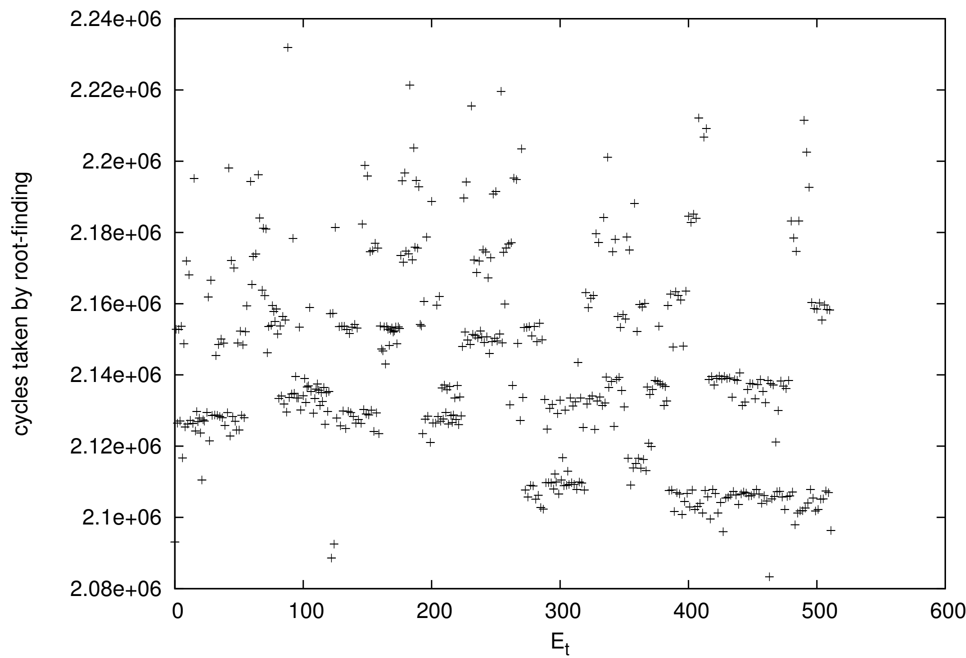
Concerning the message-aimed attacks from Section 4.1.1, which build on distinguishing the root-finding execution for an error locator polynomial $\sigma(Y)$ of degree t having t roots or only a fraction of this based on the timing, a generic countermeasure could in principle be implemented on certain platforms: the worst case running time of the root-finding for a proper ciphertext, which we shall refer to as T_{proper}, had to be determined. Then, for a proper ciphertext, after the completion of the root-finding, an artificial delay had to be created until T_{proper} is reached. Else, for an irregular ciphertext, the root-finding is broken after T_{proper} and some previously determined fake set of roots is returned. Note that the distinction of proper and improper ciphertexts at this point is only implicitly given through the number of actual roots of $\sigma(Y)$ of degree t.

One problem of this approach is that it can hardly be used to defeat power-analysis attacks, as the distinction of the execution of the continued root-finding or the artificial delay from the power trace is most likely possible. But for the case where merely timing attacks are addressed, there is also a fundamental problem: this is the determination of T_{proper}. This worst case running time would be given by the ciphertext where the t roots are found at the very end of the evaluation. This means that the first $n - t$ evaluations find no roots at all, and thus there is almost no speedup in contrast to the sole evaluation method without division (*dcmp-rf* or *eval-rf*, this analysis equally applies to both). Consequently, in order not to lose the speedup gained by the division, the worst case distribution of the roots underlying the determination of T_{proper} must be adjusted. This, however, means that there will be valid ciphertexts that are rejected by the decryption routine, which is clearly an undesirable feature. The only way to circumvent this problem is to enforce the worst case running time only after the CCA2 integrity check has failed.

The defence against the key-aimed attacks presented in Section 4.2.2.1 also poses a problem: while for *eval-div-rf*, as explained there, the vulnerability can be completely avoided if the evaluation is performed in the order $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$, this is not so simple



(a) Timings for syndrome decoding with *BTA-rf*.



(b) Timings for root-finding with *BTA-rf*.

Figure 6.2.: Running times of *BTA-rf* for $n-(t-1)$ ciphertexts, where $t-1$ error positions are fixed and the $t-th$ position varies, with code parameters $n = 512$ and $t = 33$.

for *dcmp-div-rf*. The reason is that, in this order of evaluation, a Gray code is not given in general. Thus, the question arises, whether sufficient security against mathematical attacks could be achieved by restricting the choices for the support Γ by demanding that it in whole or in parts be in Gray-code ordering.

6.5. The Problem of the Optimal Root-Finding Algorithm for Embedded Implementations with Hardware Support

Another task in the course of preparing code-based PKCs for real world applications is the implementation of code-based cryptosystems on smart cards and related platforms. Since code-based schemes are, if they will ever be used in the field, prone to high security applications, it is important that smart cards achieve practical running times for code parameters providing 256 bit security or higher. From the results of Section 5.1.2, we find that in order to achieve this, hardware support would have to be present on these platforms, as it is the case for RSA and elliptic curve based algorithms today. Thus, in view of the efficiency issues presented in Section 3.3 and the root-finding related side channel problems analysed in Section 4, the real question is that of an optimal choice of algorithms and hardware support, achieving both good performance and side-channel security on these platforms. In this context, among other aspects, it will become relevant how easily an algorithm can be parallelised. Note that *eval-rf*, *eval-div-rf*, and *dcmp-rf* can easily be parallelised by starting independent evaluations at equally distant offsets into $\mathbb{F}_{2^m}$ (in the Gray-Code order for *dcmp-rf*). However, the circuitry for any single instance of an *eval-rf* evaluator would be considerably simpler than for *dcmp-rf*. The parallelisation of *BTA-rf* seems the most complicated; it would have to be applied to the recursive structure of the algorithm. In view of these open questions, we encourage future research investigating implementations with efficient hardware support on resource constrained platforms.

7. Conclusion

In this thesis, we have presented various improvements to the efficiency of code-based PKCs as well as side channel vulnerabilities in these cryptosystems and corresponding countermeasures. Concerning efficiency issues, we have shown that on embedded systems such as smart cards, the speed of the code-based public key encryption operation with the McEliece or Niederreiter scheme is limited by the transmission speed towards the embedded device. This is an important point, because unlike for the algorithmic tasks of the decryption operation, it is not possible to simply add sufficient hardware support to reach a desired timing of the operation. The change of an interface of a smart card always incurs the upgrade of the smart card reader infrastructure, thus posing a potentially far more costly undertaking than the development and rollout of a smart card featuring a new cryptographic coprocessor. Also, we have shown that the parity check matrix as a precomputed value for the McEliece decryption is not necessary to achieve competitive computation times, and thus a large amount of non-volatile memory can be saved on embedded devices implementing this scheme.

Furthermore, we have presented a comparison of known root-finding algorithms as well as new hybrid variants in terms of performance, and a discussion of their security. Being the only part of the code-based PKC decryption operation that allows for a significant algorithmic variety, it is an important choice to make in any implementation. We chose to use code-parameters that minimize the public key size for a given security level in this evaluation, because we believe that this is the most problematic aspect of these cryptosystems. Thus, we believe to have provided a significant contribution concerning PC and embedded software implementations of code-based cryptosystems, which can be followed up on by hardware solutions for embedded devices.

Concerning side-channel security, we have shown a multitude of timing side channel vulnerabilities. Besides the various problems that can result from certain algorithms for the task of the root-finding, we pointed out timing vulnerabilities in the key equation solving EEA and evaluation of the error locator polynomial that allow practical attacks targeting the message. Furthermore, we provided a practical timing attack employing three different types of vulnerabilities allowing the recovery of the secret key. We also addressed the problems of fault attack vulnerabilities in an actual implementation of the McEliece PKC.

Concluding our contributions, we presented two embedded implementations, providing the memory demands and timing performances of the various implementation alternatives analysed and developed in this work. This provides a profound basis for the choice of the platform and algorithms for the implementation of the McEliece or Niederreiter PKC.

Finally, we addressed remaining open problems, which pose further research questions

that, in our opinion, should be addressed in the course of the evaluation of the suitability of code-based PKCs as a solution for data protection in a post-quantum cryptographic world.

Bibliography

[1] Strenzke, F., Tews, E., Molter, H., Overbeck, R., Shoufan, A.: Side Channels in the McEliece PKC. In Buchmann, J., Ding, J., eds.: Post-Quantum Cryptography. Volume 5299 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 216–229

[2] Shoufan, A., Strenzke, F., Molter, H., Stöttinger, M.: A Timing Attack against Patterson Algorithm in the McEliece PKC. In Lee, D., Hong, S., eds.: Information, Security and Cryptology - ICISC 2009. Volume 5984 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2009) 161–175

[3] Strenzke, F.: A Timing Attack against the secret Permutation in the McEliece PKC. In: The third international Workshop on Post-Quantum Cryptography PQCRYPTO 2010, Lecture Notes in Computer Science

[4] Strenzke, F.: A Smart Card Implementation of the McEliece PKC. In: Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices. Volume 6033 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2010) 47–59

[5] Strenzke, F.: Message-aimed Side Channel and Fault Attacks against Public Key Cryptosystems with homomorphic Properties. Journal of cryptographic Engineering (2011) DOI: 10.1007/s13389-011-0020-0; a preliminary version appeared at COSADE 2011 .

[6] Molter, H.G., Stöttinger, M., Shoufan, A., Strenzke, F.: A Simple Power Analysis Attack on a McEliece Cryptoprocessor. Journal of Cryptographic Engineering (2011)

[7] Strenzke, F.: Fast and secure root finding for code-based cryptosystems. In Pieprzyk, J., Sadeghi, A.R., Manulis, M., eds.: Cryptology and Network Security, CANS 2012. Volume 7712 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 232–246

[8] Strenzke, F.: Solutions for the Storage Problem of McEliece Public and Private Keys on Memory-Constrained Platforms. In: Proceedings of the 15th international conference on Information Security, ISC 2012. Lecture Notes in Computer Science, Berlin, Heidelberg, Springer-Verlag (2012) 120–135

[9] Strenzke, F.: Timing Attacks against the Syndrome Inversion in Code-Based Cryptosystems. In Gaborit, P., ed.: Post-Quantum Cryptography. Volume 7932 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 217–230

[10] Peter W. Shor: Polynomial Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing **26(5)** (1997) 1484–1509

[11] R. J. McEliece: A Public Key Cryptosystem Based on Algebraic Coding Theory. DSN progress report **42–44** (1978) 114–116

[12] Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. In: Problems Control Inform. Theory. Volume Vol. 15, number 2. (1986) 159–166

[13] Bernstein, D.J., Buchmann, J., Dahmen, E.: Post Quantum Cryptography. Springer Publishing Company, Incorporated (2008)

[14] Biswas, B., Sendrier, N.: McEliece Cryptosystem Implementation: Theory and Practice. In: PQCrypto. (2008) 47–62

[15] Heyse, S.: Low-Reiter: Niederreiter Encryption Scheme for Embedded Microcontrollers. In Sendrier, N., ed.: Post-Quantum Cryptography. Volume 6061 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2010) 165–181

[16] Eisenbarth, T., Güneysu, T., Heyse, S., Paar, C.: MicroEliece: McEliece for Embedded Devices. In: CHES '09: Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems, Berlin, Heidelberg, Springer-Verlag (2009) 49–64

[17] Shoufan, A., Wink, T., Molter, G., Huss, S., Strenzke, F.: A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms. In: ASAP '09: Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, Washington, DC, USA, IEEE Computer Society (2009) 98–105

[18] Heyse, S., Moradi, A., Paar, C.: Practical power analysis attacks on software implementations of mceliece. In Sendrier, N., ed.: PQCrypto. Volume 6061 of Lecture Notes in Computer Science., Springer (2010) 108–125

[19] Avanzi, R., Hoerder, S., Page, D., Tunstall, M.: Side-channel attacks on the mceliece and niederreiter public-key cryptosystems. J. Cryptographic Engineering **1**(4) (2011) 271–281

[20] Biswas, B., Sendrier, N.: HyMES - an open source implementation of the McEliece cryptosystem (2008) `http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`.

[21] Kocher, P.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (1996) 104–113

[22] Kocher, P., Jaff, J., Jun, B.: Differential Power Analysis. Advances in Cryptology-CRYPTO'99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings **1666** (1999) 388–397

[23] Bernstein, D.J.: Cache-timing attacks on aes. Technical report (2005)

[24] Berger, T.P., Cayrel, P.L., Gaborit, P., Otmani, A.: Reducing Key Length of the McEliece Cryptosystem. In: AFRICACRYPT '09: Proceedings of the 2nd International Conference on Cryptology in Africa, Berlin, Heidelberg, Springer-Verlag (2009) 77–97

[25] Berger, T.P., Loidreau, P.: How to Mask the Structure of Codes for a Cryptographic Use. Designs, Codes and Cryptography **35** (2005) 63–79 10.1007/s10623-003-6151-2.

[26] Misoczki, R., Barreto, P.: Compact McEliece Keys from Goppa Codes. In Jacobson, M., Rijmen, V., Safavi-Naini, R., eds.: Selected Areas in Cryptography. Volume 5867 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 376–392

[27] Otmani, A., Tillich, J.P., Dallot, L.: Cryptanalysis of Two McEliece Cryptosystems Based on Quasi-Cyclic Codes. Mathematics in Computer Science **3** (2010) 129–140

[28] Faugère, J.C., Otmani, A., Perret, L., Tillich, J.P.: Algebraic Cryptanalysis of McEliece Variants with Compact Keys. In: In Proceedings of Eurocrypt 2010. (2010)

[29] Umana, V.G., Leander, G.: Practical key recovery attacks on two McEliece variants. In: SCC 2010: Proceedings of the Second International Conference on Symbolic Computation and Cryptography. Royal Holloway, University of London, Egham, UK, 23–25, Carlos Cid, Jean-Charles Faugere (editors) (2010) 27–44

[30] Faugère, J.C., Otmani, A., Perret, L., Tillich, J.P.: A Distinguisher for High Rate McEliece Cryptosystems. In: Information Theory Workshop (ITW), 2011 IEEE, IEEE (2011) 282–286

[31] Courtois, N., Finiasz, M., Sendrier, N.: How to Achieve a McEliece-Based Digital Signature Scheme. In Boyd, C., ed.: Advances in Cryptology - ASIACRYPT 2001. Volume 2248 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2001) 157–174

[32] Goppa, V.D.: A new class of linear correcting codes. Problems of Information Transmission **6** (1970) 207–212

[33] F. J. MacWilliams and N. J. A. Sloane: The Theory of Error Correcting Codes. North Holland (1997)

[34] Patterson, N.: Algebraic decoding of Goppa codes. IEEE Trans. Info.Theory **21** (1975) 203–207

[35] Berson, T.: Failure of the McEliece Public-Key Cryptosystem under Message-Resend and Related-Message Attack. In Kaliski, B., ed.: Advances in Cryptology — CRYPTO '97. Volume 1294 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1997) 213–220 10.1007/BFb0052237.

[36] Kobara, K., Imai, H.: Semantically Secure McEliece Public-Key Cryptosystems - Conversions for McEliece PKC. Practice and Theory in Public Key Cryptography - PKC '01 Proceedings (2001)

[37] Sendrier, N.: Encoding information into constant weight words. In: Information theory 2005, IEEE (2005) 435–438

[38] Infineon Technologies AG: SLE76 Product Data Sheet `http://www.infineon.com/cms/de/product/channel.html?channel=db3a3043156fd57301161520ab8b1c4c`.

[39] Infineon Technologies AG: SLE 66CLX360PE(M) Family Data Sheet `http://www.infineon.com/dgdl/SPI_SLE66CLX360PE_1106.pdf?folderId=db3a304412b407950112b408e8c90004&fileId=db3a304412b407950112b4099d6c030a&location=Search.SPI_SLE66CLX360PE_1106.pdf`.

[40] Cooper et al.: RFC 5280 `http://tools.ietf.org/html/rfc5280`.

[41] Coronado, L.C., Buchmann, J., Carlos, L., Garcia, C., Dahmen, E., Klintsevich, E., Darmstadt, T.U.: CMSS – An Improved Merkle Signature Scheme Johannes Buchmann (2006) `www.cdc.informatik.tu-darmstadt.de/~dahmen/papers/BCDDK06.pdf`.

[42] Witschnig, H., Patauner, C., Maier, A., Leitgeb, E., Rinner, D.: High speed RFID lab-scaled prototype at the frequency of 13.56 MHz. e & i Elektrotechnik und Informationstechnik **124** (2007) 376–383 10.1007/s00502-007-0485-9.

[43] Infineon Technologies AG: SLE78 Product Data Sheet `http://www.infineon.com/cms/en/product/channel.html?channel=db3a30431ce5fb52011d47b166342af0`.

[44] Olivier Gay: `http://www.ouah.org/ogay/sha2/`.

[45] Overbeck, R., Sendrier, N.: Code-based Cryptography. In Bernstein, D., Buchmann, J., Dahmen, E., eds.: Post-Quantum Cryptography. Springer (2009) 95–145

[46] Finiasz, M.: Parallel-CFS: Strengthening the CFS McEliece-based signature scheme. In Biryukov, A., Gong, G., Stinson, D., eds.: Selected Areas in Cryptography. Volume 6544 of LNCS., Springer (2010) 159–170

[47] Kabatianskii, G., Krouk, E., Smeets, B.: A Digital Signature Scheme based on Random Error-Correcting Codes. In Darnell, M., ed.: Crytography and Coding. Volume 1355 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1997) 161–167

[48] Otmani, A., Tillich, J.P.: An Efficient Attack on All Concrete KKS Proposals. In Yang, B.Y., ed.: PQCrypto 2011. Volume 7071 of LNCS., Springer (2011) 98–116

[49] E. R. Berlekamp: Factoring Polynomials over large Finite Fields. Mathematics of Computation **24(111)** (1970) 713–715

[50] Biswas, B., Herbert, V.: Efficient Root Finding of Polynomials over Fields of Characteristic 2. WEWoRK (2009) `hal.inria.fr/hal-00626997/PDF/tbz.pdf`.

[51] Federenko, S., Trifonov, P.: Finding Roots of Polynomials over Finite Fields. IEEE Transactions on Communications **20** (2002) 1709–1711

[52] Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the McEliece cryptosystem. Post-Quantum Cryptography, LNCS **5299** (2008) 31–46

[53] Bernstein, D.J.: List Decoding for binary Goppa Codes. In: Coding and cryptology – third international workshop, IWCC 2011, Qingdao, China, May 30–June 3, 2011, Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing (editors), Lecture Notes in Computer Science 6639, Springer, 2011 (2011) 62–80

[54] The FlexiProvider group at Technische Universität Darmstadt: FlexiProvider, an open source Java Cryptographic Service Provider. Available at `http://www.flexiprovider.de`.

[55] Research Center for Information Security (RCIS): Side-channel Attack Standard Evaluation Board SASEBO-G Specification. Technical report (2008)

[56] Sugiyama, Y., Kasahara, M., Hirasawa, S., Namekawa, T.: A Method for Solving Key Equation for Decoding Goppa Codes. Information and Control **27**(1) (1975) 87–99

[57] AlFardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the tls and dtls record protocols. (2013)

[58] Dan Bernstein and Kenny Paterson and Bertram Poettering and Jacob Schuldt: On the Security of RC4 in TLS (2013) `http://www.isg.rhul.ac.uk/tls/`.

[59] Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smard Cards. Springer (2007)

[60] Boneh, D., Demillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: Eurocrypt 1997. Volume 1233., LNCS (Springer-Verlag) (1997) 37–51

[61] Dolev, D., Dwork, C., Naor, M.: Non-malleable Cryptography. SIAM Journal on Computing 3 **2** (2000) 391–497

[62] Benaloh, J.: Dense Probabilistic Encryption. In: Proceedings of the Workshop on Selected Areas of Cryptography. (1994) 120–128

[63] James, M.: A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS#1 v2.0. In: CRYPTO. (2001)

[64] RSA Laboratories, RSA Security Inc., 20 Crosby Drive, Bedford, MA 01730 USA: RSAES-OAEP Encryption Scheme (2000)

[65] RSA Data Security, Redwood City, CA: PKCS#1: RSA Encryption Standard (2002) Version 2.1.

[66] Strenzke, F.: Manger's Attack revisited. In: 12th International Conference on Information and Security on Information and Communications Security (ICICS 2010). Volume 6476., LNCS (2010)

[67] The Botan Library: `botan.randombit.net`.

[68] Pointcheval, D.: Chosen-chipertext security for any one-way cryptosystem. Proc. of PKC **1751** (2000) 129–146

[69] Overbeck, R.: An Analysis of Side Channels in the McEliece PKC (2008) available at `https://www.cosic.esat.kuleuven.be/nato_arw/slides_participants/Overbeck_slides_nato08.pdf` .

[70] Bleichenbacher, D.: Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1. In: CRYPTO, Springer-Verlag (1998) 1–12

[71] Hall, C., Goldberg, I., Schneier, B.: Reaction Attacks Against Several Public-Key Cryptosystems. In: Proc. of the 2nd International Conference on Information and Communications Security (ICICS'99). Volume 1726., LNCS (1999) 2–12

[72] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid: Special Publication 800-57 July 2012 – Recommendation for Key Management, Part 1: General (Revision 3) (2012) `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`.

A. Appendix

A.1. Cubic Equations involving less than four Basis Elements are impossible

In this section, we show a feature about the key-aimed timing attacks introduced in Section 4.2.1.6: It is not possible to have an equation of the form (4.10) where the ϵ_i contain only three basis elements and fulfil all the conditions about the ϵ_i given in Section 4.2.1.6. Specifically, in the following, we try to build an equation for C_1 of the form $\beta_{s_1}(\beta_{g_1}, \beta_{g_2})$ and show that it is not possible to satisfy all these conditions with the eight different ϵ_i that can be created from three basis elements:

- β_{s_1} appears exactly in ϵ_1 and ϵ_2. For them to be different, we need to add at least one further basis element to one of them.
- Since β_{s_1} may not appear in any of the other four ϵ_i, they must be built from the combinations of β_{g_1} and β_{g_2}. There are four such combinations so that we can build the remaining four different ϵ_i.
- But now the condition $\sum_{i=1}^{6} \epsilon_i = 0$ is violated, since the count of β_{g_1} across all ϵ_i is odd. From Table A.1, we see that (from an arbitrarily chosen point of view) the β_{g_1} that is contained in ϵ_2 is unmatched. Any attempt to create an even count of β_{g_1} violates another condition.

	β_{s_1}	β_{g_1}	β_{g_2}
ϵ_1	x		
ϵ_2	x	X	
ϵ_3		x	
ϵ_4			x
ϵ_5		x	x
ϵ_6			

Table A.1.: Example of the attempt to build a cubic equation using only three basis elements. An "x" mark denotes that the basis element is contained in the respective ϵ_i.

Akademischer Lebenslauf

1997 Erlangung der Allgemeinen Hochschulreife am Friederich-Dessauer-Gymnasium in Aschaffenburg

1997 Beginn des Physikstudiums an der Technischen Universität Darmstadt

2006 Abschluß des Physikstudiums

2006 Begin der nebenberuflichen Promotion

2013 Abschluß der Promotion

Acknowledgements

First of all, I thank Prof. Johannes Buchmann for the possibility to write this doctoral thesis at his institute and under his guidance. I also thank him for the funding of the participation in workshops, which could not be taken for granted for an external Ph.D. student.

Next, I thank Markus Ruppert and Erwin Stallenberger, the CEOs of FlexSecure GmbH, my employer, for the support I received from them: the possibility to work on interesting research projects and the funding of my attendances to scientific conferences. Furthermore, they were so helpful as to giving me the possibility to put my focus on the completion of the thesis by reducing my working time in the last year.

Also, I thank my co-authors Gregor Molter, Raphael Overbeck, Abdoulhadi Shoufan, Marc Stöttinger and Erik Tews for our productive cooperation. I am especially indebted to Raphael Overbeck for the introduction into code-based cryptography and his patient help with my very first conference paper. Furthermore, I thank Prof. Johannes Buchmann and Vangelis Karatsiolis for the proof-reading of this thesis.

Erklärung

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, November 2013

(Falko Strenzke)