Seitenkanalangriffe auf einen Post-Quanten-Kryptographie-Algorithmus

9. April 2019

Inhaltsverzeichnis

Abkürzungsverzeichnis

BSI	Bundesamt für Sicherheit in der Informationstechnik
CCNT	Cycle Counter Register
DDD	Data Display Debugger
ECC	Error Correction Code
gdb	GNU Debugger
LRPC	Low Rank Parity Check
NIST	National Institute of Standards and Technology
PMNC	Performance Monitor Control Register
TSC	Time Stamp Counter

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einleitung

Die vorliegende Arbeit beschäftigt sich mit einem Implementierungsangriff auf den Post-Quanten-Kryptografie-Algorithmus *McNie*.

1.1 Problemstellung

Post-Quanten-Kryptographie. Zur Sicherung der Vertraulichkeit und Integrität von Kommunikationskanälen werden in vielen informationstechnischen Protokollen und Anwendungen asymmetrische Kryptosysteme eingesetzt. Ein prominenter Vertreter dieser Art ist das RSA-Kryptosystem, das auf dem Faktorisierungsproblem von Ganzzahlen basiert (vgl. Menezes, van Oorschot und Vanstone 2001, S. 89). Es wird heute angenommen, dass dieses Problem mit klassischen Rechnern für hinreichend große Zahlen nicht lösbar ist: der Rechenaufwand der effizientesten klassischen Faktorisierungsalgorithmen verhält sich subexponentiell zur Länge der zu faktorisierenden Zahl in Bit (vgl. Homeister 2015, S. 194).

Der 1994 von Peter Shor publizierte Shor-Algorithmus ist in der Lage, eine Ganzzahl in polynomieller Zeit zu faktorisieren (vgl. ebd., S. 231). Somit eignet er sich dafür, RSA-Kryptogramme in überschaubarer Zeit anzugreifen. Dieser Algorithmus setzt jedoch die Existenz eines Quantencomputers voraus; nur auf einer solchen Plattform kann der Algorithmus ausgeführt werden.

Zum gegenwärtigen Zeitpunkt existiert kein Quantencomputer mit ausreichenden Rechenkapazitäten, um ein RSA-Kryptogramm von gängiger Schlüssellänge mit dem Shor-Algorithmus zu brechen. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) hält die Entwicklung eines Quantencomputers, der einen 2048 Bit RSA-Schlüssel in einigen hundert Stunden brechen kann, für theoretisch möglich, wenn eine Industrienation sehr großen Aufwand in Forschung und Entwicklung in diesem Feld investieren würde (vgl. Wilhelm u. a. 2018, S. 190).

Um auch nach einer eventuellen Entwicklung eines starken Quantencomputers auf sichere Kryptoalgorithmen zurückgreifen zu können, existiert das Forschungsgebiet der Post-Quanten-Kryptographie. Dieses befasst sich mit der Entwicklung alternativer Algorithmen, die Angriffen durch Quantenalgorithmen wie den Shor- oder Grover-Algorithmus widerstehen. Die US-amerikanische Standardisierungsinstitution *National Institute of Standards and Technology (NIST)* führt aktuell einen Standardisierungsprozess zu solchen Algorithmen durch. Die Einreichungen und die zugehörigen Referenzimplementierungen sind öffentlich einsehbar[1].

Implementierungsangriffe. Neben Angriffen auf Kryptosysteme, die auf mathematische Schwächen, das verwendete Protokoll oder den verwendeten Kommunikationskanal abzielen, muss auch die Sicherheit der Implementierungen betrachtet werden. Eine Implementierung kann geheime Nachrichten oder Schlüssel über Seitenkanäle wie die Stromaufnahme während Berechnungen offenlegen (vgl. Knežević, Rožić und Verbauwhede 2009, S. 69 ff.). Andere Seitenkanäle, die teilweise auch ohne physischen Zugriff ausgewertet werden können, sind das Zeitverhalten der Implementierung oder Änderungen in gemeinsam genutzten Speichern (vgl. z. B. Bernstein (2005), Kocher u. a. (2019) und Lipp u. a. (2018)).

[1] `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`

1.2 Zielsetzung

Ziel des Projekts ist die Untersuchung einer der Referenzimplementierungen, die im Rahmen des NIST-Standardisierungsprozesses eingereicht wurden, auf angreifbare Varianzen des Zeitverhaltens oder Angreifbarkeit über mikroarchitekturelle Seitenkanäle (Spectre).

Für diese Untersuchungen wurde die Implementierung von *McNie* ausgewählt. Ein einfacher Vorabtest offenbarte unterschiedliche Zeitdauer der Entschlüsselungsoperation in Abhängigkeit der verarbeiteten Daten. Die Histogramme in Abbildung 1.1 zeigen die Ergebnisse und wurden nach dem folgenden Verfahren erstellt:

1. Verwende ein konstantes Schlüsselpaar
2. Generiere eine zufällige Nachricht
3. Verschlüssle die generierte Nachricht
4. Miss 5000 Mal die Dauer der Entschlüsselungsoperation
5. Plotte aus den 5000 aufgezeichneten Zeiten ein Histogramm

Die Berechnungen wurden auf einem Intel Core i5-4300M durchgeführt. Der auf der NIST-Website bereitgestellte Source Code wurde zu diesem Zweck um Instruktionen zur Messung der Ausführungszeit der Entschlüsselungsoperation ergänzt.

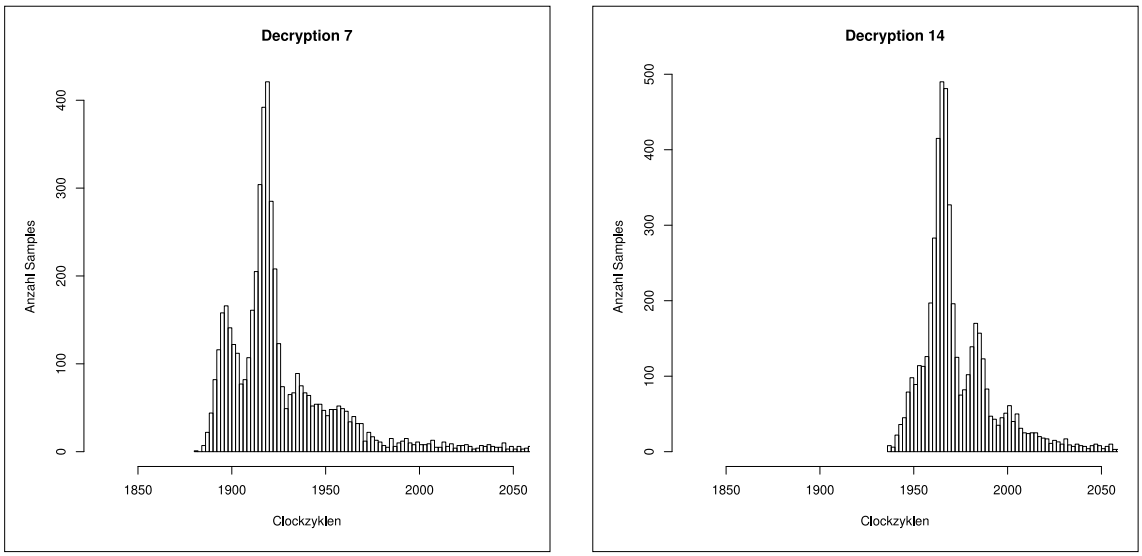


Abbildung 1.1: Zeitdauer der Entschlüsselung zweier unterschiedlicher Ciphertexte mit demselben Schlüssel. Jedes Histogramm zeigt die Verteilung der Zeitdauer bei 5000 identischen Berechnungen.

Ziel der Untersuchungen im Rahmen des Projekts ist es, zu überprüfen, ob aus der Zeitdauer und möglicherweise mithilfe von Informationen aus anderen Seitenkanälen auf Anteile des verwendeten kryptografischen Schlüssels geschlossen werden kann.

1.3 Aufbau der Arbeit

Das 2. Kapitel beschreibt die mathematischen Grundlagen des McNie-Kryptosystems und gibt einen Überblick über die Funktionsweise von Timing-Angriffen auf kryptografische Implementierungen. In Kapitel 3 ist eine entdeckte Schwachstelle in der McNie-Implementierung und ein mögliches Angriffsszenario dargestellt.

2 Grundlagen

Dieses Kapitel beschäftigt sich mit den theoretischen Grundlagen des Projektthemas. Im ersten Abschnitt wird das McNie-Kryptosystem beschrieben. Abschnitt 2.2 gibt eine Einführung in Timing-Angriffe auf kryptografische Implementierungen.

2.1 McNie-Kryptosystem

Das McNie-Kryptosystem wurde von Wissenschaftlern der Sogang Univerity in Seoul und der Chosun University in Gwangju, Südkorea entwickelt. Es ist in Galvez u. a. (2017) dokumentiert und trägt den Untertitel „Compact McEliece-Niederreiter Cryptosystem“. Daraus geht auch die Grundlage für McNie hervor: es basiert auf den Kryptosystemen McEliece und Niederreiter, die schon vor über 30 Jahren in der Literatur beschrieben wurden (McEliece 1978; Niederreiter 1986).

Wie auch Niederreiter und McEliece ist McNie ein codebasiertes Verfahren. Es greift auf eine Klasse von Error Correction Codes (ECCs) zurück, die als LRPC-Codes bezeichnet werden. Diese wurden in Gaborit u. a. (2013) erstmalig beschrieben.

In Abschnitt 2.1.1 werden der LRPC-Code und dessen Dekodierungsalgorithmus eingeführt. Abschnitt 2.1.2 gibt einen Überblick über das generelle Vorgehen bei der Schlüsselgenerierung, Verschlüsselung und Entschlüsselung mit McNie. Zuletzt konkretisiert Abschnitt 2.1.3 die allgemeinen Berechnungsvorschriften des McNie-Kryptosystems für den LRPC-Code.

2.1.1 LRPC-Codes

Der LRPC-Code ist ein linearer Blockcode, er arbeitet also mit Codeworten einer festen Blocklänge n über einem Untervektorraum von $\mathbb{F}_{q^m}$ (vgl. Lütkebohmert 2003, S. 13 f.).

Dieser Abschnitt liefert zunächst einige Grundlagen zum LRPC-Code, zum Distanzmaß der Rang-Metrik und zu quasi-zyklischen Codes. Der Rest des Abschnitts ist dem Dekodierungsalgorithmus Φ_H gewidmet.

Parameter

Ein LRPC-Code hat die folgenden charakteristischen Parameter (vgl. Galvez u. a. 2017, S. 6):

- n: Blöcklänge (Länge der Codewörter)
- k: Länge der Nachrichtenwörter und Dimension des Vektorraums, aus dem die Codewörter stammen
- d: Rang des Codes
- blk (Blockmatrix-Größe) und $l > n - k$ beeinflussen die Dimensionen der zyklischen Matrizen.

Der Code verfügt über eine Generatormatrix G und eine Kontrollmatrix $H = (h_{ij})$. Die Elemente der Kontrollmatrix h_{ij} stammen aus einem Unterraum F von $\mathbb{F}_{q^m}$. Die Dimension dieses Unterraums ist höchstens d. Die Basis von F besteht somit aus höchstens d Basisvektoren F_i: $\{F_1, F_2, \ldots, F_d\}$.

Rang-Metrik

Der LRPC-Code verwendet statt der oft gebräuchlichen Hamming-Metrik die Rang-Metrik als Distanzmaß (vgl. Gabidulin 1985, S. 1; vgl. Gaborit u. a. 2013, S. 169 f.).

Jedes Element aus $\mathbb{F}_{q^m}$ lässt sich als Vektor über $\mathbb{F}_q$ darstellen. Sei $\beta = \{\beta_1, \ldots, \beta_m\}$ eine Basis von $\mathbb{F}_{q^m}$ über $\mathbb{F}_q$, also des Vektorraums, der insofern mit $\mathbb{F}_{q^m}$ korrespondiert, dass jedes Element $v_i \in \mathbb{F}_{q^m}$ als eine Linearkombination der Basisvektoren dieses Vektorraums darstellbar ist: $v_i = a_{1i}\beta_1 + a_{2i}\beta_2 + \cdots + a_{mi}\beta_m$. Die $a_{ji} \in \mathbb{F}_q$ heißen Koordinaten von v_i.

Sei V_n ein n-dimensionaler Vektorraum über $\mathbb{F}_{q^m}$. Jedem Vektor $v = (v_1, v_2, \ldots, v_n)$ in V_n, $v_i \in \mathbb{F}_{q^m}$, lässt sich dann eine Matrix $\bar{v}(x)$ der folgenden Form zuordnen:

$$\bar{v}(x) = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Der Rang eines Vektors v, $\operatorname{rang}(v)$, ist definiert als der Rang der Matrix $\bar{v}(x)$, also der maximalen Anzahl der Koordinatenvektoren seiner Komponenten bezüglich der Basis β, die linear unabhängig voneinander sind.

Auf Basis dieser Definition lässt sich die Rang-Metrik oder Rang-Distanz definieren:

$$d(x, y) = \operatorname{rang}(x - y), \quad x, y \in V_n.$$

Quasi-zyklische Codes

McNie benutzt quasi-zyklische LRPC-Codes. Die Kontroll- und Generatormatrizen solcher Codes haben eine besondere Form und müssen daher nicht vollständig gespeichert werden, sondern sind aus einigen Vektoren geringer Länge vollständig rekonstruierbar.

Ein quasi-zyklischer Code hat die Eigenschaft, dass eine zyklische Verschiebung eines Codeworts um n_0 Stellen wieder ein Codewort ergibt. Zyklische Codes erfüllen diese Eigenschaft bei Verschiebungen um beliebig viele Stellen (vgl. Lütkebohmert 2003, S. 47).

Ein quasi-zyklischer $[n, k]$-Code ist wie folgt definiert (vgl. Chen, Peterson und Weldon 1969, S. 408 f.; vgl. Galvez u. a. 2017, S. 6): Sei $n = blk \cdot n_0$ und $k = blk \cdot k_0$. Der Code hat eine Generatormatrix G der Form

$$G = \begin{pmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,n_0} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,n_0} \\ \vdots & \vdots & \ddots & \vdots \\ C_{k_0,1} & C_{k_0,2} & \cdots & C_{k_0,n_0} \end{pmatrix},$$

wobei C_{ij} zyklische $blk \times blk$-Matrizen der folgenden Form sind:

$$C = \begin{pmatrix} c_0 & c_1 & \cdots & c_{blk-1} \\ c_{blk-1} & c_0 & \cdots & c_{blk-2} \\ c_{blk-2} & c_{blk-1} & \cdots & c_{blk-3} \\ \vdots & \vdots & \ddots & \vdots \\ c_1 & c_2 & \cdots & c_0 \end{pmatrix}. \tag{2.1}$$

Eine zyklische Matrix C_{ij} ist durch ihre erste Zeile eindeutig definiert. Daher lässt sie sich auch als Polynom ausdrücken:

$$c(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{blk-1} x^{blk-1}$$

Die Summen und Produkte zweier solcher Polynome definieren ebenfalls wieder eine zyklische Matrix.

Dekodierungsalgorithmus Φ_H

Der Dekodierungsalgorithmus Φ_H, der in Algorithmus 2.1 angegeben ist, rekonstruiert aus einem potentiell fehlerbehafteten Codewort $y = mG + e$ mit höchstens r Fehlern das ursprüngliche Nachrichtenwort m. Dazu greift er auf die Matrizen G und H und optional auf eine vorberechnete Dekodierungsmatrix D_H zurück.

Input: H, y, d
Output: e, m

1 $s \leftarrow H \cdot y^T$
2 $S \leftarrow \langle s_1, \ldots, s_{n-k} \rangle$
3 **for** $i := 1$ **to** d **do**
4 $\quad S_i \leftarrow F_i^{-1} S$
5 **end for**
6 $E \leftarrow S_1 \cap S_2 \cap \cdots \cap S_d$
7 $\{E_1, E_2, \ldots, E_r\} \leftarrow \textbf{basis}(E)$
8 $s' \leftarrow (s_{1,1,1}, \ldots, s_{1,1,r}, \ldots, s_{(n-k),d,r})$
9 $e' \leftarrow \textbf{Resolve}(A_H^r, s')$
10 $(e_{1,1}, e_{1,2}, \ldots, e_{n,r}) \leftarrow e'$
11 **for** $i := 1$ **to** n **do**
12 $\quad e_i \leftarrow \sum_{j=1}^{r} e_{i,j} E_j$
13 **end for**
14 $m \leftarrow \textbf{Resolve}(G, y - e)$

Algorithmus 2.1: Dekodierungsalgorithmus Φ_H (nach: Yazbek u. a. 2017, S. 5)

Der Dekodierungsalgorithmus nimmt die folgenden Berechnungen vor (vgl. Gaborit u. a. 2013, S. 172 ff.; vgl. Yazbek u. a. 2017, S. 4 f.):

1. Syndrom und Syndromraum berechnen (Z. 1–2). Berechne den Syndrom-Vektor $s = (s_1, \ldots, s_{n-k}) = H \cdot y^T$. Berechne außerdem den Unterraum von $\mathbb{F}_{q^m}$, in dem die $s_i \in \mathbb{F}_{q^m}$ liegen (Syndromraum): $S = \langle s_1, \ldots, s_{n-k} \rangle$.

2. Wiederherstellen des Fehler-Vektorraums E (Z. 3–7). Definiere d Unterräume von S, die mit $S_i = F_i^{-1} S$ bezeichnet werden. Die Generatoren des Unterraums S_i sind die Generatoren von S multipliziert mit der Inversen des i-ten Basisvektors von F in $\mathbb{F}_{q^m}$. Die Generatoren von S_1 und S_2 sind exemplarisch in (2.2) dargestellt.

$$\begin{aligned}
S_1 &= \langle F_1 E_1 F_1^{-1}, \quad F_1 E_2 F_1^{-1}, \quad \ldots, F_1 E_r F_1^{-1}, \quad F_2 E_1 F_1^{-1}, \quad \ldots, F_2 E_r F_1^{-1}, \quad \ldots, F_d E_r F_1^{-1} \rangle \\
&= \langle E_1, \quad E_2, \quad \ldots, E_r, \quad F_2 E_1 F_1^{-1}, \quad \ldots, F_2 E_r F_1^{-1}, \quad \ldots, F_d E_r F_1^{-1} \rangle \\
S_2 &= \langle F_1 E_1 F_2^{-1}, \quad F_1 E_2 F_2^{-1}, \quad \ldots, F_1 E_r F_2^{-1}, \quad F_2 E_1 F_2^{-1}, \quad \ldots, F_2 E_r F_2^{-1}, \quad \ldots, F_d E_r F_2^{-1} \rangle \\
&= \langle F_1 E_1 F_2^{-1}, \quad F_1 E_2 F_2^{-1}, \quad \ldots, F_1 E_r F_2^{-1}, \quad E_1, \quad \ldots, E_r, \quad \ldots, F_d E_r F_2^{-1} \rangle
\end{aligned} \tag{2.2}$$

Die Schnittmenge der Vektorräume S_i bildet, wie (2.2) nahelegt, den gesuchten Vektorraum E: $E = S_1 \cap S_2 \cap \cdots \cap S_d$. Sei $\{E_1, E_2, \ldots, E_r\}$ eine Basis von E.

3. Wiederherstellen des Fehlervektors e (Z. 8–13). Der Dekodierungsalgorithmus Φ_H bestimmt den Vektor e nicht direkt, sondern berechnet den Koordinatenvektor von e bezüglich der Basis von E. Aus diesem Koordinatenvektor kann e rekonstruiert werden. Die Grundidee ist es, die Gleichungen in $\mathbb{F}_{q^m}$ in Gleichungen über $\mathbb{F}_q$ zu transformieren. Die folgende Auflistung gibt einen Überblick über den Ablauf dieses Schritts:

- Stelle die Elemente der Matrix $H = (h_{i,j})$ bezüglich der Basis des Vektorraums F dar.

- Stelle die Komponenten des Fehlervektors $e = (e_1, \ldots, e_n)$ bezüglich der Basis von E dar.

- Stelle die Komponenten des Syndromvektors $s = (s_1, \ldots, s_{n-k})$ bezüglich der Basis des Produktraums $P = \langle E \cdot F \rangle$ dar.
- Löse das Gleichungssystem $H \cdot e^T = s$ in den oben angegebenen Vektorräumen.
- Transformiere das Ergebnis zurück in den Vektorraum über $\mathbb{F}_{q^m}$ und erhalte so den Fehlervektor e.

Um den Rechenaufwand beim Lösen des Gleichungssystems zu verringern, kann eine Dekodierungsmatrix D_H vorberechnet werden, die nur von der Kontrollmatrix H und der Dimension des Fehlervektorraums r abhängt. Dann ist nur noch eine Matrix-Vektor-Multiplikation zum Lösen des Gleichungssystem mit beliebigen Fehler- bzw. Syndromvektoren erforderlich.

Transformation in $\mathbb{F}_q$. Der Fehlervektor $e = (e_1, \ldots, e_n)$ mit Gewicht r liege in einem r-dimensionalen Vektorraum E mit der Basis $\{E_1, \ldots, E_r\}$. Dann können alle e_i als Linearkombination der Basisvektoren geschrieben werden: $e_i = \sum_{j=1}^{r} e_{ij} E_j$, $e_i \in \mathbb{F}_q$.

Die Matrix $H = (h_{ij})$ ist so konstruiert, dass die h_{ij} aus einem d-dimensionalen Vektorraum F mit der Basis $\{F_1, \ldots, F_d\}$ stammen. Also kann jedes h_{ij} als Linearkombination der Basisvektoren von F dargestellt werden: $h_{ij} = \sum_{l=1}^{d} h_{ijl} F_l$, $h_{ijl} \in \mathbb{F}_q$.

Ein weiterer relevanter Vektorraum ist der Produktraum von E und F, der hier mit $\langle E \cdot F \rangle$ bezeichnet wird und die Basis $\{F_1E_1, F_1E_2, \ldots, F_1E_r, F_2E_1, \ldots, F_2E_r, \ldots, F_dE_r\}$ hat. Der Produktraum habe die maximale Dimension rd, d. h. die Produkte der Basisvektoren von E und F sind alle linear unabhängig voneinander.

Dann können die Syndromgleichungen $H \cdot e^T = s$ über $\mathbb{F}_{q^m}$ in Gleichungen über $\mathbb{F}_q$ transformiert werden. Die Syndromgleichungen über $\mathbb{F}_{q^m}$ ergeben sich aus dem folgenden Gleichungssystem:

$$H \cdot e^T = \begin{pmatrix} h_{1,1} & h_{1,2} & \ldots & h_{1,n} \\ h_{2,1} & h_{2,2} & \ldots & h_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-k,1} & h_{n-k,2} & \ldots & h_{n-k,n} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{n-k} \end{pmatrix}$$

Durch Multiplikation ergeben sich die Gleichungen über $\mathbb{F}_{q^m}$, die zu lösen sind:

$$\begin{cases} h_{1,1}e_1 + h_{1,2}e_2 & + \ldots + h_{1,n}e_n & = s_1 \\ h_{2,1}e_1 + h_{2,2}e_2 & + \ldots + h_{2,n}e_n & = s_2 \\ \vdots & & \vdots \\ h_{n-k,1}e_1 + h_{n-k,2}e_2 & + \ldots + h_{n-k,n}e_n & = s_{n-k} \end{cases} \tag{2.3}$$

Um das Gleichungssystem in eines über $\mathbb{F}_q$ zu transformieren, schreiben wir zunächst die h_{ij} in der Basis von F, die e_i in der Basis von E und die s_i in der Basis von $\langle E \cdot F \rangle$. Das Produkt von $h_{ij} \cdot e_j$ hat dann die in (2.4) angegebene Form. Die Operation $\otimes$ bezeichnet das dyadische Produkt. Das Ergebnis ist eine $d \times r$-Matrix, die die Koordinaten des Produkts

zur Basis des Produktraums $\langle E \cdot F \rangle$ angibt.

$$h_{ij} \cdot e_j = \begin{pmatrix} h_{ij1} \\ h_{ij2} \\ \vdots \\ h_{ijd} \end{pmatrix} \otimes \begin{pmatrix} e_{j1} & e_{j2} & \cdots & e_{jr} \end{pmatrix} = \begin{pmatrix} h_{ij1}e_{j1} & h_{ij1}e_{j2} & \dots & h_{ij1}e_{jr} \\ h_{ij2}e_{j1} & h_{ij2}e_{j2} & \dots & h_{ij2}e_{jr} \\ \vdots & \vdots & \ddots & \vdots \\ h_{ijd}e_{j1} & h_{ijd}e_{j2} & \dots & h_{ijd}e_{jr} \end{pmatrix} \tag{2.4}$$

Jede Komponente s_i des Syndromvektors $s = (s_1, \dots, s_{n-k})$ lässt sich ebenfalls im Produktraum $\langle E \cdot F \rangle$ darstellen. Die Koordinaten von s_i zur Basis des Produktraums können ebenfalls in einer $d \times r$-Matrix angegeben werden:

$$s_i = \begin{pmatrix} s_{i11} & s_{i12} & \dots & s_{i1r} \\ s_{i21} & s_{i22} & \dots & s_{i2r} \\ \vdots & \vdots & \ddots & \vdots \\ s_{id1} & s_{id2} & \dots & s_{idr} \end{pmatrix} \tag{2.5}$$

Hinweis: Es ist $s_i \neq h_{ij} \cdot e_j$.

Mit den Vorüberlegungen aus (2.4) und (2.5) ergibt sich nun – durch Addition der $h_{ij} \cdot e_j$ gemäß den Gleichungen aus (2.3) – das folgende Gleichungssystem über $\mathbb{F}_q$:

$$\begin{pmatrix} \sum_{j=1}^{n} h_{1,j,1}e_{j,1} & \sum_{j=1}^{n} h_{1,j,1}e_{j,2} & \dots & \sum_{j=1}^{n} h_{1,j,1}e_{j,r} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=1}^{n} h_{1,j,d}e_{j,1} & \sum_{j=1}^{n} h_{1,j,d}e_{j,2} & \dots & \sum_{j=1}^{n} h_{1,j,d}e_{j,r} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=1}^{n} h_{(n-k),j,1}e_{j,1} & \sum_{j=1}^{n} h_{(n-k),j,1}e_{j,2} & \dots & \sum_{j=1}^{n} h_{(n-k),j,1}e_{j,r} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=1}^{n} h_{(n-k),j,d}e_{j,1} & \sum_{j=1}^{n} h_{(n-k),j,d}e_{j,2} & \dots & \sum_{j=1}^{n} h_{(n-k),j,d}e_{j,r} \end{pmatrix}$$

$$= \begin{pmatrix} s_{1,1,1} & s_{1,1,2} & \dots & s_{1,1,r} \\ \vdots & \vdots & \ddots & \vdots \\ s_{1,d,1} & s_{1,d,2} & \dots & s_{1,d,r} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ s_{(n-k),1,1} & s_{(n-k),1,2} & \dots & s_{(n-k),1,r} \\ \vdots & \vdots & \ddots & \vdots \\ s_{(n-k),d,1} & s_{(n-k),d,2} & \dots & s_{(n-k),d,r} \end{pmatrix}$$

Daraus ergibt sich ein neues Gleichungssystem:

$$\begin{cases} \sum_{j=1}^{n} h_{1,j,1} e_{j,1} = s_{1,1,1} \\ \vdots \\ \sum_{j=1}^{n} h_{1,j,1} e_{j,r} = s_{1,1,r} \\ \vdots \\ \sum_{j=1}^{n} h_{1,j,d} e_{j,1} = s_{1,d,1} \\ \vdots \\ \sum_{j=1}^{n} h_{1,j,d} e_{j,r} = s_{1,d,r} \\ \vdots \\ \vdots \\ \sum_{j=1}^{n} h_{(n-k),j,1} e_{j,1} = s_{(n-k),1,1} \\ \vdots \\ \sum_{j=1}^{n} h_{(n-k),j,1} e_{j,r} = s_{(n-k),1,r} \\ \vdots \\ \sum_{j=1}^{n} h_{(n-k),j,d} e_{j,1} = s_{(n-k),d,1} \\ \vdots \\ \sum_{j=1}^{n} h_{(n-k),j,d} e_{j,r} = s_{(n-k),d,r} \end{cases}$$

Generieren der Dekodierungsmatrix D_H. Um dieses Gleichungssystem einfach lösen zu können, sieht der Dekodierungsalgorithmus folgende Schritte vor:

Zunächst wird die $(n-k)rd \times nr$-Matrix A_H^r aufgestellt, die sich als Blockmatrix von $r \times r$-Matrizen $H_{i,j,v}$ zusammensetzt:

$$\begin{pmatrix} H_{1,1,1} & H_{1,2,1} & \cdots & H_{1,n,1} \\ \vdots & \vdots & \ddots & \vdots \\ H_{1,1,d} & H_{1,2,d} & \cdots & H_{1,n,d} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ H_{(n-k),1,1} & H_{(n-k),2,1} & \cdots & H_{(n-k),n,1} \\ \vdots & \vdots & \ddots & \vdots \\ H_{(n-k),1,d} & H_{(n-k),2,d} & \cdots & H_{(n-k),n,d} \end{pmatrix} \text{ mit } H_{i,j,v} = \begin{pmatrix} h_{i,j,v} & 0 & \cdots & 0 \\ 0 & h_{i,j,v} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & h_{i,j,v} \end{pmatrix}$$

Es gilt dann $A_H^r \cdot e'^T = s'$, mit:

- $e' = (e_{1,1}, e_{1,2}, \ldots, e_{1,r}, \ldots, e_{n,r})$
- $s' = (s_{1,1,1}, \ldots, s_{1,1,r}, \ldots, s_{(n-k),d,r})$

A_H^r hängt nur von H und r ab, nicht jedoch von den Basen von F, E oder $\langle E \cdot F \rangle$.

Aus der Matrix A_H^r wird eine invertierbare $nr \times nr$-Matrix A_H extrahiert. Die Inverse $A_H^{-1} =: D_H$ wird Dekodierungsmatrix genannt.

Bestimmung des Fehlervektors und Rücktransformation in $\mathbb{F}_{q^m}$. Die Bestimmung des Fehlervektors bezüglich der Basis von E (hier mit e' bezeichnet) erfordert nun nur noch eine Matrix-Vektor-Multiplikation: $D_H \cdot s' = e'$.

Durch Rücktransformation von e' in den $\mathbb{F}_{q^m}$ lässt sich so der tatsächliche Fehlervektor e bestimmen.

4. Wiederherstellen der Nachricht m (Z. 14). Es gilt die Beziehung $y = mG + e$. Durch Lösen des linearen Gleichungssystems $mG = y - e$ lässt sich m bestimmen.

2.1.2 McNie: Genereller Ablauf der Schlüsselgenerierung, Ver- und Entschlüsselung

Ähnlich wie McEliece generiert auch McNie zufällige Matrizen, die die Kontrollmatrix des Codes verschleiern. Nur die Kenntnis dieser Matrizen erlaubt es dem Empfänger einer verschlüsselten Nachricht, den Klartext zu rekonstruieren.

Schlüsselgenerierung

Vorab sind die folgenden Parameter festzulegen. Empfehlungen für sinnvolle Größen sind in Galvez u. a. (2017, S. 14) angegeben.

- Die Parameter n, k, d, r, blk und l des verwendeten Codes, der r Fehler korrigieren kann
- Der endliche Körper $\mathbb{F}_{q^m}$, auf dem der Code operiert. q ist eine Primzahl, $m \in \mathbb{N}$.

Ein Teilnehmer generiert sein Schlüsselpaar wie folgt (vgl. ebd., S. 4):

- Erzeuge zufällig die $(n-k) \times n$-Kontrollmatrix H über $\mathbb{F}_{q^m}$.
- Erzeuge eine invertierbare $(n-k) \times (n-k)$-Matrix S über $\mathbb{F}_{q^m}$ und eine $n \times n$-Permutationsmatrix P zum Verschleiern der Generatormatrix.
- Erzeuge zufällig die $l \times n$-Generatormatrix G' über $\mathbb{F}_{q^m}$.
- Berechne das Matrixprodukt $F = G'P^{-1}H^TS$. F ist eine $l \times (n-k)$-Matrix.

- Der geheime Schlüssel setzt sich zusammen als $K_{\text{priv}} = (H, S, P)$.
- Der öffentliche Schlüssel setzt sich zusammen als $K_{\text{pub}} = (G', F)$.

Verschlüsselung

Bob möchte eine Nachricht, kodiert als Vektor m der Länge l über $\mathbb{F}_{q^m}$, an Alice übermitteln. Alice' Public Key $K_{\text{pub}} = (G', F)$ ist ihm bekannt. Folgende Berechnungsschritte sind erforderlich (vgl. ebd.):

- Bob generiert einen zufälligen Fehlervektor e der Länge n über $\mathbb{F}_{q^m}$ mit einem Gewicht von maximal r.
- Bob berechnet den Vektor $c_1 = mG' + e$ der Länge n.
- Bob berechnet den Vektor $c_2 = mF$ der Länge $n-k$.
- Bob überträgt den Chiffratvektor $c = (c_1, c_2)$ der Länge $2n-k$ an Alice.

Entschlüsselung

Alice empfängt den Vektor $c = (c_1, c_2)$ und entschlüsselt ihn mithilfe ihres privaten Schlüssels $K_{\text{priv}} = (H, P, S)$ wie folgt (vgl. ebd., S. 5):

- Alice berechnet das (maskierte) Syndrom s' :

$$
\begin{aligned}
s' &= c_1P^{-1}H^T && - c_2S^{-1} && \mid c_1 = mG' + e,\ c_2 = mF \\
&= (mG' + e)P^{-1}H^T && - (mF)S^{-1} && \mid F = G'P^{-1}H^TS \\
&= (mG' + e)P^{-1}H^T && - (m(G'P^{-1}H^TS))S^{-1} && \\
&= mG'P^{-1}H^T + eP^{-1}H^T && - mG'P^{-1}H^T && \\
&= eP^{-1}H^T &&
\end{aligned}
$$

  Da es sich bei dem verwendeten Code um einen linearen Code handelt, ist das Produkt von fehlerbehaftetem Wort und Kontrollmatrix mit dem Produkt von Fehlervektor und Kontrollmatrix identisch (vgl. Witt 2013, S. 92 f.).

- Alice decodiert das Syndrom s' mit dem Dekodierungsalgorithmus Φ_H des Codes und erhält als Ergebnis den maskierten Fehlervektor $e' = eP^{-1}$.
- Alice hebt die Maskierung von e' auf, indem sie mit P multipliziert: $e'P = eP^{-1}P = e$.
- Alice berechnet m, indem sie das Gleichungssystem $mG' = c_1 - e$ löst.
  Einfacher ist die Berechnung der Nachricht, wenn G' in der Normalform vorliegt. Als Normalform wird die Gestalt $G' = \left[\ I_l \mid R_{n-l}\ \right]$ bezeichnet, wobei I_l die $l \times l$-Einheitsmatrix und R_{n-l} die Restmatrix bezeichnet (vgl. Witt 2013, S. 88). Dann geben bereits die ersten l Komponenten des Vektors $c_1 - e$ die Nachricht m an.

2.1.3 McNie mit LRPC-Codes

Für McNie werden 3-quasi-zyklische ($n_0 = 3$) oder 4-quasi-zyklische ($n_0 = 4$) Codes verwendet, da sich für diese Parameter die kürzesten Schlüssellängen ergeben. Diese zeichnen sich durch die folgende Parameterwahl aus (vgl. Galvez u. a. 2017, S. 6):

- 3-quasi-zyklisch
  - n: Vielfaches von 3
  - $k = l = \frac{2n}{3}$
    $\Rightarrow n_0 = 3, k_0 = 2$
  - Öffentlicher Schlüssel:
    - G' ist eine $\frac{2n}{3} \times n$-Matrix
    - F ist eine $\frac{2n}{3} \times \frac{n}{3}$-Matrix
- 4-quasi-zyklisch
  - n: Vielfaches von 4
  - $k = \frac{n}{2}$
    $\Rightarrow n_0 = 4, k_0 = 3$
  - $l = \frac{3n}{4}$
  - Öffentlicher Schlüssel:
    - G' ist eine $\frac{3n}{4} \times n$-Matrix
    - F ist eine $\frac{3n}{4} \times \frac{n}{2}$-Matrix

Die folgenden Abschnitte beschreiben die McNie-Berechnungsschritte für die Schlüsselgenerierung sowie die Ver- und Entschlüsselung bei der Verwendung von 3-quasi-zyklischen LRPC-Codes (vgl. ebd., S. 6 ff.).

Schlüsselgenerierung

Die Parameter für einen 3-quasi-zyklischen Code werden wie oben angegeben gewählt. Sei $blk = \frac{n}{3}$.

- Erzeuge zufällig einen d-dimensionalen Unterraum F von $\mathbb{F}_{q^m}$ über $\mathbb{F}_q$. Die Basis von F sei $\{F_1, \ldots, F_d\}$.
- Generiere drei Vektoren h_1, h_2 und h_3 der Länge $\frac{n}{3}$ mit zufälligen Elementen aus F.
- Konstruiere aus den Vektoren h_i drei zyklische $\frac{n}{3} \times \frac{n}{3}$-Matrizen H_i nach dem Schema der Gleichung 2.1.
- Die $\frac{n}{3} \times n$-Kontrollmatrix des Codes setzt sich zusammen als Blockmatrix der Matrizen H_i:
$$H = \begin{pmatrix} H_1 & H_2 & H_3 \end{pmatrix}$$

Um H zu speichern, müssten nun lediglich die Vektoren h_i gespeichert werden. Da im Dekodierungsalgorithmus jedoch ohnehin nicht die Matrix H, sondern die Darstellung der Elemente h_{ij} bezüglich der Basis von F relevant ist, ist es effizienter, nur diese Darstellungsform zu speichern. Statt der Matrix H werden daher die d Basisvektoren von F, sowie für jeden Vektor h_i die Koeffizienten $a_i \in \mathbb{F}_q$ aus der Darstellung $h_i = \sum_{i=1}^{d} F_i a_i$ gespeichert.

Die folgende Auflistung beschreibt die Generierung der Matrizen G', P und S.

- Generiere zwei Vektoren g_1, g_2 der Länge $\frac{n}{3}$ mit zufälligen Elementen aus $\mathbb{F}_{q^m}$.
- Konstruiere aus den Vektoren g_1 und g_2 die zwei zyklischen $\frac{n}{3} \times \frac{n}{3}$-Matrizen G_1 und G_2 nach dem Schema der Gleichung 2.1.

- Die $\frac{2n}{3} \times n$-Matrix G' ergibt sich als folgende Blockmatrix ($I_{\frac{n}{3}}$ bezeichnet die $\frac{n}{3} \times \frac{n}{3}$-Einheitsmatrix):
$$G' = \begin{pmatrix} I_{\frac{n}{3}} & 0 & G_1 \\ 0 & I_{\frac{n}{3}} & G_2 \end{pmatrix}$$
- Sei P die $n \times n$-Einheitsmatrix: $P = I_n$.
- Berechne die zyklische $\frac{n}{3} \times \frac{n}{3}$-Matrix $S = (H_1 + H_3 G_1^T)^{-1}$.
- Berechne die $\frac{2n}{3} \times \frac{n}{3}$-Matrix $F = G' P^{-1} H^T S$:
$$\begin{aligned} F &= G' P^{-1} H^T S \\ &= \begin{pmatrix} I_{\frac{n}{3}} & 0 & G_1 \\ 0 & I_{\frac{n}{3}} & G_2 \end{pmatrix} \cdot I_n \cdot \begin{pmatrix} H_1^T \\ H_2^T \\ H_3^T \end{pmatrix} \cdot S \\ &= \begin{pmatrix} H_1^T + G_1 H_3^T \\ H_2^T + G_2 H_3^T \end{pmatrix} \cdot (H_1 + H_3 G_1^T)^{-1} \\ &= \begin{pmatrix} I_{\frac{n}{3}} \\ (H_2^T + G_2 H_3^T)(H_1 + H_3 G_1^T)^{-1} \end{pmatrix} = \begin{pmatrix} I_{\frac{n}{3}} \\ F' \end{pmatrix} \end{aligned}$$
  Im letzten Schritt definieren wir $F' := (H_2^T + G_2 H_3^T)(H_1 + H_3 G_1^T)^{-1}$.
- Falls F nicht in Spaltenstufenform reduziert werden kann, beginne das Verfahren von vorn.
- Es ergeben sich die folgenden Schlüssel:
  - Privater Schlüssel $k_{\text{priv}} = (H, S)$
  - Öffentlicher Schlüssel $k_{\text{pub}} = (G', F)$

  Die Matrix P ist kein Schlüsselbestandteil, da sie auf die $n \times n$-Einheitsmatrix gesetzt ist.

Verschlüsselung

Gegeben sei ein Nachrichtenvektor $\bar{m}$ über $\mathbb{F}_{q^m}$. Die Länge von $\bar{m}$ wird in den 4-Byte-String a kodiert. Die Konkatenation der Nachricht und ihrer Länge bildet den zu kodierenden Nachrichtenvektor $m = (a||\bar{m})$.

Sei $s = \text{length}(m)$ die Länge des Vektors m. s darf die doppelte Blockgröße ($2 \cdot blk = l$) nicht überschreiten. Sei α die Anzahl der Bytes, die in zwei Blöcken transportiert werden kann. Wir betrachten im Folgenden nur die Fälle, in denen $s \leq \alpha$.

- Falls $s < \alpha$, fülle m vor der Übertragung mit Zufallsbits auf, bis die Länge α erreicht ist: Sei v ein $(\alpha - s)$-Byte-String aus zufällig gleichverteilten Bits. Definiere $x := (m||v) = ((a||\bar{m})||v)$.
- Falls $s = \alpha$, definiere $x := m = (a||\bar{m})$.
- Generiere einen zufälligen Fehlervektor e der Länge n über $\mathbb{F}_{q^m}$ mit einem Gewicht $\text{weight}(e) \leq r$.
- Berechne $c_1 = xG' + e$.
- Berechne $c_2 = xF$.
- Der Ciphertext ist $c = (c_1, c_2)$.

Entschlüsselung

Alice empfängt $c = (c_1, c_2)$ und entschlüsselt wie folgt:

- Berechne das maskierte Syndrom $s' = c_1 P^{-1} H^T - c_2 S^{-1}$.
- Durch Anwendung des Dekodierungsalgorithmus Φ_H erhalte $\hat{e} = eP^{-1}$.
- Multipliziere mit P, um den tatsächlichen Fehlervektor zu erhalten: $e = \hat{e}P$.

- Berechne x durch Lösen des Gleichungssystems $xG' = c_1 - e$. Einsetzen von c_1 zeigt die Korrektheit:

$$\begin{aligned} xG' &= c_1 - e \\ xG' &= (xG' + e) - e \\ xG' &= xG' \end{aligned}$$

  An dieser Stelle kann ggf. die in Abschnitt 2.1.2 beschriebene Vereinfachung angewandt werden, wenn G' bei der Verschlüsselung in Normalform vorlag.
- Teile x auf: die ersten 4 Byte geben die Länge a der tatsächlichen Nachricht an, der Rest entfällt auf die Nachricht $\bar{m}$ und ggf. Padding v. Stelle so die Nachricht $\bar{m}$ wieder her.

2.2 Timing-Angriffe auf kryptografische Implementierungen

Timing-Angriffe auf kryptografische Implementierungen nutzen aus, dass die Laufzeit einer Ver- oder Entschlüsselungsfunktion daten- oder schlüsselabhängig variiert. Diese Unterschiede im Zeitverhalten, die aufgrund von Optimierungen, Verzweigungen oder Caching auftreten, werden ausgewertet, um den verwendeten kryptografischen Schlüssel zu rekonstruieren (vgl. Kocher 1996, S. 104).

2.2.1 Timing-Angriff auf den Square-and-Multiply-Algorithmus

Kocher beschreibt diese Art der Angriffe am Beispiel des Square-and-Multiply-Algorithmus für die modulare Exponentiation (Algorithmus 2.2), der u. a. im RSA-Kryptosystem und dem Diffie-Hellman-Schlüsselaustausch eingesetzt werden kann. Dieser Algorithmus berechnet die modulare Exponentiation $y^x \bmod n$.

INPUT: y, x, n
OUTPUT: $R = y^x \bmod n$

```
1  w ← length(x)
2  s_0 ← 1
3  for k := 0 to w − 1 do
4     if bit k of x == 1 then
5        R_k ← (s_k · y) mod n                    {Multiply}
6     else
7        R_k ← s_k
8     end if
9     s_{k+1} ← R_k^2 mod n                       {Square}
10 end for
11 return R_{w−1}
```

Algorithmus 2.2: Square-and-Multiply-Algorithmus (nach: Kocher 1996, S. 105)

Angreifermodell. Dem Angreifer stehen zwei identische Implementierungen des Algorithmus zur Verfügung, eine Opfer-Implementierung und eine Test-Implementierung. Die Opfer-Implementierung enthält den geheimen Wert x, bspw. den geheimen Exponenten eines RSA-Schlüssels. Dieser Wert ist für den Angreifer nicht zugreifbar. Er kann jedoch die Basis y frei wählen. Der Modul n ist konstant und dem Angreifer bekannt. Im Falle einer RSA-Implementierung entsprechen diese Voraussetzungen einem Chosen-Ciphertext-Szenario. Die Test-Implementierung erlaubt es dem Angreifer, Berechnungen mit beliebigen Werten für die Parameter x, y und n durchzuführen. Zudem kann der Angreifer bei beiden

Implementierungen die Gesamtausführungszeit T des Square-and-Multiply-Algorithmus messen. Abbildung 2.1 visualisiert dieses Modell.

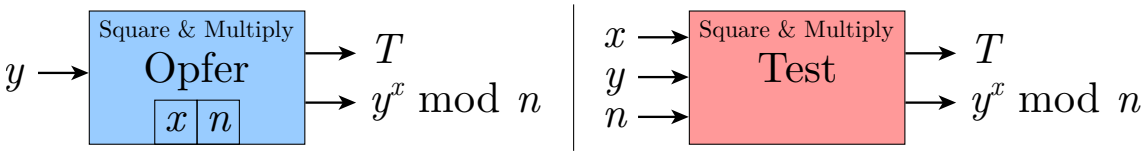


Abbildung 2.1: Angreifermodell

Definitionen. Sei w die Länge des Exponenten x in Bit. Die Zählung der Exponentenbits beginnt beim höchstwertigsten Bit, wie in Abbildung 2.2 dargestellt. x_b bezeichne die ersten (höchstwertigsten) b Bits des Exponenten, z. B.:

$$x = 1011\ 1001$$
$$x_1 = 1,\ x_2 = 10,\ x_3 = 101,\ \ldots,\ x_8 = 1011\ 1001 = x$$

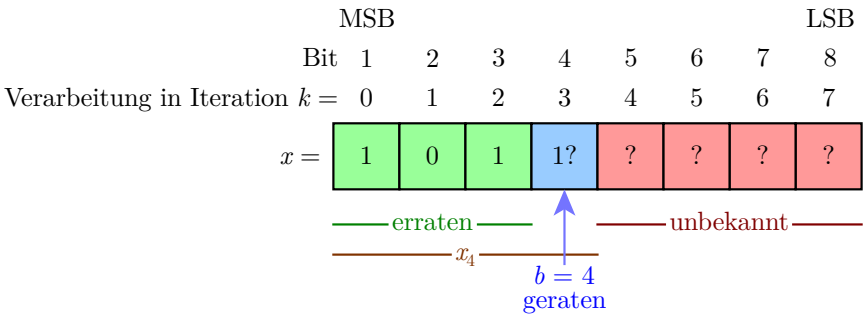


Abbildung 2.2: Bezeichnungen und iteratives Vorgehen beim Angriff auf den Square-and-Multiply-Algorithmus

Iteratives Vorgehen. Der Angreifer geht iterativ vor und bestimmt ein Exponentenbit nach dem anderen, beginnend mit dem höchstwertigsten. Kennt er die ersten b Exponentenbits (zu Beginn ist $b = 0$), bestimmt er das $b + 1$-te Exponentenbit wie folgt. Er berechnet mit der Test-Implementierung die ersten b Iterationen der For-Schleife; da er die ersten b Exponentenbits kennt, kann er den Anfang des Square-and-Multiply-Algorithmus problemlos und fehlerfrei berechnen, bis einschließlich des Wertes s_b, dem Zwischenergebnis am Ende der b-ten Iteration ($k = b - 1$).

Zur Berechnung der $b + 1$-ten Iteration ($k = b$) benötigt der Angreifer das nächste, ihm unbekannte Exponentenbit. Wenn es gesetzt ist, wird die Multiplikation $R_b \leftarrow (s_b \cdot y) \bmod n$ gerechnet (Zeile 5), andernfalls erfolgt lediglich die Zuweisung $R_b \leftarrow s_b$ (Zeile 7) (vgl. ebd., S. 105).

Prinzip des Angriffs in einem günstigen Szenario. Zur Verdeutlichung des Angriffsprinzips geht Kocher zunächst von äußerst günstigen Bedingungen aus: Die Berechnungszeit der modularen *Multiplikation* (Zeile 5) sei normalerweise sehr gering, in seltenen Fällen dauere sie jedoch deutlich länger als die gesamte modulare *Exponentiation* normalerweise dauert. Wenn der Angreifer weiß, für welche Werte s_b und y die modulare Multiplikation langsam ist, kann er daraus eine Hypothese für das nächste Exponentenbit ableiten:

- Wenn die gesamte Berechnungszeit der modularen Exponentiation *jemals* kurz ist, obwohl die Berechnung der modularen Multiplikation $(s_b \cdot y) \bmod n$ für die gegebenen Werte s_b und y lange dauern müsste, dann wird die Multiplikation offenbar übersprungen und das gesuchte Exponentenbit ist wahrscheinlich 0.

- Wenn die gesamte Berechnungszeit der modularen Exponentiation für solche Werte s_b und y, die eine langsame modulare Multiplikation hervorrufen, *immer* lang ist, dann ist das gesuchte Bit wahrscheinlich 1.

Der Angreifer kann seine Hypothese überprüfen, indem er die dann folgende modulare Multiplikation $R_b^2 \bmod n$ in Zeile 9 nach demselben Schema untersucht: Für Werte von R_b, die eine langsame modulare Multiplikation hervorrufen, sollte auch die gesamte modulare Exponentiation langsam sein.

Dieses Verfahren wird wiederholt, bis alle w Bits des Exponenten x bestimmt sind.

Fehler lassen sich leicht erkennen: Wurde ein Exponentenbit b falsch geraten, sind die folgenden Werte $R_{k \geq b}$ nicht korrekt und für den Angreifer nicht vorhersehbar. Die erwartete Zeitdauer der modularen Multiplikationen wird sich dann nicht mehr in der tatsächlichen Berechnung widerspiegeln. Durch diese fehlende Korrelation kann der Angreifer erkennen, dass er einen Fehler gemacht haben muss, und diesen korrigieren (vgl. Kocher 1996, S. 105 f.).

Generalisierung des Angriffs

Ein realer Angreifer kann nicht von derart günstigen Bedingungen ausgehen. Der Angriff lässt sich jedoch mit Mitteln der Stochastik generalisieren.

Der Angriff ist prinzipiell ein Signalerkennungsproblem: Es ist nur die Gesamtausführungszeit T des Algorithmus bekannt. Diese setzt sich aus dem gesuchten Signal und Rauschen zusammen. Das gesuchte Signal ist die Zeitvarianz aufgrund des angegriffenen Exponentenbits, während das Rauschen durch Messungenauigkeiten und die Zeitvarianz aufgrund folgender, unbekannter Exponentenbits entsteht (vgl. ebd., S. 106).

Der Angreifer betrachtet nun die Ausführungszeiten von j verschiedenen Durchführungen des Square-and-Multiply-Algorithmus. Er nutzt zunächst die Opfer-Implementierung und variiert das y, wählt also j verschiedene Samples $y_0, \ldots, y_{j-1}$. Die Werte für x und n bleiben konstant. Er erhält j Ausführungszeiten $T_0, \ldots, T_{j-1}$. Jede Ausführungszeit T setzt sich aus den unbekannten einzelnen Ausführungszeiten der w Schleifeniterationen t_i und einem unbekannten Fehler e zusammen, der sich u. a. durch Messfehler und den Overhead der Schleifenausführung ergibt:

$$T = e + \sum_{i=0}^{w-1} t_i$$

Danach verwendet der Angreifer die Test-Implementierung. Auch hier rechnet er j Durchführungen des Algorithmus mit denselben y-Werten $y_0, \ldots, y_{j-1}$ wie zuvor. Er setzt n auf den ihm bekannten Modul, x auf die aktuelle Hypothese x_b. Er erhält j Ausführungszeiten $T_0^*, \ldots, T_{j-1}^*$. Jede Ausführungszeit T^* setzt sich aus einem Fehler e^* und den einzelnen Ausführungszeiten t_i^* der b Iterationen zusammen. Da die ersten $b-1$ Bits von x_b als bereits korrekt bestimmt angenommen werden, stimmen die Ausführungszeiten der ersten $b-1$ Iterationen mit denen Opfer-Implementierung (näherungsweise) überein ($\sum_{i=0}^{b-2} t_i^* = \sum_{i=0}^{b-2} t_i$). Somit ist T^* wie folgt zusammengesetzt:

$$T^* = e^* + \sum_{i=0}^{b-1} t_i^* = e + \underbrace{\sum_{i=0}^{b-2} t_i}_{\text{korrekt}} + \underbrace{t_{b-1}^*}_{\text{ungewiss}}$$

Im nächsten Schritt bildet der Angreifer die Differenz $T - T^*$:

$$\begin{aligned} T - T^* &= \left(e + \sum_{i=0}^{w-1} t_i \right) && - \left(e^* + \sum_{i=0}^{b-2} t_i + t^*_{b-1} \right) \\ &= \left(e + \sum_{i=0}^{b-2} t_i + t_{b-1} + \sum_{i=b}^{w-1} t_i \right) && - \left(e^* + \sum_{i=0}^{b-2} t_i + t^*_{b-1} \right) \\ &= \underbrace{(e - e^*)}_{:=e'} + \sum_{i=b}^{w-1} t_i + t_{b-1} - t^*_{b-1} \end{aligned} \tag{2.6}$$

Betrachtung der Varianzen. Jetzt berechnet der Angreifer die Varianz der j Differenzen $T - T^*$. Dies hat folgenden Hintergrund: Die Berechnung einer modularen Multiplikation, z. B. $(s_k \cdot y) \bmod n$ (Algorithmus 2.2, Zeile 5), dauert abhängig von den Operanden unterschiedlich lange (vgl. ebd., S. 109).

Darum variiert die Ausführungszeit t_i der i-ten Iteration des Square-and-Multiply-Algorithmus, über alle j Samples betrachtet, in einem gewissen Ausmaß, sofern die modulare Multiplikation in dieser Iteration gerechnet wird. Das ist der Fall, wenn das i-te Bit von x gesetzt ist. Die Varianz wird insbesondere dadurch hervorgerufen, dass das y in jedem Sample variiert. Wird die Berechnung nicht ausgeführt, weil das Exponentenbit nicht gesetzt ist, fällt der Einfluss dieser Berechnung auf die Zeitvarianz der Iteration weg. Diese Beobachtung nutzt der Angreifer zur Entscheidung, ob seine Hypothese korrekt ist oder nicht: Nur wenn die Hypothese korrekt ist, heben sich t_{b-1} und t^*_{b-1} in Gleichung 2.6 (näherungsweise) auf $(t_{b-1} - t^*_{b-1} = 0)$.

Für die Varianz der Summe oder Differenz zweier unabhängiger Zufallsvariablen gilt (vgl. G. Teschl und S. Teschl 2014, S. 283 f.):

$$\mathrm{Var}(X \pm Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$$

Da die Ausführungszeit der einzelnen Iterationen t_i unabhängig voneinander ist, ergibt sich folgende Varianz für die berechneten Differenzen:

$$\begin{aligned} \mathrm{Var}(T - T^*) &= \mathrm{Var}\Bigg(e' + \underbrace{\sum_{i=b}^{w-1} t_i}_{w-b \text{ Iterationen}} + t_{b-1} - t^*_{b-1} \Bigg) \\ &= \begin{cases} \mathrm{Var}(e') + (w-b) \cdot \mathrm{Var}(t) & \text{falls } x_b \text{ korrekt} \\ \mathrm{Var}(e') + (w-b+2) \cdot \mathrm{Var}(t) & \text{falls } x_b \text{ falsch} \end{cases} \end{aligned}$$

Ein korrekt geratenes, b-tes Bit verringert somit die Varianz, während ein falsch geratenes Bit die Varianz erhöht. Das gilt auch für in vorherigen Iterationen falsch geratene Bits (vgl. Kocher 1996, S. 107). Dies kann der Angreifer nutzen, um das b-te Bit zu bestimmen: er berechnet für die beiden Hypothesen $x_{b_0} = [x_{b-1}|0]$ und $x_{b_1} = [x_{b-1}|1]$ jeweils j Differenzen $T - T^*$ und berechnet deren Varianzen $\mathrm{Var}(T - T^*[0])$ und $\mathrm{Var}(T - T^*[1])$. Er entscheidet sich dann für die Hypothese mit der geringeren Varianz.

2.2.2 Modifikation des Angriffs: Betrachtung der modularen Multiplikationen

Dhem u. a. (2000) modifizieren Kochers Angriff, indem sie die modularen Multiplikationen genauer betrachten. Die Variablenbezeichnungen und Ablaufbeschreibungen der in diesem

Abschnitt beschriebenen Angriffe wurden so adaptiert, dass sie mit der in Algorithmus 2.2 angegebenen Variante des Square-and-Multiply-Algorithmus harmonieren.

Montgomery-Algorithmus. Dabei gehen sie davon aus, dass die modularen Multiplikationen nach dem Montgomery-Algorithmus (Algorithmus 2.3) gerechnet werden. Das Verfahren basiert auf der Idee, Multiplikationen modulo n durch Multiplikationen modulo einer Zweierpotenz 2^k zu ersetzen, da diese sehr effizient durch Shift-Operationen berechenbar sind.

Input: a, b, n (n ungerade)
Output: $a \cdot b \bmod n$

1 $r := 2^s$ mit $2^{s-1} \leq n \leq 2^s$.
2 Berechne $1 = r'r - n'n$ mit dem erweiterten euklidischen Algorithmus
3 $a' \leftarrow ar \bmod n$
4 $t \leftarrow a'b$
5 $m \leftarrow tn' \bmod r$
6 $u \leftarrow (t + mn)/r$
7 **if** $u \geq n$ **then**
8 **return** $u - n$
9 **else**
10 **return** u
11 **end if**

Algorithmus 2.3: Montgomery-Algorithmus zur modularen Multiplikation (nach: Welschenbach 2013, S. 108)

Die rechenaufwendigen Zeilen 1 und 2 müssen nur einmalig pro Modulus n berechnet werden und können für spätere Berechnungen mit demselben n wiederverwendet werden. Ist auch ein Faktor a konstant, kann auch der Zwischenwert a' aus Zeile 3 wiederverwendet werden.

Timing-Angriff auf die Multiply-Operation. Die Berechnungsdauer des Montgomery-Algorithmus mit einem konstanten Modulus n ist bis einschließlich Zeile 6 nahezu konstant, unterscheidet sich dann aber in Abhängigkeit davon, ob der berechnete Zwischenwert u größer ist als n oder nicht (vgl. Dhem u. a. 2000, S. 4).

Der Angreifer beginnt mit dem Angriff auf das zweite Bit ($k = 1$ im Square-and-Multiply-Algorithmus), da das höchstwertigste Bit immer als 1 angenommen werden kann. Falls das zweite Bit 1 ist, wird die Multiplikation $R_1 = s_1 \cdot y \bmod n$ gerechnet.

Für manche Nachrichten y ist das Zwischenergebnis u im Montgomery-Algorithmus größer oder gleich n. In diesen Fällen erfolgt eine zusätzliche Subtraktion. Der Angreifer rechnet selbst den Montgomery-Algorithmus für die Multiply-Operation für jeden ihm bekannten Ciphertext y_i. Dabei beobachtet er, ob eine zusätzliche Subtraktion ausgeführt wird oder nicht. Er teilt die y_i in zwei Mengen M_1 und M_2 ein. Die Menge M_1 enthält die y_i, die eine zusätzliche Subtraktion erfordern, die Menge M_2 enthält die übrigen y_i.

Falls das zweite Bit tatsächlich 1 ist, erwartet der Angreifer eine leicht erhöhte Berechnungsdauer für alle Ciphertexte aus M_1. Falls das zweite Bit 0 ist, muss das ausgewählte Separierungskriterium anhand des Montgomery-Zwischenwerts wie eine zufällige Separierung erscheinen, sodass sich die Berechnungszeiten zwischen den beiden Mengen nicht signifikant voneinander unterscheiden sollten. Der Angreifer wiederholt das Verfahren für die übrigen Exponentenbits (vgl. ebd., S. 5 f.).

Probleme. Dieser Lösungsansatz führt zu zwei Schwierigkeiten in der Praxis: erstens beobachteten Dhem u. a. bei den Multiplikationen mit konstantem Faktor y und konstantem Modulus n, dass die Wahrscheinlichkeit für eine zusätzliche Subtraktion – abhängig von y

und n – zwischen 0 und 0,5 schwankt, sodass sich eine Schiefe in dem verwendeten Auswahlkriterium offenbart. Zweitens muss der Angreifer entschieden, wann sich die Zeitmessungen für die beiden Mengen *signifikant* voneinander unterscheiden.

Dem zweiten Problem begegneten die Autoren mit Heuristiken und dem χ^2-Test. Außerdem stellen sie einen weiteren Ansatz zur Umgehung beider Probleme vor: die Betrachtung der Square-Operation (vgl. ebd., S. 6 ff.).

Timing-Angriff auf die Square-Operation. Der Angreifer greift die Iteration k des Square-and-Multiply-Algorithmus an. Er kennt die ersten k Bit des Exponenten und sucht das $k+1$-te Bit.

Für jedes Ciphertext-Sample y_i rechnet der Angreifer selbst die ersten $k-1$ Iterationen. Anschließend rechnet er zwei Mal die Iteration k:

1. Er nimmt an, das Bit k sei gesetzt. Er rechnet die Multiplikation und das Quadrieren mit dem Montgomery-Algorithmus. Falls beim Quadrieren eine Subtraktion notwendig ist, fügt er den Ciphertext y_i der Menge M_1 hinzu, andernfalls fügt er ihn der Menge M_2 hinzu.
2. Er nimmt an, das Bit k sei nicht gesetzt. In diesem Fall wird nur die Square-Operation ausgeführt. Der Angreifer separiert auch hier die Ciphertext-Samples y_i in zwei Mengen: M_3 (Subtraktion) und M_4 (keine Subtraktion).

Nur eines der beiden erstellten Modelle bildet die tatsächliche Berechnung ab. Der Angreifer berechnet für jede der vier Mengen die mittlere Berechnungsdauer und bildet die Differenzen zwischen M_1 und M_2 sowie zwischen M_3 und M_4. Ist die Differenz zwischen M_1 und M_2 größer als die Differenz zwischen M_3 und M_4, war das Bit k wahrscheinlich gesetzt, andernfalls war es wahrscheinlich nicht gesetzt (vgl. ebd., S. 8).

Die beiden zuvor beobachteten Probleme sind nun obsolet: Durch das Fehlen eines konstanten Faktors in der Square-Operation entfällt die Schiefe. Außerdem muss der Angreifer nicht mehr entscheiden, ob eine Differenz signifikant ist, sondern muss nur noch entscheiden, welche von zwei Differenzen signifikanter ist (vgl. ebd., S. 10).

3 Beschreibung des Angriffs

Dieses Kapitel beschreibt einen Timing-Angriff auf den McNie-Code. Abschnitt 3.1 zeigt mögliche Angriffsstellen im Code auf und legt dar, welche Informationen dadurch gefährdet sind. Abschnitt 3.2 beschreibt eine besonders gefährdete Codestelle genauer. In Abschnitt 3.3 ist eine Angriffsidee beschrieben, die in Abschnitt 3.4 zunächst simuliert und anschließend auf realen Systemen evaluiert wird. Das Kapitel schließt mit einer Bewertung der Ergebnisse und der Skizzierung möglicher Gegenmaßnahmen.

3.1 Auswahl eines Angriffspunkts zur weiteren Betrachtung

Um Angriffspunkte zu finden, wurde der Code durch Lesen und Ausführen mithilfe des GNU Debuggers (gdb)[1] und dessen Frontends Data Display Debugger (DDD)[2] nachvollzogen und auf Stellen, die variierendes Zeitverhalten in Abhängigkeit der verarbeiteten Daten aufweisen können, untersucht. Versuche einer automatischen Auswertung der Laufzeit einzelner Funktionen mit dem Profiler gprof[3] erwiesen sich aufgrund der zu geringen Zeitauflösung (0,01s) für diesen Zweck als nicht aussagekräftig. Der Code wurde daher manuell untersucht und dabei mit erklärenden Kommentaren versehen[4]. Alle Zeilennummern, die auf Stellen des McNie-Codes verweisen, beziehen sich auf diese kommentierte Version.

Es wurden insbesondere folgende Codestellen als mögliche Angriffspunkte identifiziert:

1. `gf.c`: `gf_mul(gf x, gf y)` (Z. 87)
   Diese Funktion multipliziert zwei Elemente `x` und `y` eines endlichen Körpers. Der Algorithmus iteriert bitweise über den Parameter `y` und addiert `x` genau dann auf eine temporäre Variable, wenn das betrachtete Bit von `y` gesetzt ist. Daher ist zu vermuten, dass die Ausführungszeit der Funktion abhängig von `x` und `y` variiert.
2. `gf.c`: `Div(gf x, gf y, gf *r)` (Z. 127)
   Diese Funktion dividiert zwei Körperelemente. Ähnlich zu der Funktion `gf_mul` wird der Operand `x` bitweise verarbeitet. Auch hier werden Bitoperationen in Abhängigkeit des betrachteten Bits ausgeführt oder nicht.

Der Rest dieses Kapitels konzentriert sich auf die Untersuchung des Zeitverhaltens der Funktion `gf_mul`. Ein Timing-Angriff auf diese Funktion erscheint geeignet zur Rekonstruktion des privaten Schlüssels, da sie im Code häufig und insbesondere auch zu Beginn des Entschlüsselungsvorgangs aufgerufen wird. Dort wird sie auf alle Elemente der Matrizen H und S^{-1} angewandt, wie Algorithmus 3.1 zeigt.

Unter Berücksichtigung der Tatsache, dass die Permutationsmatrix P gemäß der McNie-Dokumentation der Einheitsmatrix entspricht und S^{-1} durch Inversenbildung aus der invertierbaren Matrix S hervorgeht (siehe Abschnitt 2.1.3), reichen die zu diesem Zeitpunkt von der Funktion `gf_mul` verarbeiteten Daten aus, um den privaten McNie-Schlüssel $K_{\text{priv}} = (H, S, P)$ zu rekonstruieren. Zwei weitere Aspekte machen die Funktion zu einem interessanten Angriffsziel: Dadurch, dass S^{-1} zyklisch und H quasi-zyklisch ist, wiederholen sich deren Matrixelemente nach einem bekannten Muster. So muss der Angreifer einerseits weniger individuelle Werte erraten und bekommt andererseits noch die Möglichkeit, mit einer einzigen Ausführung des Entschlüsselungsalgorithmus mehrere Berechnungen mit demselben gesuchten Wert zu beobachten. Für die McNie-Parameter $n = 93, k = 62, d = 3$

[1] `https://www.gnu.org/software/gdb/`
[2] `https://www.gnu.org/software/ddd/`
[3] `https://www.gnu.org/software/binutils/`
[4] Der kommentierte Quellcode ist unter `https://git.fslab.de/tschlu2s/mcnie/tree/comments/Reference_Implementation/encrypt/3Q_128_1` abrufbar.

```
971   /* compute c1 * H^T */
972   syn = (gf *)malloc(nk * sizeof(*syn));
973   for( i=0 ; i<nk ; i++ ) {
974     // syn[i] wird vorab genullt
975     for( j=0, syn[i]=0 ; j<n ; j++ ) {
976       syn[i] ^= gf_mul(H.M.dat[i][j],gf_dat[j]);
977     }
978   }
979
980   /* compute c * S^(-1) */
981   c3 = (gf *)malloc(nk * sizeof(*c3));
982   for( j=0 ; j<nk ; j++ ) {
983     // c3[j] wird vorab genullt
984     for( i=0,c3[j]=0 ; i<nk ; i++ ){
985       c3[j] ^= gf_mul(gf_dat[n+i],Sinv.dat[i][j]);
986     }
987   }
```

Algorithmus 3.1: Auszug aus der Funktion `decrypt_one_block_return_error` (aus der kommentierten Version von `crypt.c`)

müssen zur eindeutigen Bestimmung der Matrix H $(n-k)\cdot d = 93$ Körperelemente ermittelt werden. Um die Matrix S bzw. S^{-1} zu erhalten, sind $n-k = 31$ Körperelemente erforderlich. Der zweite Operand aller `gf_mul`-Berechnungen in Algorithmus 3.1, `gf_dat[]`, leitet sich direkt aus dem Chiffrattext ab. In einem Chosen-Ciphertext-Szenario können diese Werte vom Angreifer frei gewählt werden.

3.2 Die Funktion `gf_mul`

Algorithmus 3.2 zeigt die Funktion `gf_mul`. Der Datentyp `gf` ist ein Alias für den vorzeichenlosen 64-Bit-Datentyp `unsigned long long` und repräsentiert ein Element aus einem endlichen Körper $\mathbb{F}_{2^m}$ mit $m < 64$. Die Koeffizienten des Elements werden in den unteren m Bit der Variable gespeichert. Das folgende Beispiel verdeutlicht diese Repräsentation.

Polynomdarstellung: $x^5 + x^3 + x + 1$
Darstellung im Datentyp `gf`:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00101011

Russische Bauernmultiplikation. Der Algorithmus 3.2 basiert auf dem Grundprinzip der Russischen Bauernmultiplikation, welches um Modulo-Reduktion erweitert wurde. Die Russische Bauernmultiplikation wird ursprünglich zur Multiplikation zweier natürlicher Zahlen verwendet. Einer der Faktoren wird fortgesetzt verdoppelt, der andere wird fortgesetzt ohne Rest halbiert, bis er 1 erreicht. Die Zwischenergebnisse der Verdopplungen und Halbierungen werden in je einer Tabellenspalte untereinander notiert. Anschließend werden die Zwischenergebnisse der Verdopplung in den Zeilen miteinander addiert, in denen das Ergebnis der nebenstehenden Halbierung ungerade ist. Diese Summe entspricht dem gesuchten Produkt (vgl. Forster 2015, S. 12 f., dort ist auch ein Beispiel gegeben).

`gf_mul` überträgt dieses Verfahren auf Elemente endlicher Körper. Die Verdopplung und Halbierung der Faktoren sowie die Addition von Zwischenergebnissen sind auf den Binärwerten effizient durch Shift- und XOR-Operationen implementiert.

Die `while`-Schleife iteriert so lange, bis `y` durch fortgesetzte Halbierung (ohne Rest) gleich Null ist. Immer dann, wenn `y` ungerade ist, also das niederwertigste Bit gesetzt ist, wird der aktuelle Wert von `x` auf die Ergebnisvariable `rst` aufaddiert. Falls das m-te Bit von `x` gesetzt

```
 7 const short _m = 37;
 :
10 const gf base_poly[65] = {
 :
22 0x3300000019, /* x^37 + x^36 + x^33 + x^32 + x^4 + x^3 + 1 */
 :
56 };
 :
86 /* multiplication x * y */
87 gf gf_mul(gf x, gf y) {
88   // Result
89   gf rst = 0;
90   // Flag zum Abfragen des 37. Bits
91   gf max_flag = 1;
92   max_flag <<= (_m - 1); // = 2^36
93   // Solange y != 0...
94   while( y ) {
95     // Falls LSB(y) == 1 (y ungerade), addiere x zu rst.
96     if( y & 1 ) rst ^= x;
97     // Modulo-Rechnung: Falls x > 2^36 (_m-tes Bit gesetzt)
98     if( x & max_flag ) {
99       // Verdopple x
100      x <<= 1;
101      // Addiere irreduzibles/primitives Polynom
102      x ^= base_poly[_m];
103    } else {
104      // Verdopple x
105      x <<= 1;
106    }
107    // Halbiere y (ohne Rest)
108    y >>= 1;
109  }
110  return rst;
111 }
```

Algorithmus 3.2: Funktion `gf_mul` (aus der kommentierten Version von `gf.c`)

x	y	rst
1010 ($= x^3 + x$) (x & max_flag) ✗	1011 ($= x^3 + x + 1$) (y & 1) ✓	0 rst $= 0 \oplus 1010 = 1010$
10100 (x & max_flag) ✗	101 (y & 1) ✓	1010 rst $= 1010 \oplus 10100 = 11110$
101000 (x & max_flag) ✗	10 (y & 1) ✗	11110
1010000 (x & max_flag) ✗	1 (y & 1) ✓	11110 rst $= 11110 \oplus 1010000 = 1001110$
10100000	0	1001110
Ergebnis: 1001110 ($= x^6 + x^3 + x^2 + x$)		

Tabelle 3.1: Beispielhafter Ablauf der Funktion `gf_mul` für kleine Werte

ist, kann x durch Subtraktion des irreduziblen Polynoms reduziert werden. Tabelle 3.1 zeigt einen beispielhaften Durchlauf mit kleinen Werten für x und y.

Variable Ausführungszeit. Der Code legt nahe, dass die Ausführungszeit einer Schleifeniteration von zwei Faktoren abhängt:

1. Dem niederwertigsten Bit von y: Nur wenn es gesetzt ist, wird die XOR-Operation ausgeführt.
2. Dem m-ten Bit von x: Nur wenn es gesetzt ist, wird die Modulo-Reduktion ausgeführt.

3.3 Angriffsidee

Voraussetzungen. Im Kontext des gesamten McNie-Codes besteht für den Angreifer die Herausforderung, die Zeitdauer einzelner gf_mul-Operationen zu messen. Je nach Plattform könnte dies durch die Auswertung der Stromaufnahme oder anderer Seitenkanäle möglich sein. Die Betrachtung im Rahmen dieser Arbeit beschränkt sich zunächst auf die Funktion gf_mul ohne Berücksichtigung des weiteren Anwendungskontexts und unter der Annahme, dass die Zeitmessung jeder einzelnen gf_mul-Ausführung möglich ist.

Analogie zu Kochers Angriff. Insbesondere bei der Verarbeitung von y ist eine Analogie zum Square-and-Multiply-Algorithmus erkennbar. y wird, wie das x des Square-and-Multiply-Algorithmus, bitweise durchlaufen. Die Betrachtung des Codes führt zu der Hypothese, dass die Ausführungszeit einer Iteration abhängig vom jeweils betrachteten Bit variiert. Ist das Bit gesetzt, wird eine Addition von rst und x ausgeführt. Lediglich die Laufrichtung ist vertauscht: gf_mul läuft vom niederwertigsten zum höchstwertigsten Bit von y. Es liegt somit nahe, Kochers Timing-Angriff (siehe Abschnitt 2.2.1) zu adaptieren. Die Erweiterung von Dhem u. a. (siehe Abschnitt 2.2.2) ist dagegen kaum anwendbar, da bisher kein Kriterium zur Auswahl von x-Werten erkennbar ist, das einen Einfluss auf die Berechnungsdauer hat.

Angreifermodell. Das Angreifermodell gestaltet sich analog zu Kochers Ansatz: Der Angreifer hat Zugriff auf zwei Implementierungen von gf_mul, wie in Abbildung 3.1 dargestellt. Eine dieser Implementierungen enthält einen privaten Wert y, der auf eine Komponente der Matrix S^{-1} gesetzt ist. Der Angreifer kann einen beliebigen x-Wert eingeben, den die Implementierung mit y multipliziert. Die andere Implementierung erlaubt das freie Setzen beider Operanden x und y. Außerdem kann der Angreifer bei beiden Implementierungen die Gesamtausführungszeit einer Multiplikation messen.

Übertragen auf den McNie-Code handelt es sich dabei um ein Chosen-Ciphertext-Szenario. Der Angreifer kann bei der Berechnung von syn[i] (Algorithmus 3.2, Z. 976) das x frei wählen, indem er die Entschlüsselungsfunktion mit einem entsprechenden Ciphertext aufruft.

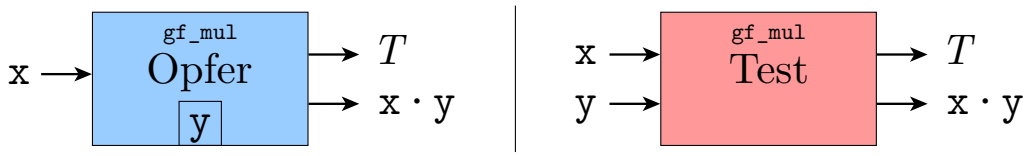


Abbildung 3.1: Angreifermodell

Ablauf. Der Angreifer wählt zufällig j Samples x_j. Diese gibt er zunächst in die Opfer-Implementierung ein und misst j Zeiten $T_0, \ldots, T_{j-1}$. Anschließend nähert er sich unter Verwendung der Test-Implementierung dem geheimen y iterativ an.

Dazu beginnt er mit den y-Hypothesen $y_{0_0} = 0$ und $y_{0_1} = 1$. Für jede y-Hypothese und für alle j x-Samples berechnet er gf_mul(x_i, y_0) und misst die Ausführungszeiten. Anschließend berechnet er die Varianz $\mathrm{Var}(T - T^*[0])$ über alle Zeitdifferenzen mit der

Hypothese $\mathtt{y}_{0_0}$ und die Varianz $\mathrm{Var}(T - T^*[1])$ über alle Zeitdifferenzen mit der Hypothese $\mathtt{y}_{0_1}$.

Er entscheidet sich für die Hypothese mit der geringeren Varianz und wiederholt das Verfahren für das zweitniedrigste und alle weiteren y-Bits.

3.4 Durchführung des Angriffs

Der beschriebene Angriff wurde zunächst simuliert und anschließend auf reale Plattformen übertragen. Das Ziel ist es jeweils, den geheimen y-Wert `10101 00011110 10110100 00100001 01011001` zu finden.

3.4.1 Simulation

Eine Simulation des Angriffs erlaubt dessen Überprüfung ohne den Einfluss von Messfehlern. Dazu wurde der Ablauf des Angriffs in C implementiert. Die Funktion `gf_mul` wurde in diesen C-Code eingefügt und so modifiziert, dass sie nach jeder Instruktion eine globale Variable `cyclecount` erhöht. Falls das niederwertigste Bit von `y` gesetzt ist, bestimmt sich die Höhe der Erhöhung in Abhängigkeit von `x`. Die Variable `cyclecount` simuliert somit einen CPU-Zyklen-Zähler während der Ausführung von `gf_mul`. Dieser wird von dem Code, der den Angriff steuert, wie in Abschnitt 3.3 beschrieben ausgewertet[5].

Abbildung 3.2 zeigt die in jeder Iteration berechneten Varianzen $\mathrm{Var}(T - T^*)$ für beide möglichen Hypothesen. Der Simulator entscheidet sich in jeder Runde korrekt, das gesuchte `y` ist aus dem Graphen ablesbar.

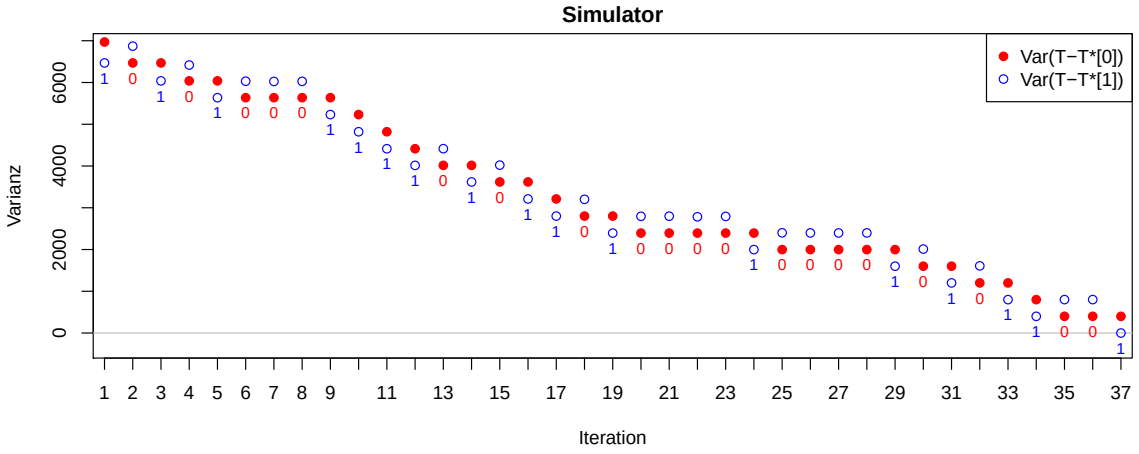


Abbildung 3.2: Verlauf der Varianzen im Simulator

3.4.2 Übertragung auf reale Plattformen

Der Simulator-Code wurde im nächsten Schritt auf die in Tabelle 3.2 aufgelisteten Plattformen übertragen. Die Messung der Ausführungszeit erfolgt durch einen plattformabhängigen Befehl, der vor und nach jedem `gf_mul`-Aufruf aufgerufen wird und eine möglichst genaue Zeitangabe oder den Zustand eines CPU-Instruktionszählers abfragt. Dieser Wert wird jeweils gespeichert. Die Differenz der beiden Zeitstempel ist die Ausführungszeit.

x86 und x86_64. Die Testsysteme 1 bis 3 sind mit 32- bzw. 64-Bit-CPUs von Intel bzw. AMD ausgestattet. Alle drei CPUs führen einen 64-Bit-Time Stamp Counter (TSC). Dieser

[5] Der Simulator-Code ist unter `https://git.fslab.de/tschlu2s/mcnie/tree/timing-experiments/Reference_Implementation/encrypt/timing-experiments/04-gfmul-sim-2` abrufbar.

#	**Architektur/Befehlssatz**	**Testsystem**	**CPU/Controller**	**Betriebssystem**	**Methode der Zeitmessung**
1	x86_64	HP EliteBook 2760p	Intel Core i7-2620M	Linux (Ubuntu 18.04.2, Kernel 4.18.0-16-generic x86_64)	Instruktion `rdtscp`
2	x86_64	Acer Aspire 5536	AMD Athlon X2 QL-64	Linux (Alpine Linux 3.9.2, Kernel 4.19.26-0-vanilla x86_64)	Instruktion `rdtscp`
3	x86	Fujitsu Siemens Futro A240	AMD Geode LX800	Linux (Debian 9, Kernel 4.9.0-8-686)	Instruktion `rdtsc`
4	ARMv6	Raspberry Pi Model B (1. Generation)	Broadcom BCM2835	Linux (Raspbian 9, Kernel 4.14.79+ armv6l)	Auslesen des Cycle-Count-Registers
5	AVR	Arduino Uno (Nachbau)	Atmel ATmega 328P (16MHz)	—	Timer
6	AVR	SparkFun Pro Micro 3,3V (Nachbau)	Atmel ATmega 32U4 (8MHz)	—	Pin Toggle und Oszilloskop (24MS/s)

Tabelle 3.2: Testsysteme

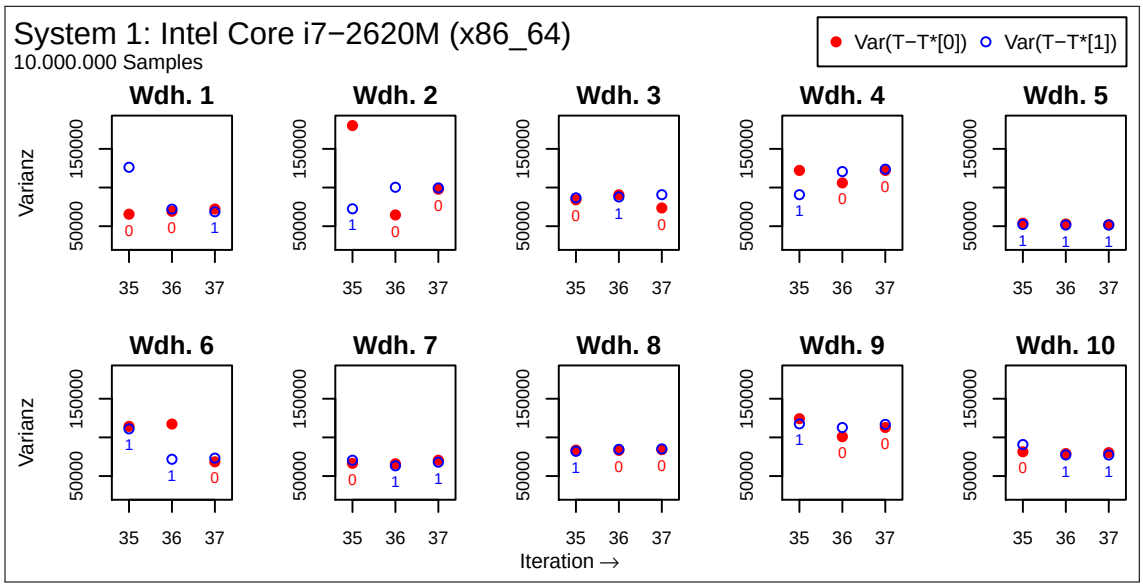
Zähler startet bei einem Reset der CPU bei 0 und wird mit jedem Taktzyklus erhöht (vgl. Advanced Micro Devices Inc. 2018, S. 401; Intel Corp. 2016, S. 4–547; Advanced Micro Devices Inc. 2009, S. 110).

Der TSC lässt sich auf diesen Systemen mit der Assembler-Instruktion `rdtsc` bzw. `rdtscp` auslesen. Die Instruktion `rdtscp` wartet mit dem Auslesen des Zählers, bis alle vorherigen Instruktionen ausgeführt wurden. `rdtsc` garantiert das nicht (vgl. Intel Corp. 2016, S. 4–545 ff.). Daher wird `rdtscp` bevorzugt verwendet. Diese Instruktion steht im Befehlssatz der x86-CPU allerdings nicht zur Verfügung (vgl. Advanced Micro Devices Inc. 2009, S. 640).

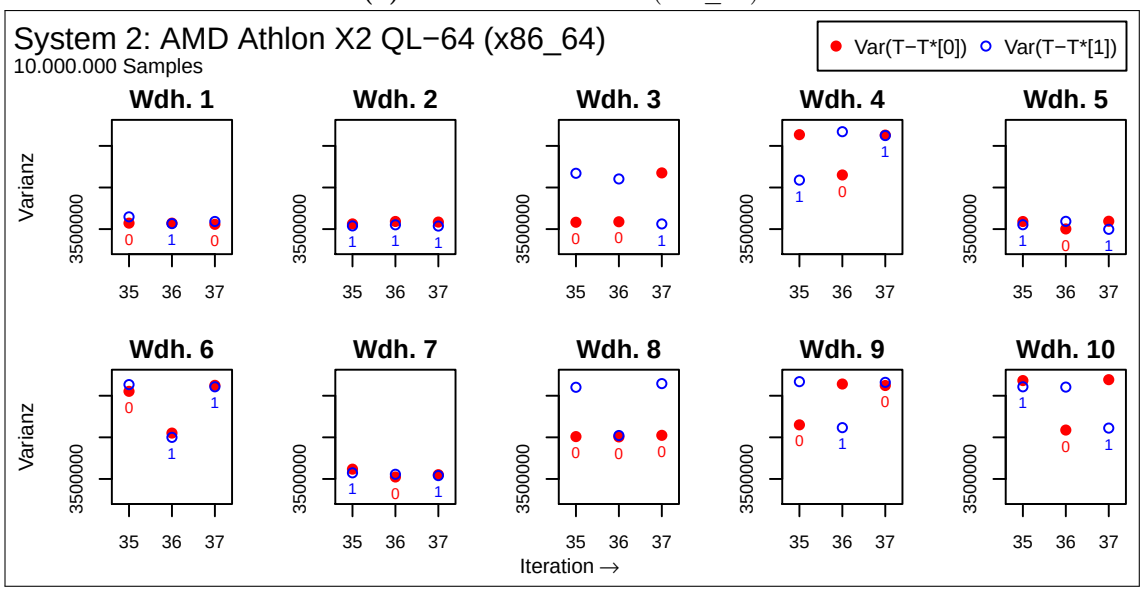
Die Abbildungen 3.3a bis 3.3c zeigen die Auswertung der Varianzen von 10 nacheinander ausgeführten Versuchen, die letzten drei Stellen von `y` zu bestimmen[6]. Es wurden in jedem Versuch $j = 10.000.000$ `x`-Samples verwendet.

Es ist zu erkennen, dass die Systeme 1 und 2 das korrekte Ergebnis `001` nicht oder zumindest nicht zweifelsfrei erreichen. Der Grund dafür kann einerseits ein großer Fehler aufgrund von Messungenauigkeiten und parallelen Systemaktivitäten, andererseits eine geringe `x`-abhängige Zeitvarianz der XOR-Berechnung sein. Besser schneidet die x86-CPU (System 3) ab, die in 4 von 10 Versuchen das korrekte Ergebnis erkennt. Aber auch dort führen offenbar Störeinflüsse dazu, dass der Angriff nicht stabil und zuverlässig ausgeführt werden kann. Auch eine höhere Sample-Anzahl konnte die Stabilität nicht verbessern, sondern verschlechterte das Ergebnis sogar.

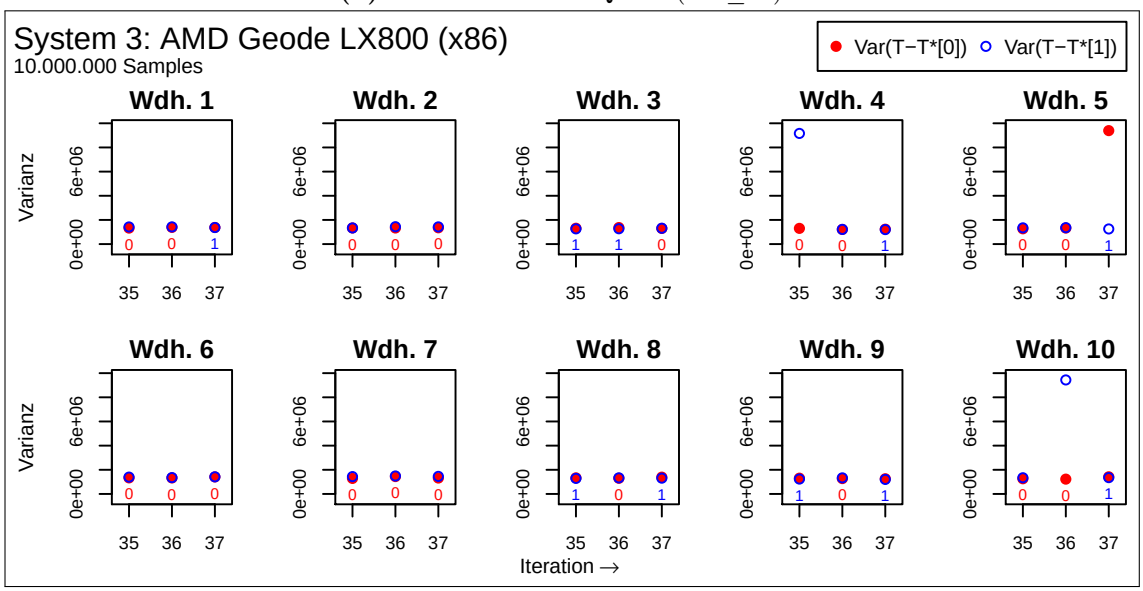
[6]Der dazu verwendete Quellcode ist unter `https://git.fslab.de/tschlu2s/mcnie/tree/timing-experiments/Reference_Implementation/encrypt/timing-experiments/05-gfmul-2` abrufbar.



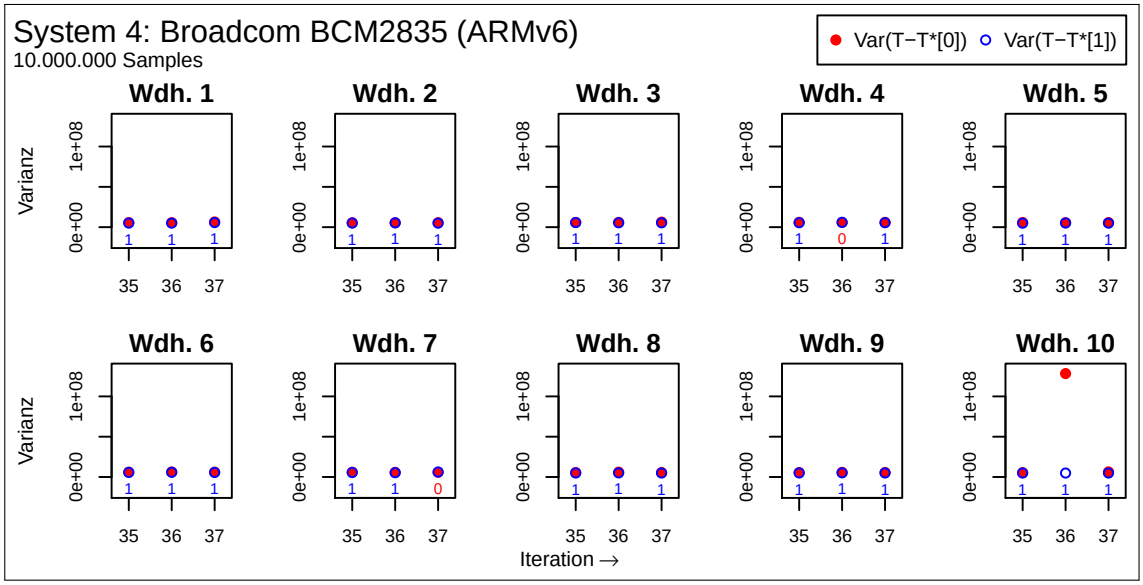
(a) Intel Core i7-2620M (x86_64)



(b) AMD Athlon X2 QL-64 (x86_64)



(c) AMD Geode LX800 (x86)



(d) Broadcom BCM2835 (ARMv6)

Abbildung 3.3: Verlauf der Varianzen für die letzten 3 Stellen in je 10 Wiederholungen auf realen Testsystemen

ARMv6 (Raspberry Pi). Das Analogon zum TSC der x86/x86_64-Architekturen in der ARM-Architektur ist das Cycle Counter Register (CCNT). In diesem 32-Bit-Register werden die Taktzyklen der CPU gezählt. Bei einem Reset der CPU ist das Register in einem undefinierten Zustand, es kann jedoch über das Performance Monitor Control Register (PMNC) zurückgesetzt werden (vgl. Arm Ltd. 2009, S. 3–137 f.). Um das CCNT in einem Programm auslesen zu können, muss der Zugriff auf das Register aus dem User Mode zunächst freigegeben werden. Dazu muss mit Kernel-Privilegien das niederwertigste Bit des Secure User and Non-secure Access Validation Control Register auf 1 gesetzt werden (vgl. ebd., S. 3–138, 3–132). Dies kann bspw. durch das Laden eines entsprechenden Kernel-Moduls geschehen[7].

Die Auswertung des Tests mit einem ARM-System in Abbildung 3.3d[8] zeigt, dass auch hier offenbar viele Störeinflüsse vorliegen. Auffällig ist, dass die Kombination `111` gehäuft vorkommt.

AVR (Arduino Uno). Die Ausführung von `gf_mul` auf einem Mikrocontroller ermöglicht es, die Störeinflüsse durch parallel ausgeführte Prozesse und Interrupts deutlich zu verringern. Das Arduino-Uno-Entwicklerboard eignet sich für die Durchführung des Experiments, da es neben dem ATmega 328P die notwendigen Komponenten zur seriellen Kommunikation zwischen Rechner und Mikrocontroller per USB mitbringt und sich sehr einfach in der Sprache C++ programmieren lässt. Da der Mikrocontroller geringere Ressourcen als die zuvor verwendeten Systeme bietet, muss der Programmcode des Angriffs aufgeteilt werden: Der Mikrocontroller rechnet lediglich die Funktion `gf_mul` mit den Parametern `x` und `y`, die über eine serielle Schnittstelle von einem PC vorgegeben werden. Der Arduino antwortet auf derselben Schnittstelle mit der benötigten Berechnungszeit[9]. Zur Zeitmessung kann

[7] Der Quellcode zu einem entsprechenden Kernel-Modul ist unter `https://git.fslab.de/tschlu2s/mcnie/tree/timing-experiments/Reference_Implementation/encrypt/timing-experiments/05-gfmul-2/arm_module` zu finden.

[8] Es wurde derselbe Quellcode wie bei x86/x86_64 verwendet. Der Code ist so gestaltet, dass er für ARM, x86 und x86_64 jeweils die passende Messmethode verwendet.

[9] Der Quellcode für den Arduino sowie ein Python-Skript zu dessen Steuerung ist unter `https://git.fslab.de/tschlu2s/mcnie/tree/timing-experiments/Reference_Implementation/encrypt/timing-experiments/06-gfmul-arduino` zu finden.

der integrierte Timer des ATmega 328P verwendet werden, der einen Zähler besitzt, der mit derselben Taktrate wie die CPU inkrementiert werden kann (vgl. Atmel Corp. 2015, S. 110). Dieser Zähler wird im Programmcode ausgewertet.

Die Auswertung ist in Abbildung 3.4 dargestellt, sämtliche Varianzen sind 0: Die Ausführungszeit von `gf_mul` variiert zwar insgesamt in Abhängigkeit von `y`, jedoch überhaupt nicht in Abhängigkeit des gewählten `x`. Das gleiche Ergebnis stellt sich dar, wenn auch `x`-Werte gewählt werden, bei denen nur die unteren 8, 16 oder 32 zufällig gesetzt und die oberen Bit 0 sind. Die Dauer einer XOR-Operation scheint auf dem Arduino Uno – zumindest mit der Messauflösung, die der Timer-Counter bietet – unabhängig von den Operanden zu sein.

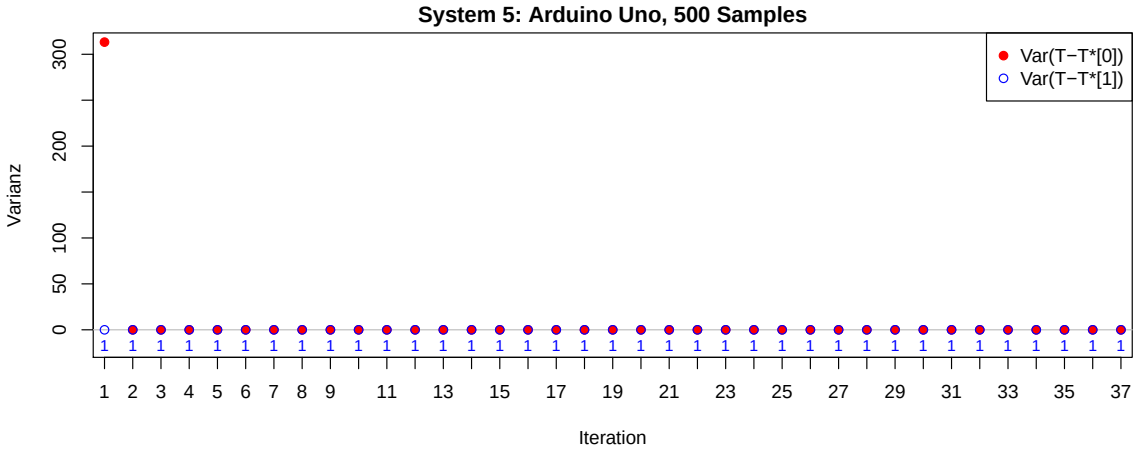


Abbildung 3.4: Verlauf der Varianzen auf dem Arduino Uno

AVR (SparkFun Pro Micro). Ein Versuch, die Auflösung der Zeitmessung weiter zu erhöhen, erfolgte mit dem Entwicklerboard SparkFun Pro Micro (3,3V). Dieses Board verfügt über einen ATmega32U4-Mikrocontroller und die notwendige Peripherie, die zur seriellen Kommunikation mit einem PC über USB erforderlich ist. Der Mikrocontroller wird mit einer Taktfrequenz von 8 MHz betrieben. Das Board ist kompatibel mit der Arduino-IDE, sodass der Code vom Arduino Uno wiederverwendet werden kann.

In diesem Versuch kommt eine andere Messmethode zum Einsatz, die eine höhere zeitliche Auflösung verspricht: Vor jeder `gf_mul`-Ausführung wird ein Pin des Boards auf HIGH gesetzt, nach der Berechnung auf LOW. Dieses Signal wird von einem digitalen Oszilloskop mit 24 MS/s abgetastet. Das Steuerprogramm[10] erhält die Messwerte des Oszilloskops und wertet diese aus: Bei einer steigenden Flanke beginnt eine `gf_mul`-Berechnung, bei einer fallenden Flanke endet sie. Die Anzahl der Samples zwischen der steigenden und der fallenden Flanke gibt die Ausführungszeit an.

Abbildung 3.5 zeigt den Versuchsaufbau. Für das verwendete Oszilloskop vom Typ Hantek 6022BE existiert eine freie Python-Bibliothek[11], die zur Ansteuerung des Messgeräts verwendet wurde. Nach dem Nyquist-Shannon-Abtasttheorem wäre die Abtastrate von 24 MS/s zu gering, um diesen Versuch mit dem Arduino Uno durchzuführen, dessen Mikrocontroller mit 16 MHz taktet. Für den Pro Micro (8 MHz) reicht sie jedoch aus.

Die Messergebnisse sind in Abbildung 3.6 dargestellt. Die Varianzen sind nun ungleich 0, liefern dem Angreifer aber kein klares Ergebnis. Offenbar sind auch bei der Betrach-

[10] Der Quellcode für den Pro Micro sowie ein Python-Skript, das den Mikrocontroller und das Oszilloskop ansteuert und den Angriff durchführt, ist unter `https://git.fslab.de/tschlu2s/mcnie/tree/timing-experiments/Reference_Implementation/encrypt/timing-experiments/07-gfmul-arduino-osc` zu finden.

[11] `https://github.com/rpcope1/Hantek6022API`

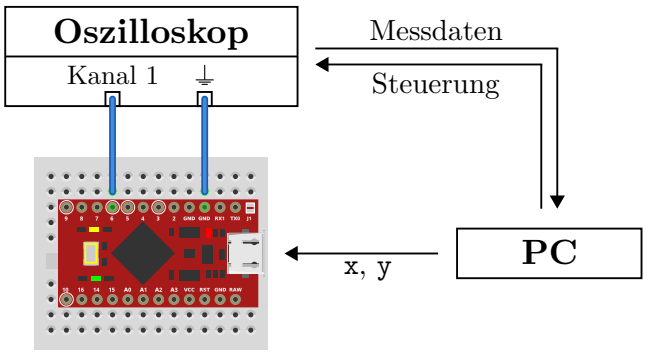


Abbildung 3.5: Versuchsaufbau

tung der Ausführungszeiten in einer erhöhten zeitlichen Auflösung keine ausreichenden Abhängigkeiten der Ausführungszeit vom Parameter `x` vorhanden.

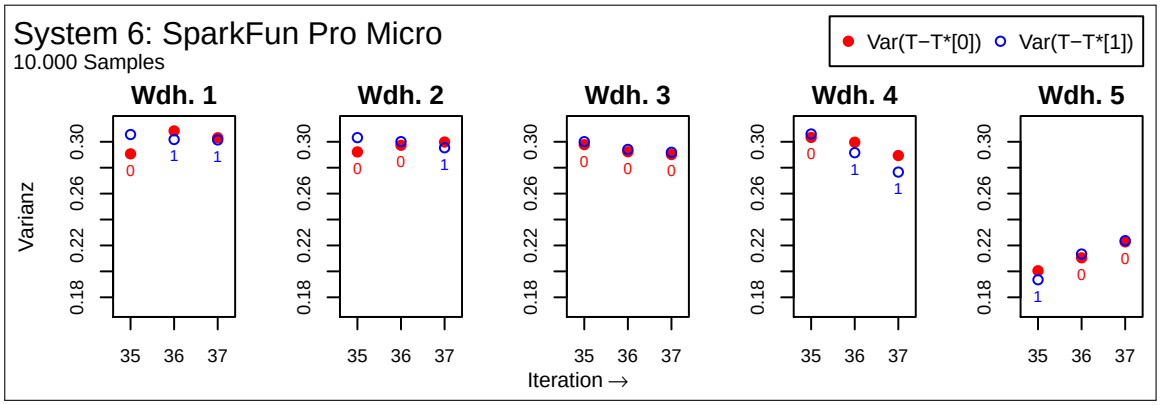


Abbildung 3.6: Verlauf der Varianzen für die letzten 3 Stellen in 5 Wiederholungen auf dem SparkFun Pro Micro

3.5 Bewertung

Die oben beschriebenen Versuche zeigen, dass das Ausnutzen der Schwachstelle auf den betrachteten Plattformen nicht trivial ist. Es ist im Rahmen des Projekts nicht gelungen, den Angriff stabil und reproduzierbar in die Praxis umzusetzen. Die Ausnutzbarkeit unter der Bedingung, dass die Ausführungszeit der XOR-Operation in Abhängigkeit des Operanden `x` variiert, konnte jedoch mithilfe einer Simulation theoretisch gezeigt werden. Übertragen auf McNie geht davon eine Gefährdung der Matrix S bzw. S^{-1} aus. Weitere Überlegungen und Versuche, die an diese Arbeit anknüpfen können, müssen zeigen, ob Störeinflüsse aus den Zeitmessungen herausgefiltert werden können und ob so bessere Ergebnisse erzielbar sind. Außerdem kann untersucht werden, ob der Funktionsparameter `x` rekonstruierbar ist (Angriff auf die Matrix H).

Letztlich ist nicht auszuschließen, dass Optimierungen des Angriffs die Ergebnisse verbessern. Den Entwicklern des McNie-Codes ist daher anzuraten, die Implementierung von `gf_mul` zu überarbeiten.

3.6 Gegenmaßnahmen

In der Literatur sind verschiedene Gegenmaßnahmen beschrieben, die Timing-Angriffe verhindern oder die Erfolgswahrscheinlichkeit des Angreifers verringern.

Hiding. Durch die zufällige Ausführung von Dummy-Operationen oder das Einbringen von zufälligen Verzögerungen kann das Signal-zu-Rausch-Verhältnis verschlechtert werden. Der Angreifer kann darauf mit der Betrachtung von mehr Samples reagieren, in der Hoffnung, dass die künstlichen, zufälligen Verzögerungen sich bei großen Sample-Zahlen statistisch ausgleichen (vgl. Kocher 1996, S. 111).

Kritische Operation immer ausführen. Alternativ kann die angreifbare Operation in jedem Fall ausgeführt werden. Gerade auf ressourcenschwachen Geräten kann dies jedoch zu Performance-Verlusten führen. Außerdem sind Timing-Angriffe damit nicht grundsätzlich ausgeschlossen, sondern lediglich erschwert: Variiert die Ausführungszeit einer immer ausgeführten Addition abhängig von den Summanden, kann der Angreifer dennoch versuchen, diese Zeitvarianzen auszuwerten (vgl. ebd.). Algorithmus 3.3 zeigt einen Verbesserungsvorschlag für die Funktion `gf_mul`, die die Operation `rst ^ x` immer ausführt.

```
1  while(y) {
2    gf tmp_y[2] = {rst, rst};
3    tmp_y[1] ^= x;
4    rst = tmp_y[y & 1];
5
6    gf condition = (x & max_flag);
7    x <<= 1;
8    gf tmp_y = {x, x};
9    tmp_x[1] ^= base_poly[_m];
10   x = tmp_x[condition ? 1 : 0];
11
12   y >>= 1;
13 }
```

Algorithmus 3.3: Entwurf einer Alternativimplementierung der Schleife in `gf_mul`

Blinding. Der geheime Wert kann vor der Ausführung der Berechnung mit einem zufälligen Wert verknüpft werden, der nachher wieder herausgerechnet wird (vgl. ebd.). Durch die zufällige Wahl kann der Angreifer die Funktionsparameter nicht mehr frei bestimmen. Unter der Annahme, dass der `y`-Wert besonders gefährdet ist, könnte folgendes Verfahren angewandt werden:

- Wähle ein zufälliges Körperelement `a`.
- Bestimme inverses Element $\texttt{a}^{-1}$.
- Berechne:

$$\begin{aligned} \texttt{tmp1} &= \texttt{gf\_mul(a, x)} \\ \texttt{tmp2} &= \texttt{gf\_mul(y, tmp1)} \\ \texttt{result} &= \texttt{gf\_mul(tmp2, a}^{-1}\texttt{)} \end{aligned}$$

Protokoll-Einschränkungen. Der Zugriff auf die kritische Funktion kann so reglementiert werden, dass es dem Angreifer nicht möglich ist, ausreichend viele Samples zu sammeln und/oder die Parameter `x` und `y` in der erforderlichen Weise zu beeinflussen.

4 Zusammenfassung und Ausblick

Die Funktion `gf_mul` zur Multiplikation von Elementen eines endlichen Körpers, die die Referenz-Implementierung des McNie-Kryptosystems verwendet, ist anfällig für Timinig-Angriffe. Diese Funktion wird zu Beginn einer jeden Entschlüsselungsoperation verwendet und verarbeitet sämtliche geheimen Anteile des privaten Schlüssels. In einem Chosen-Ciphertext-Szenario hätte ein Angreifer die Möglichkeit, einen der beiden Funktionsparameter von `gf_mul` frei zu wählen. Somit bestehen prinzipiell günstige Bedingungen für einen Angrifer, um den privaten Schlüssel zu rekonstruieren.

In der vorliegenden Arbeit wurde ein Angriff auf den Funktionsparameter `y` analog zu Kochers Timing-Angriff auf den Square-and-Multiply-Algorithmus entwickelt und evaluiert. Dieser Angriff konnte in einer Simulation erfolgreich ausgeführt werden. Vermutlich aufgrund von Störeinflüssen und relativ konstanter Berechnungsdauer der XOR-Operation auf den betrachteten Plattformen konnte der Angriff jedoch nicht stabil und reproduzierbar in die Praxis übertragen werden.

Dieses Ergebnis wirft weiterführende Fragen auf, die in weiteren Untersuchungen bearbeitet werden können. Dazu zählen die Angriffsmöglichkeiten auf den Funktionsparameter `x`. Außerdem muss untersucht werden, ob die Zeitdauer einer einzigen `gf_mul`-Ausführung im Kontext einer vollständigen McNie-Entschlüsselungsoperation bestimmt werden kann. Je nach Plattform könnte dafür bspw. die Stromaufnahme der Implementierung ausgewertet werden (Simple/Differential Power Analysis). Außerdem können andere Implementierungen von Kryptosystemen, die auf endlichen Körpern operieren, dahingehend untersucht werden, ob sie ähnliche Schwachstellen aufweisen.

5 Literaturverzeichnis

Advanced Micro Devices Inc., Hrsg. (2009). *AMD Geode™ LX Processors Data Book*. Feb. 2009. URL: https://www.amd.com/system/files/TechDocs/33234H_LX_databook.pdf (besucht am 03. 04. 2019).

Advanced Micro Devices Inc., Hrsg. (2018). *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions*. Mai 2018. URL: https://www.amd.com/system/files/TechDocs/24594.pdf (besucht am 03. 04. 2019).

Arm Ltd. (2009). *ARM1176JZF-S Technical Reference Manual*. 27. Nov. 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf (besucht am 03. 04. 2019).

Atmel Corp., Hrsg. (2015). *ATmega328P Datasheet*. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (besucht am 03. 04. 2019).

Bernstein, Daniel J. (2005). *Cache-timing attacks on AES*. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf (besucht am 08. 12. 2018).

Chen, C. L., W. W. Peterson und E. J. Weldon (1969). *Some results on quasi-cyclic codes*. In: *Information and Control*, Jg. 15, Nr. 5 (Nov. 1969), S. 407–423. DOI: 10.1016/S0019-9958(69)90497-5.

Dhem, Jean-François u. a. (2000). *A Practical Implementation of the Timing Attack*. In: *Smart Card Research and Applications*. Hrsg. von Jean-Jacques Quisquater und Bruce Schneier. Bd. 1820. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 167–182. DOI: 10.1007/10721064_15. URL: http://www.cs.jhu.edu/~fabian/courses/CS600.624/Timing-full.pdf (besucht am 14. 01. 2019).

Forster, Otto (2015). *Algorithmische Zahlentheorie*. Wiesbaden: Springer Fachmedien Wiesbaden. DOI: 10.1007/978-3-658-06540-9.

Gabidulin, Ernst (1985). *Theory of Codes with Maximum Rank Distance*. In: *Problems of Information Transmission*, Jg. 21, Nr. 1. Englische Übersetzung aus dem Russischen, S. 1–12. URL: https://www.researchgate.net/publication/235008632_Theory_of_codes_with_maximum_rank_distance_translation (besucht am 04. 12. 2018).

Gaborit, Philippe u. a. (2013). *Low Rank Parity Check codes and their application to cryptography*. In: *Preproceedings of the Workshop on Coding and Cryptography (WCC 13)*. Bergen, S. 168–180. URL: http://www.selmer.uib.no/WCC2013/pdfs/Gaborit.pdf (besucht am 09. 11. 2018).

Galvez, Lucky u. a. (2017). *McNie: Compact McEliece-Niederreiter Cryptosystem*. Dateipfad innerhalb der ZIP-Datei: McNie/Supporting_Documentation/document.pdf. URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/McNie.zip (besucht am 26. 09. 2018).

Homeister, Matthias (2015). *Quantum Computing verstehen*. Hrsg. von Wolfgang Bibel, Rudolf Kruse und Bernhard Nebel. 4. Aufl. Wiesbaden: Springer Vieweg. DOI: 10.1007/978-3-658-10455-9.

Intel Corp., Hrsg. (2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, M-U*. Sep. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf (besucht am 03. 04. 2019).

Knežević, Miroslav, Vladimir Rožić und Ingrid Verbauwhede (2009). *Design methods for embedded security*. In: *Telfor Journal*, Jg. 1, Nr. 2, S. 69–72. URL: https://www.esat.kuleuven.be/cosic/publications/article-1190.pdf (besucht am 26. 09. 2018).

Kocher, Paul (1996). *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. In: *Advances in Cryptology — CRYPTO '96*. Hrsg. von Neal Koblitz. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, S. 104–113. DOI: 10.1007/3-540-68697-5_9.

Kocher, Paul u. a. (2019). *Spectre Attacks: Exploiting Speculative Execution*. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. URL: https://www.computer.org/csdl/proceedings/sp/2019/6660/00/666000a019.pdf (besucht am 26. 09. 2018).

Lipp, Moritz u. a. (2018). *Meltdown: Reading Kernel Memory from User Space*. In: *27th USENIX Security Symposium (USENIX Security 18)*. URL: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf (besucht am 26. 09. 2018).

Lütkebohmert, Werner (2003). *Codierungstheorie*. Wiesbaden: Vieweg+Teubner Verlag. DOI: 10.1007/978-3-322-80233-0.

McEliece, Robert J. (1978). *A Public-Key Cryptosystem Based On Algebraic Coding Theory*. In: *Deep Space Network Progress Report*, Jg. 44 (1. Jan. 1978), S. 114–116. URL: https://tmo.jpl.nasa.gov/progress_report2/42-44/44N.PDF (besucht am 03. 12. 2018).

Menezes, Alfred, Paul van Oorschot und Scott Vanstone (2001). *Handbook of Applied Cryptography*. 5. Aufl. CRC Press. URL: http://cacr.uwaterloo.ca/hac/ (besucht am 26. 09. 2018).

Niederreiter, Harald (1986). *Knapsack-type cryptosystems and algebraic coding theory*. In: *Problems of Control and Information Theory*, Jg. 15, Nr. 2, S. 159–166. URL: http://real-j.mtak.hu/7997/1/MTA_ProblemsOfControl_15.pdf (besucht am 03. 12. 2018).

Teschl, Gerald und Susanne Teschl (2014). *Mathematik für Informatiker*. Bd. 2. eXamen.press. Berlin, Heidelberg: Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-54274-9.

Welschenbach, Michael (2013). *Cryptography in C and C++*. Berkeley, CA: Apress. DOI: 10.1007/978-1-4302-5099-9.

Wilhelm, Frank K. u. a. (2018). *Entwicklungsstand Quantencomputer*. Hrsg. von Bundesamt für Sicherheit in der Informationstechnik. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Quantencomputer/P283_QC_Studie.pdf?__blob=publicationFile (besucht am 26. 09. 2018).

Witt, Kurt-Ulrich (2013). *Lineare Algebra für die Informatik*. Wiesbaden: Springer Vieweg. DOI: 10.1007/978-3-658-00189-6.

Yazbek, Abdul Karim u. a. (2017). *Low Rank Parity Check Codes and their application in Power Line Communications smart grid networks*. In: *International Journal of Communication Systems*, Jg. 30, Nr. 12 (1. Aug. 2017). DOI: 10.1002/dac.3256.