UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR IT SECURITY

Fault Injection: An Attack on the post quantum KEM BIKE

Fault Injection: Ein Angriff auf den postquanten KEM BIKE

Bachelor Thesis

within the study courses of **computer science security**
at the University of Lübeck

handed in by **Till Jirka Eifert**

issued and supervised by **Dr. Sebastian Berndt**

with the assistance of M.Sc. Alexander Treff

Lübeck, February 21, 2023

Abstract

As research on quantum computers succeeds, it seems to be only a matter of time until they are powerful enough to break today's public key cryptographic systems based on integer factorization and discrete logarithm. To face this vital security issue, the *National Institute of Standards and Technology* (NIST) initiated a standardization process to provide quantum resistant public key cryptographic systems. After selecting one algorithm for key establishment and three algorithms for digital signatures as standard in its third round, this process is currently in its fourth round to determine another key establishment algorithm. One of the currently three candidates (one is already considered insecure) is the key encapsulation mechanism BIKE.

The recent work from [Ket22] describes a key recovery attack against BIKE and assumes the attack can be fostered by fault injection. In this thesis we investigated the feasibility of [Ket22]'s attack by injecting faults on a physical target. As hardware we chose the ChipWhisperer-Lite ARM and as software implementation of BIKE we chose the pqm4 project. The ChipWhisperer environment provides an elaborated api to inject faults in various ways.

During the preparations of fault injection, we discovered that we can distinguish several *fault modes* for faulting the secret keys. These *fault modes* result from faulting different parts of the secret key in various ways. While the key recovery attack from [Ket22] depends on an increased decoding failure rate (DFR) during the decapsulation of BIKE, we could observe that different *fault modes* cause the DFR to behave differently.

As [Ket22] suggests to fault two subroutines, key generation and decapsulation, we were able to successfully fault both. Hence, we can state that the key recovery attack via fault injection is possible.

Da die Forschung zu Quanten-Computern voranschreitet, scheint es nur eine Frage der Zeit zu sein, bis sie mächtig genug sind um heutige kryptographische Systeme zu brechen die auf der Faktorisierung von natürlichen Zahlen oder dem diskreten Logarithmus beruhen. Um diesem essentiellen Sicherheitsbedenken zu begegnen hat das *National Institute of Standards and Technology* (NIST) einen Prozess zur Standardisierung von quantensicheren asymmetrischen kryptographischen Systemen initiiert. Nachdem in der dritten Runde bereits ein Schlüsselaustausch- und drei Signatur-Algorithmen festgelegt wurden, ist der Prozess in der vierten Runde um einen weiteren Schlüsselaustausch-Algorithmus zu standardisieren. Einer der drei verbleibenden Kandidaten (einer wird bereits als unsicher erachtet) ist der Key Encapsulation Mechanism BIKE.
In der jüngsten Arbeit von [Ket22] wird ein Key Recovery Angriff beschriben und vermutet, dass dieser durch Fault Injection ermöglicht werden kann. In dieser Abschlussarbeit untersuchen wir die Durchführbarkeit des Angriffes von [Ket22] durch Fault Injection auf einem physischen Angriffsziel. Als Hardware wählten wir den ChipWhisperer-Lite ARM und als Software Implementierung von BIKE das pqm4 Projekt. Die ChipWhisperer Umgebung stellt eine ausgefeilte API zur Verfügung um auf verschiedene Weisen Faults zu injizieren.
Während der Vorbereitungen für die Fault Injection, ist uns aufgefallen, dass wir bei der Manipulation des geheimen Schlüssels mehrere *Fault Modes* unterscheiden können. Diese *Fault Modes* resultieren daraus, dass der geheime Schlüssel an verschiedenen Stellen auf unterschiedliche Arten verändert werden kann. Bei dem Key Recovery Angriff von [Ket22] wird eine erhöhte Decoding Failure Rate (DFR) während der decapsulation von BIKE vorausgesetzt. Wir konnten beobachten, dass unterschiedliche *Fault Modes* dazu führen, dass sich die DFR unterschiedlich verhält.
Wohingegen [Ket22] vorschlägt zwei Sub-Rutinen zu maipulieren, die Schlüsselerstellung und die Decapsulation, war es uns möglich beide erfolgreich anzugreifen. Folglich können wir festhalten, dass der Key Recovery Angriff durch Fault Injection möglich ist.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, February 21, 2023

Contents

1 Introduction

Nowadays research on quantum computers is focused by a large community of scientists and companies. While quantum computers are growing larger and more powerful, a lot of advantages arise from that development. At the same time a major security issue has to be stated: Most of the encryption we use to secure communication and data world wide is not quantum secure. The mathematical problems (most of) our today encryption relies upon are integer factorization and discrete logarithm. These problems can be solved in a post quantum world (when there are quantum computers with sufficient power) with ease.

In advance to address this security issue, in 2016 the U.S. American *National Institute of Standards and Technology* (NIST) initiated a process to standardize quantum resistent public key algorithms for both, key establishment and digital signatures. While round 3 of the standardization process already ended with the lattice based *CRYSTALS-KYBER* as key establishment algorithm and three signature schemes to be standardized [AAC+22], NIST started a fourth round. This fourth round aims to provide another standard algorithm for key establishment. One of the fourth round candidates is the code based BIKE algorithm [NIS].

The authors of BIKE address the key recovery attack introduced by [GJS16] by specification. Nevertheless, [Ket22] suggests using fault injection to enable an attack based on the work of [GJS16] against BIKE. While [Ket22]'s approach is based on a more theoretical aspect, this thesis aims to research the feasibility of faulting the algorithm on actual hardware.

[Ket22] suggests to fault either the key generation to force *faulty keys* to be generated or the threshold, which is an important part of decapsulating the ciphertext with the secret key. Besides the overall objective, we aim to provide a workable setup for fault injections against BIKE. As matter of injecting faults we use the ChipWhisperer-Lite ARM [Newa] board as hardware and the pqm4 project [KRSS] as software.

In Chapter 2, we give an introduction to BIKE. There will be a general overview of its composition, highlighting the subroutines which are supposed to be faulted and the properties, that have to hold for BIKE's parameters. This is the foundation to explain the key recovery attack introduced by [Ket22] in Chapter 3. As fault injection is the means to accomplish [Ket22]'s key recovery attack, we provide a definition of fault injection in Chapter 4. The hardware we work with and its setup is described in Chapter 5. To run BIKE

on the ChipWhisperer-Lite ARM, we had to heavily adapt the implementation and add lower security levels to BIKE as described in Chapter 6. With all the preparations from the previous chapters, in Chapter 7 we analyze the impact of different *fault modes*, which result from altering different parts of the secret keys in various ways. We discuss the spots in the algorithm, which seem to be vulnerable to fault injection and finally, we present the settings that allowed us to successfully fault BIKE and proof the feasibility of [Ket22]'s key recovery attack.

2 BIKE - Bit Flipping Key Encapsulation

In this chapter we will introduce the subject of this thesis, the key encapsulation mechanism BIKE. We will start with a brief introduction of key encapsulation mechanisms (KEM), then recap the notation used by the algorithm's authors. This is the foundation to understand the specification of BIKE. Afterwards, we can have a brief look at the security of this KEM and thereby understand the parameters needed to define security levels. Finally we introduce the parameter sets we were working with and how we calculated them.

2.1 Key Encapsulation Mechanism

Similar to a key exchange algorithm (KEX) a KEM is designed to establish a secure channel. In both cases this is achieved by negotiating a symmetric key or shared secret via asymmetric cryptography. For both, we can assume one party has a pair of public and secret keys. In KEX the symmetric key K itself is encrypted and exchanged, while for KEM only a message m is encapsulated and transmitted and the actual symmetric key K is derived from that.
Algorithms 1 and 2 show a rough draft of both, KEX and KEM. For both we assume a Public Key Encryption System (PKES) given. Alice has a valid key pair (pk_{Alice}, sk_{Alice}) for this PKES and Bob has Alice's public key. The functions $enc()$, $dec()$ refer to either encryption/decryption (KEX) or encapsulation/decapsulation (KEM) of the PKES. For KEM the key derivation $kd()$ usually is part of encapsulation/decapsulation, but highlighted here to emphasize the difference between m and K as well as between KEX and KEM.

Alice		Bob
		$c = enc(pk_{Alice}, k)$
	$\xleftarrow{c}$	
$K = dec(sk_{Alice}, c)$		

Algorithm 1: Key Exchange Algorithm

Alice		Bob
		$c = enc(pk_{Alice}, m)$
	$\xleftarrow{c}$	
$m = dec(sk_{Alice}, c)$		
$K = kd(m)$		$k = kd(m)$

Algorithm 2: Key Encapsulation Mechanism: $kd()$ is a key derivation function

A KEM is composed of three algorithms:

1. One is *keypair* to *generate* a pair of public and secret keys (pk, sk).
2. The *encapsulation* which takes a public key pk to generate a symmetric key K and a ciphertext c. The ciphertext c encapsulates the message m, from which the shared secret K can be derived.
3. And *decapsulation* which takes the secret decapsulation key sk and the encapsulated message c to output the shared secret K or a decapsulation failure symbol $\perp$.

2.2 Specification

To specify the algorithm it is important to know what we are talking about. For consistency we stick to the notation introduced by the BIKE Specification [AGB+21]. We refer to the specification version 4.1, the reasons for this decision are discussed in Section 6.4.1

2.2.1 Notation and Definitions

The general mathematical notation is shown in Table 2.1. In Definition 2.1 essential parameters are listed and the properties they have to fullfil for security reasons are named.

Definition 2.1 (Parameters). Parameters defining an instance of BIKE.

- Block size r has to be prime, 2 has to be a generator of $\mathbb{F}_r$ to avoid any undesirable structure in the polynomial ring $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$. The polynomial $(X^r - 1)/(X - 1) \in \mathbb{F}_2[X]$ has to be irreducible. To speedup inversion, $|bin(r - 2)|$ should be small.
- Code length $n = 2 \cdot r$.
- Row weight $w \approx \sqrt{n}$ has to be even.
- Block weight $d = \frac{w}{2}$ has to be odd to ensure that h_0 is always invertible in $\mathcal{R}$.
- Error weight t has to be smaller than $\sqrt{2 \cdot r}$.

- Shared secret size l.

Definition 2.2 (Hash Functions). The functions **H**, **K**, **L** have to selected uniformly at random from the set of functions with the following respective domains and ranges.

- **H** : $\mathcal{M} \to \mathcal{E}_t$.
- **K** : $\mathcal{M} \times \mathcal{R} \times \mathcal{M} \to \mathcal{K}$.
- **L** : $\mathcal{R}^2 \to \mathcal{M}$.

The functions are modeled as random oracles. A concrete instantiation of {**H**, **K**, **L**} needs to be associated with the scheme.

For BIKE, $\mathcal{M} = \mathcal{K} = \{0,1\}^l$.

$\mathbb{F}_2$:	Binary finite field.
$\mathcal{R}$:	The polynomial ring $\mathbb{F}_2[X]/\left(X^r - 1\right)$.
$\mathcal{H}_w$:	Private key space $\{(h_0, h_1) \in \mathcal{R}^2 : \ \lvert h_0\rvert = \lvert h_1\rvert = w/2\}$
$\mathcal{E}_t$:	Error space $\{(e_0, e_1) \in \mathcal{R}^2 : \ \lvert e_0\rvert + \lvert e_1\rvert = t\}$
$\lvert v\rvert$:	The Hamming weight of a binary polynomial $v \in \mathcal{R}$.
$x \xleftarrow{\$} X$:	Variable x is sampled uniformly at random from the set X.
$\oplus$:	exclusive or of two bits, componentwise with vectors

Table 2.1: BIKE Notation

Definition 2.3 (QC-MDPC Code). A Quasi-Cyclic Moderate Density Parity Check code of index 2, i.e. binary, length n, and row weight w is defined as a pair of sparse parity polynomials $(h_0, h_1) \in \mathcal{H}_w$.

Definition 2.4 (Decoder). Takes as input a syndrome $s = e_0h_0 + e_1h_1 \in \mathcal{R}$ and parity polynomials $(h_0, h_1) \in \mathcal{H}_w$ and outputs a sparse vector $(e_0, e_1) \in \mathcal{R}^2$. The decoder guarantees that for $\Big((e_0, e_1) \in \mathcal{R}^2$ and $\lvert e_0\rvert + \lvert e_1\rvert \leq t\Big)$, we have $\Rightarrow (e_0, e_1) = decoder(e_0h_0 + e_1h_1, (h_0, h_1))$

BIKE is a code-based KEM, based on QC-MDPC (Quasi-Cyclic Moderate Density Parity-Check) codes [Mea]. Such a code is defined by the secret key (h_0, h_1). To keep it simple, one could say these codes allow error correction whlie they are looking sufficently random. Given the secret key and a sufficient decoder, if the error weight t is low enough we can decode the ciphertext c back to the error vectors (e_0, e_1). This properties allow us to use such codes as KEM. As encapsulation we can obfusecate the message m with the public key and some random error vectors to get the ciphertext c. While decapsulation

is reconstructing the message m from the ciphertext c with the secret key to obtain the shared secret K. The exact steps are shown below, in Algorithm 3. [Ket22] gives a deep dive into the mathematical background, therfore we will skip that in this thesis.

2.2.2 Decoder and Threshold

It becomes aparent that the decoder is a key part of the BIKE algorithm, because of this we have to take a closer look at the decoder. The security of the BIKE levels is directly linked to the Decoding Failure Rate (DFR). E.g. to achieve a 128 bit security the DFR has to be smaller than 2^{-128} [DGKP20].

Definition 2.5 (Decoding Failure Rate). With $((h_0, h_1), (e_0, e_1)) \xleftarrow{\$} \mathcal{H}_w \times \mathcal{E}_t$, the Decoding Failure Rate of a decoder is defined as

$$DFR(decoder) = Pr[(e_0, e_1) \neq decoder(e_0 h_0 + e_1 h_1, h_0, h_1)].$$

One key component for the decoder is the threshold function, because the decoder's decision to flip bits depends on the threshold. The function depends on the weight of the syndrome $|s|$ and the iteration number i, so it can be denoted as $th(|s|, i)$, but an approximation only depending on the syndom weight is sufficient. The calculated threshold indicates whether the decoder has to flip a bit or if it can be left unchanged. Definition 2.7 shows the approximated threshold, while Definition 2.8 gives the accurate threshold. Because the calculation of the correct threshold is quite expensive, the implementations use the approximation.

2.2.3 Algorithm Overview

Algorithm 3 gives an overview of the KEM elements of BIKE. They will be used as described in Algorithm 2.

Definition 2.6 (BIKE Keys). The secret key is composed of the QC-MDPC code $(h_0, h_1) \in \mathcal{H}_w$ and $\sigma \in \mathcal{M}$ which serves as failure message if the decoder fails. The public key $h \in \mathcal{R}$ is derived from the secret key: $h = h_1 h_0^{-1}$. The shared secret $K \in \mathcal{K}$ is the result of the key agreement and functions as symmetric key.

BIKE is primarily designed to be used with ephemeral keys for synchronous communication protocols, like TLS. In this case forward, secrecy is achieved. It can as well be adapted to asynchronous protocols like email, but does no longer provide forward secrecy as key reuse is needed in this case.

KeyGen: $() \to (h_0, h_1, \sigma), h$ Output: $(h_0, h_1, \sigma) \in \mathcal{H}_w \times \mathcal{M}, h \in \mathcal{R}$ 1: $(h_0, h_1) \xleftarrow{\$} \mathcal{H}_w$ 2: $h \leftarrow h_1 h_0^{-1}$ 3: $\sigma \xleftarrow{\$} \mathcal{M}$	**Encaps**: $h \to K, c$ Input: $h \in \mathcal{R}$ Output: $K \in \mathcal{K}, c \in \mathcal{R} \times \mathcal{M}$ 1: $m \xleftarrow{\$} \mathcal{M}$ 2: $(e_0, e_1) \leftarrow \mathbf{H}(m)$ 3: $c \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$ 4: $K \leftarrow \mathbf{K}(m, c)$
Decaps: $(h_0, h_1, \sigma), c \rightarrowtail K$ Input: $((h_0, h_1), \sigma) \in \mathcal{H}_w \times \mathcal{M}, c = (c_0, c_1) \in \mathcal{R} \times \mathcal{M}$ Output: $K \in \mathcal{K}$ 1: $e' \leftarrow \texttt{decoder}(c_0 h_0, h_0, h_1)$ $\quad \triangleright\ e' \in \mathcal{R}^2 \cup \perp$ 2: $m' \leftarrow c_1 \oplus \mathbf{L}(e')$ $\quad \triangleright$ with the convention $\perp = (0, 0)$ 3: **if** $e' = \mathbf{H}(m')$ **then** $K \leftarrow \mathbf{K}(m', c)$ **else** $K \leftarrow \mathbf{K}(\sigma, c)$	

Algorithm 3: The BIKE Key Encapsulation Mechanism [AGB+21]

2.3 Security

In the BIKE specification [AGB+21] two claims are stated about the security:

- BIKE is proven IND-CPA secure under Assumptions 1 and 2.
- BIKE is proven IND-CCA secure under Assumptions 1, 2 and 3.

Assumption 1 (Quasi Cyclic Syndrome Decoding). Hardness of $\text{QCSD}_{r,t}$

Assumption 2 (Quasi Cyclic Codeword Finding). Hardness of $\text{QCCF}_{r,w}$

Assumption 3 (`decoder`). Correctness of `decoder`.

[Ket22] goes into detail about how IND-CCA security is achieved and the meaning of it. For this thesis it is enough to know, that IND-CCA (indistinguishability of a chosen ciphertext attack) is one of the best properties a cryptographic scheme can achieve.
[AGB+21] state that Assumptions 1 and 2 relate to standard hard problems of coding theory and that the proof framework for assumption 3 is also valid in the QROM (quantum random oracle model). This means, that the algorithm is believed to be secure against a quantum adversary.
We won't further discuss this here. For the mathematical background see [Ket22].

2.3.1 Parameters

To define a security level for BIKE, the four parameters r, w, t, l are needed. In practice the shared secret length l is always at 256, so to define a level it takes a triple of (r, w, t). The meaning and the properties that have to hold for these parameters can be found in Definition 2.1.
For the implementation three more parameters are needed. They determine the threshold function and can be derived from the previous ones.

Definition 2.7 (Threshold Function). As approximation the threshold function used is independent of the iteration number and reduced to a linear function:

$$th(|s|) = \max(\lfloor th_1 \cdot |s| + th_0 \rfloor, th_{min}).$$

The threshold coefficients are th_0 and th_1, while th_{min} is the threshold minimum, $th_{min} = \frac{d+1}{2} = \frac{(w/2)+1}{2}$.

[Ket22] extracted the threshold function from [Vas21] and defined it as follows.

Definition 2.8 (Accurate Threshold Function).

$$th_{acc}(\|s\|_1) = \max\left(\frac{d+1}{2}, \left\lceil \frac{d \log \frac{1-\pi_1}{1-\pi_0} + \log \frac{\|e\|_1}{n-\|e\|_1}}{log\frac{1-\pi_1}{1-\pi_0} + \log \frac{\pi_0}{\pi_1}} \right\rceil\right) \tag{2.1}$$

with

$$\pi_0 = \frac{(w-1)\|s\|_1 - X}{d(n-\|e\|_1)} \qquad \pi_1 = \frac{\|s\|_1 + X}{d\|e\|_1}$$

$$X = \|s\|_1 \frac{\sum_l 2l\rho_{2l+1}}{\sum_l \rho_{2l+1}} \qquad \rho_l = \frac{\binom{w}{l}\binom{n-w}{t-l}}{\binom{n}{t}}$$

So the approximation is generated from a linear function through $th_{acc}(1)$ and $th_{acc}(|s|)$ for the highest value for $|s| = S_{max}$ which causes $th_{acc}(|s|)$ to be real. This maximum is reached when $th_{acc}(|s|) = t$. Note that $S_{max} < r$.

Parameter Selection

According to [AGB+21], to reach λ bits of IND-CCA security, the parameters have to be chosen such that:

1. $\text{QCCF}_{r,w}$ offers λ bits of security
2. $\text{QCSD}_{r,t}$ offers λ bits of security
3. $|\mathcal{M}| = 2^l \geq 2^{-\lambda}$.

- Choosing l is straightforward, and actually $l = 256$ for all levels.
- w and t depend on the computational problems.
- With w and t fixed, the block length r is selected. This is done by simulation and extrapolation, so that the DFR is low enough.

Parameter Calculation

For hardware reasons, as shown in Chapter 6, we had to define additional parameter sets. Contrary to the recommended way of finding parameter sets, we started with a r which fulfilled all requirements mentioned in Definition 2.1 and calculated the other parameters accordingly in an approximating way.

- $t_{max} = \lfloor \sqrt{2 \cdot r} \rfloor$ as a temporary value.
- $d = \lfloor \frac{t_{max}}{2} - 10 \rfloor$ the block weight from which the row weight w is derived.
- $t = \lceil t_{max} \cdot 0.85 \rceil$ the error weight.
- $th_{min} = \frac{d+1}{2}$ the threshold minimum.
- For the threshold coefficients $th_0,\ th_1$ [Ket22] implemented the accurate threshold th_{acc} and an extrapolation of the approximation. With this implementation we can get correct coefficients. Though it does not perfectly fit the approximations the BIKE team defined for their levels, it is very close and considered sufficient.

An implementation can be found in BIKE_params.sage. It can be preparsed to a python script, which can easily be used by other python scripts.

2.3.2 Security Levels

The submitted security levels claim to achieve respectively the security levels 128, 192 and 256 bits demanded by NIST. The parameter sets in Table 2.2 were used.

Security	r	w	t	th_0	th_1	DFR
Level 1	12 323	142	134	13.530	0.0069722	2^{-128}
Level 3	24 659	206	199	15.2588	0.005265	2^{-192}
Level 5	40 973	274	264	17.8785	0.00402312	2^{-256}

Table 2.2: Suggested BIKE Parameters [AGB+21]

Because the hardware we were working with could not run any of those levels (too low memory), we defined several new levels. Here, we just mention two of them, the smallest we defined and largest we found running stable on the hardware, see Table 2.3. All the levels we defined can be found in *bike_def.h*, e.g., *pqm4/mupq/crypto_kem/bikel1/opt/bike_defs.h*.

Security	r	w	t	th_0	th_1
Level 11	773	18	34	3.9828767	0.017123
Level 17	4 021	70	76	8.98932536	0.0106746

Table 2.3: additional security levels

3 Key Recovery Attack with Faulty Keys

This chapter introduces a key recovery attack against BIKE described in [Ket22]. First we will introduce the underlying key recovery attack introduced in [GJS16]. Afterwards, we explain the adaptions taken by [Ket22], their effects and why this is a major improvement.

3.1 Key Recovery Attack

[GJS16] introduced the first key recovery attack on QC-MDPC, it aims to find the secret matrix H_0, given only the public key matrix H. Because the codes are quasi cyclic the matrices can be represented by the first lines, $h_0 \hat{=} H_0$ and $h \hat{=} H$. If we know h_0 we can easily compute $h_1 = hh_0 = h_1 h_0^{-1} h_0$, compare Definition 2.6.
The attack model here is, that we have Bob's public key and can send arbitrarily chosen encapsulated error vectors (e_0, e_1). Bob will decapsulate them and we can observe different behavior if the decapsulation was successful or not. So we do not need to know the exact output of the decapsulation, but it is enough to get an answer to the question, whether the decapsulation was successful.
The attack is based on recovering the distance spectrum. To achieve this we choose special error vectors.

Definition 3.1 (Error Subspace). Let Ψ_d be a subset of $\mathcal{E}_t$, where all the t ones are set in e_0 as random pairs with distance d, thus $e_1 \in \{0\}^r$ and

$$\Psi_d = \{v = (e, f) : |f| = 0 \wedge \forall i < r \text{ with } e_i = 1 : e_{i+d \mod r} = 1 \vee e_{i-d \mod r} = 1\}.$$

It follows, that $e \in \Psi_d$ has at least $\lfloor t/2 \rfloor$ occurrences of set bits with a distance of d to each other.
[GJS16] observed that the DFR is *much smaller* for an $e \in \Psi_d$, if there are two ones with distance d in h_0. So we can distinguish if the distance d occurs between two ones in h_0 by determining the DFR for $e \in \Psi_d$. Repeating this for $d \in \{1, 2, \ldots, u\}$ for a suitable upper bound u reveals the distance spectrum of h_0, denoted $D(h_0)$.

Definition 3.2 (Distance Spectrum). The distance spectrum of a vector h_0 is a set of numbers d_i. Each number d indicates that there is a pair of bits set to 1 with a distance of d

between those bits.

$$D(h_0) = \{d : 1 \leq d \leq u, \text{ at least one pair of ones with distance } d \text{ occur in } h_0\}$$

Finally the reconstruction of h_0 happens in an iterative procedure. Let's call the reconstructed vector h'. First we set h'_0, the next bit is set at i_0 where i_0 is the smallest value in $D(h_0)$. The third bit is set at the next possible position and we check if the distances to the previous bits occurs in $D(h_0)$. This will be repeated until h' has the desired weight. For a more detailed explanation of the reconstruction algorithm, see [GJS16].
This algorithm may return a h' which is a rotation of h_0, but this is considered negligible, because we can try all $2r$ rotations. If h' differs from h (no rotation), but $D(h') = D(h_0)$, the algorithm has to discard h' and recursively has to try other key patterns.

3.2 Key Recovery Attack with Faulty Keys against BIKE

While the previously described attack focuses on QC-MDPC in general and profits from a rather high DFR, for BIKE the decoder is chosen such that the DFR fits the overall security level. There are other papers, like [SV20], analyzing weak keys for BIKE. These weak keys mainly increase the DFR. Nevertheless those weak keys occur seldom enough to not affect the overall security of BIKE.
One attempt could be to fault the key pairing to generate weak keys, but [Ket22] found even more effective keys defined as *faulty keys*. [Ket22] distinguishes type one and type two faulty keys. For type two faulty keys we found a further distinction. Type two.a when h_0 is faulted and type two.b when h_1 is faulted. This makes sense since the block weight d has to be odd to guarantee h_1 is invertible (to calculate the public key).

Definition 3.3 (Type one faulty key). Let the block-circulant matrix $H = (h_0, h_1)$ be a valid private key of a BIKE implementation with parameters r, d, t and a threshold function $th(r, d, t, s)$, where s is the current syndrome.
Then a private key H' with parameters r, $d' \neq d$, t and $th(r, d, t, s)$ is called *type one faulty key*.

Definition 3.4 (Type two faulty key). Let the block-circulant matrix $H = (h_0, h_1)$ be the parity check matrix of a QC-MDPC Code. The weight of each of the circulant matrices is denoted by d. A block-circulant matrix $H' = (h'_0, h'_1)$ is called *type two faulty key*, if either $h'_0 = h_0$ and $|h'_1| \neq |h_1|$ or $h'_1 = h_1$ and $|h'_0| \neq |h_0|$.
To be more precise, a Block-Circulant matrix $H' = (h'_0, h'_1)$ is called *type two.a faulty key*, if $h'_1 = h_1$ and $|h'_0| \neq |h_0|$, and a Block-Circulant matrix $H' = (h'_0, h'_1)$ is called *type two.b faulty key*, if $h'_0 = h_0$ and $|h'_1| \neq |h_1|$.

One can see faulty keys are not necessarily valid according to the specification, but they are accepted by the Reference Implementation and can be used for correct key decapsulation except for some decoding errors. Therefore we expect that faulty keys will also be accepted by other BIKE implementations [Ket22].

3.2.1 Effect of Faulty Keys on the DFR

Besides the fact that we use faulty keys, the attack with faulty keys is exactly the same as described in Section 3.1. The effect of faulty keys is to push the DFR above the specified range. This happens mainly because the threshold function $th(|s|)$ depends on the row weight w of (h_0, h_1). So if we use faulty keys the decoders threshold function does not fit the key weight any more and thus the DFR rises.
The advantage of this attack is that we do not rely on the decoder's DFR by design, but choose keys to provoke a desired DFR. By statistically analyzing the DFR for different faulty keys and different weights we can precisely aim a specific DFR.
Another attempt is to fault the threshold directly at its computation. [Ket22] shows that this has a similar effect.

4 Fault Injection

To explain fault injection we first define some terms [Ber21].

Definition 4.1 (Fault). A Fault can cause an *error*. Active if error already occurred, otherwise dormant.

Definition 4.2 (Error). An error is a state of a subsystem that does not follow specification.

Definition 4.3 (Failure). A failure is an incident in which the functionality of a subsystem is needed but can not be fulfilled.

Processors always need specific circumstances to work the way they are supposed to. These circumstances can for example be temperature, humidity, voltage range, available current, just to name a few. If those leave the processor's specification range, we call it a *fault*. Often faults are not recognized because they do not cause an *error* or even a *failure*.
Faults can be the effect of a variety of factors, e.g., electrical shorts, voltage glitches, cosmic radiation, electro magnetic impulses, to name some physical factors.
If faults lead to errors, e.g., a variable with a wrong value, a failure can occur. This could cause the whole system to crash, return a wrong value in a calculation or, if we think about a login, permit access with the wrong password, which is a failure. To speak of it in a more general way: errors can be used to change the control flow. To stick with the password example see Figure 4.1, the decision which branch to take (in general) depends on a single processor instruction. So if this instruction is faulted and an error occurs in some way one can gain access with the wrong password. From this example we can derive the following definition:

Definition 4.4 (Fault Injection). A *Fault Injection* provokes faults within a processor with the aim to trigger failures which manipulate the processor's behavior in an unspecified but intended way.

In most use cases these faults occur very seldom, because devices are run within their specification range. But if we use fault injection we can target a system to reveal information that is meant to stay secret, by exploiting the errors we cause.

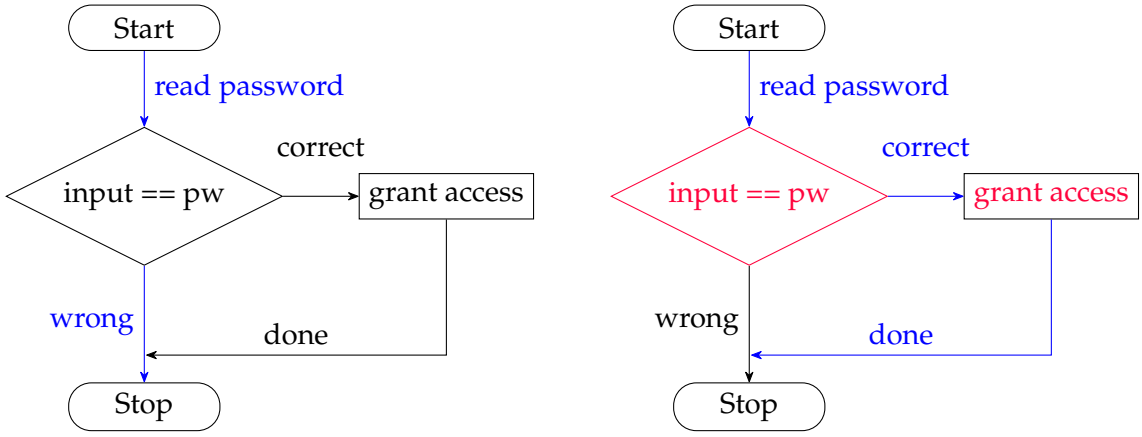

Figure 4.1: Example Program Control Flow: both with incorrect password, left without fault, right successfully faulted. Path taken is marked in this color.

4.1 Categories of fault injection

To classify fault injections, the following criteria can be used [Bal15]

- Non-invasive: the advantages are that we do not have to physically damage the device and in most cases we can reuse it as often as we want to. The equipment needed is less expensive compared to other methods. Working conditions can externally be modified, like controlling the power supply. There is no special knowledge needed, like where the inner circuits are placed inside the chip. The information from the chip's manual, e.g., pin outs, are sufficient.

- Semi-invasive: this already includes chip decapsulation, i.e., removing the cover of a chip to reveal its inner structure. For this usually milling, etching and cleaning are required. Besides the need of specialized equipment the risk of destroying the chip is inherent, so the more expensive a chip is the more this has to be considered. On the other hand the information gained from this will increase precision and can proliferate the research outcomes.

- Invasive: this is taking chip decapsulation on the next level. In contrast to a semi-invasive attack an electrical contact to the chip is established. So this can be used to change the gates, e.g., modify or destroy the chip layout. The risk of destruction is increased and the equipment needed is even more specialized. In general, a human being is not accurate enough to manipulate gates and transistors on a chip. But this can achieve the highest accuracy and manipulate the chip in any desireable way.

4.2 Fault Model

To compare properties of different fault injection attacks, they can be classified with respect to criteria like granularity, modification, control and duration [Bal15, EGVL+19].

Granularity precision

- Single bit: most accurate, a single bit can be influenced. Can also be limiting if one bit is not enough.
- Word: usually achieved by manipulating one register at a time, the whole register can be set to a specific value or at least be manipulated to alter the value it holds.
- Few bits: in between the others.

Modification fault type

- Stuck-at: a bit (sequence) always turns to 1, stuck-at-1, or turns to 0, stuck-at-0.
- Flip: the affected bits can be flipped.
- Random: the affected bits will hold random values.
- Selective: the affected bits can be set on one's behalf. The most powerful.

Control location *and* time

- Precise: the fault can be set at a very specific point.
- Loose: the fault happens more or less in the same area.
- None: there is no control when we fault.
- Reliable: is it possible to produce the same fault repeatedly.

Duration effect of fault

- Transient: the fault takes effect only once (or for a short period of time).
- Permanent: the effect is constantly present.
- Destructive: the chip (or parts of its functionality) is destroyed.

4.3 Self Classification

After introducing these different properties, we relate our attack to them to give a prospect of the further work.
The fault injection attack we implement is non-invasive, we use voltage and clock glitching. The granularity we achieve may vary as we try to set single bits in the secret key, but this can as well be achieved by manipulating whole words. Modification with such a rough method as clock and voltage glitching is in most cases random, like in our case. Through the ChipWhisperer environment we are able to inject faults very precisely and reliably. The effects inside the chip will be transient because we only fault the key generation and the decapsulation. The generated keys might be used for multiple key exchanges, but from the chips perspective the fault stays transient, as the keys will be lost after reboot, because they are only stored in RAM.

Category	non-invasive
Granularity	word - single bit
Modification	Random
Control	precise
Duration	transient

Table 4.1: Attack Classification

5 Environment

As environment for this research, we focus on the CW1173, also known as ChipWhisperer-Lite. Here, we will give an extended introduction to CW1173 and touch on the discovery board stm32f407, as we worked with this one, too. Both targets implement the ARM Cortex-M4 architecture. There is a widespread support from NewAE, the developers of ChipWhisperer, including target firmware as well as host communication and control libraries. We will have a look at these and mention the relevant settings for injecting faults, as far as they are firmware independent.

5.1 Hardware

There are two pieces of hardware we were working with, the ChipWhisperer-Lite ARM and the discovery board stm32f407. The former is designed to inject faults, so it is an ideal tool for this research. It is composed of two main components the capture and the target board. The latter is only a target board, compared to the ChipWhisperer-Lite ARM.

5.1.1 ChipWhisperer

NewAE provide a wide range of boards for side channel analysis and fault injection. One of their products is the CW-Lite ARM [Newa] from the ChipWhisperer series. This board is designed for fault injection and capturing power traces at the same time. Besides the attack infrastructure, it holds a target board which can be programmed on one's behalf. The capture and the target board can be separated from each other, so other targets could be connected with the capture part, as well as other capture and/or glitch equipment could be connected to the target. All of NewAE's work is open source including hardware, software, firmware and FPGA code.

Capture Board

The CW1173 [Inc] contains a bunch of microcontrollers. Two of them run their own firmware and make use of the others. The first one to mention is the *Atmel SAM3U* which handles the USB interface and allows to program the *SPARTAN-6 FPGA*. Among other things, the latter handles the glitch and capture hardware. One major advantage of this

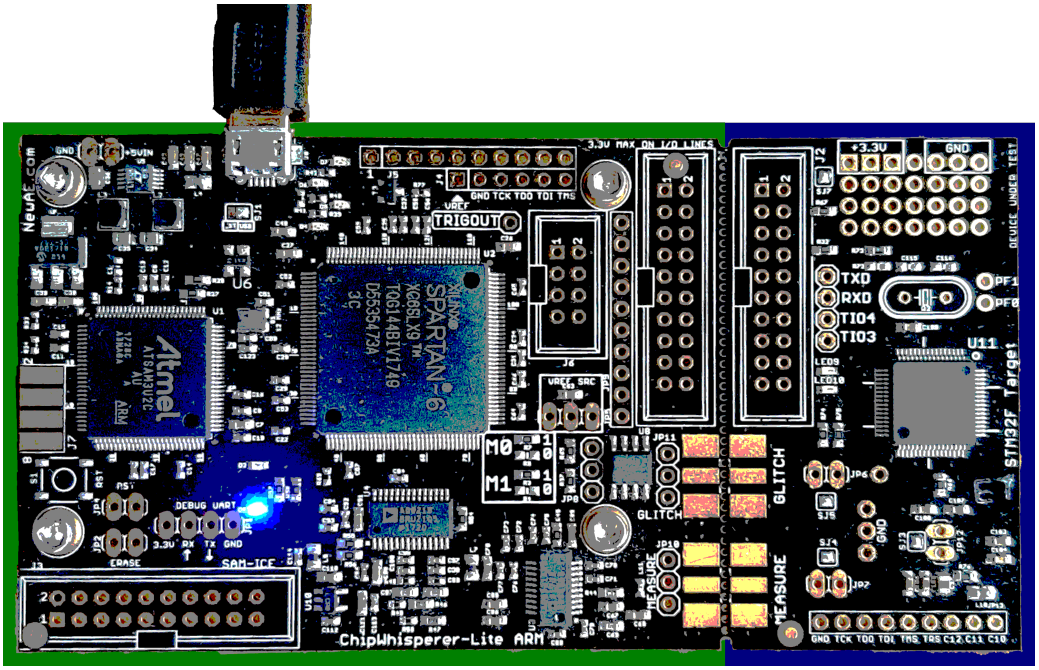

Figure 5.1: ChipWhisperer-Lite with ARM Target. Capture part is in the green frame, target board is in the blue one.

setup is, that both target and capture board use the same clock, so the oscilloscope to capture power traces runs synchronous to the target and therefore a way slower clock is sufficient compared to asynchronous oscilloscopes. In fact the oscilloscope can run with the same clock or four times faster than the target. Clock and voltage glitches are generated via FPGA-based pulse generation. One downside however is, that the capture sample stream cannot be passed on to the host. This limits capturing to 24400 samples per run, which makes it impossible to capture a complete run of BIKE with a single capture. But it is possible to only capture every i^{th} clock cycle, where i can be set on one's behalf. There are multiple interfaces that can be used for connecting with the capture board, e.g., we could use an external power supply and communicate via USART, so there would be no need to use the micro USB cable. But this would require some modifications to change the configuration, e.g., some solder jumpers would have to be altered. As the board is initially configured to be used with the USB interface, we simply stick to this one.

As mentioned, the capture and target boards can be separated, therefore the perforation between those parts has additionally to be cut before breaking them apart. Afterwards the 20 pin header, the glitch and the measure interface can be used. It is recommended to solder some jacks to the boards, if they are separated from each other.

The target board can be programmed via the capture board, so there is no additional

equipment required.

Target Board

The target board is called CW303 and it holds an *ARM STM32F303* chip. The most remarkable features are already mentioned in the previous section, so we will focus on the glitch infrastructure and highlight some hardware specifications of the target chip.
The stm32f303 has no clock generator included, thus it depends on an external clock. This makes it very easy to fault the clock and as target and capture board run synchronously with the same clock it gets even more easy. There are five different clock glitch modes:

i Clock only: Output only the original input clock.

ii Glitch only: Output only the glitch pulses - do not use the clock.

iii Clock OR: Output is high if either the clock or glitch are high.

iv Clock XOR: Output is high if clock and glitch are different.

v Enable only: Output is high for `glitch.repeat` cycles.

For voltage glitching, there are two MOSFET elements to either give a low power or a high power impulse. These short the target's power supply and should be used carefully, as incorrect usage can cause permanent damage.
As additional component the target board contains two LEDs which can be controlled by its firmware (if setup correctly).
The chip's most significant hardware features are 256 KB of flash memory, 48 KB of SRAM and clock frequency range from 4 to 32 MHz [STMa].

5.1.2 Discovery Board

The *STM32F407 Discovery kit* was originally designed to develop audio applications, but it became a widespread development board for a broad variety of algorithms like cryptographic implementations.
In Chapter 6 we will introduce the software project pqm4 on which this research is based. One of the supported hardware targets is the stm32f4discovery. Because of its larger memory (192 KB of RAM and 1 MB of flash) in comparison with the CW-Lite ARM target, the algorithms run more stable on this board and we used it for debugging.
There is a mini USB connector which can be used for powering the device. There are two 50 pin headers which provide access to the 100 pins of the stm32f407 chip. The pqm4

project uses the pins A2 and A3 for USART communication. According to the chip's manual [STMb] those are the pins of USART2 but can be used as GPIO pins, too. Besides several other interfaces (e.g. aux, micro USB, reset and user button) there are four LEDs which can be controlled by the firmware.
Figure 5.2 shows the hardware setup. The picture was taken while the target sent data via USART to the host, note the bright led on the upper part of the USART to USB adapter. It might be to small but it is right below the letters `RXD`.

5.2 Software

NewAE provides a preconfigured virtual machine image. When using this image one can access the ChipWhisperer in a comfortable way through *Jupyter Notebooks* [1] and utilize the provided *python* api. As mentioned earlier all the software is open source and licensed under the GPL. The community may participate and this has led to plenty of resources. This includes courses as jupyter notebooks to get familiar with the whole ChipWhisperer environment. They explain the whole idea of fault injection, provide setup scripts to connect to the hardware, explain side channel attacks via differential power analysis and illustrate the meaning of the different glitch parameters. To create reproducibility most methods used to interact with the targets are put in jupyter notebooks, too. This makes it easy for others to understand the way the results are generated as well as getting the hardware setup fast.

[1] https://jupyter.org/

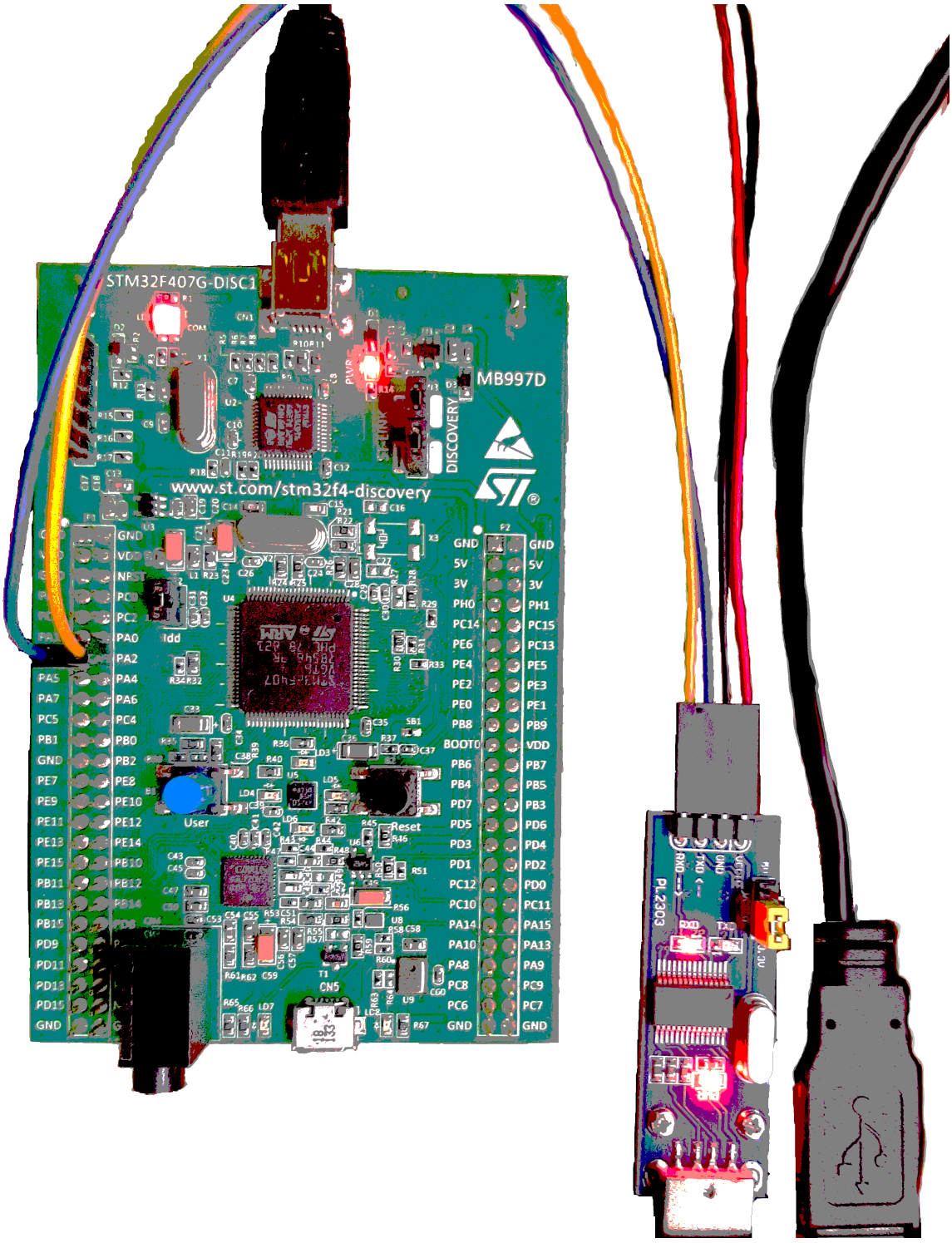

Figure 5.2: Discovery Board with stm32f407 Chip, connected USART via USB adapter and power supply via mini USB cable. USART: RX from host is connected to Pin A2 (yellow wire), TX from host is connected to Pin A3 (blue wire)

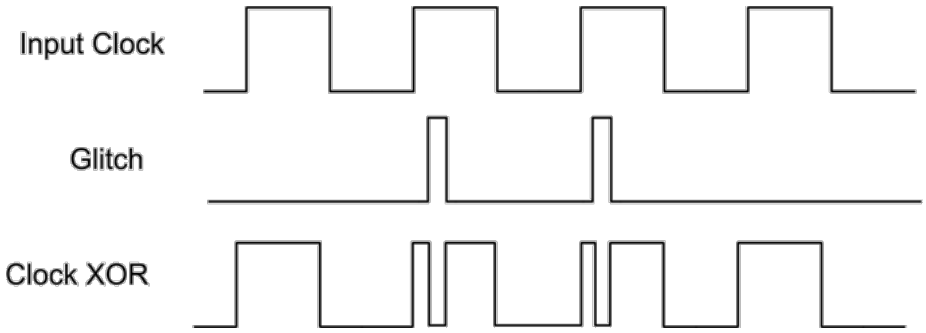

Figure 5.3: Clock XOR Generation [Newb]

5.3 Glitch Settings

To find glitch settings that seem promising, we used a simple firmware that checks if the input matches a given password, similar to the scenario described in Chapter 4. There are basically four parameters to consider. Two are firmware independent (and hardware dependent) and two are firmware dependent. The dependency of the latter ensues from the trigger event. For the ease of learning fault injection and taking power traces the targets firmware usually contains a `trigger_high()` which sets a specific pin to high and the counter part `trigger_low()`, this can be used to mark the start and end of a subroutine, or, more general speaking, this can mark relevant spots within the algorithm.
The parameters and their ranges are:

1. Width: can be between -49.8 and 49.8, it defines the length of the glitch pulse and is interpreted as percent of one clock cycle.

2. Offset: can be between -49.8 and 49.8, it defines when to start the glitch pulse and is interpreted as percent of one clock cycle.

3. Repeat: can be 0 to $2^{13} = 8192$, it defines for how many clock cycles the glitch pulse will be repeated.

4. Ext Offset: can be 0 to 2^{32}, it defines how many cycles to wait between trigger event and first glitch pulse.

We have to use different parameter sets for voltage and clock glitching and focus on width and offset. In the related figures we see successful glitches marked with a green + and the resets (when the glitch caused the target to crash) as red x.

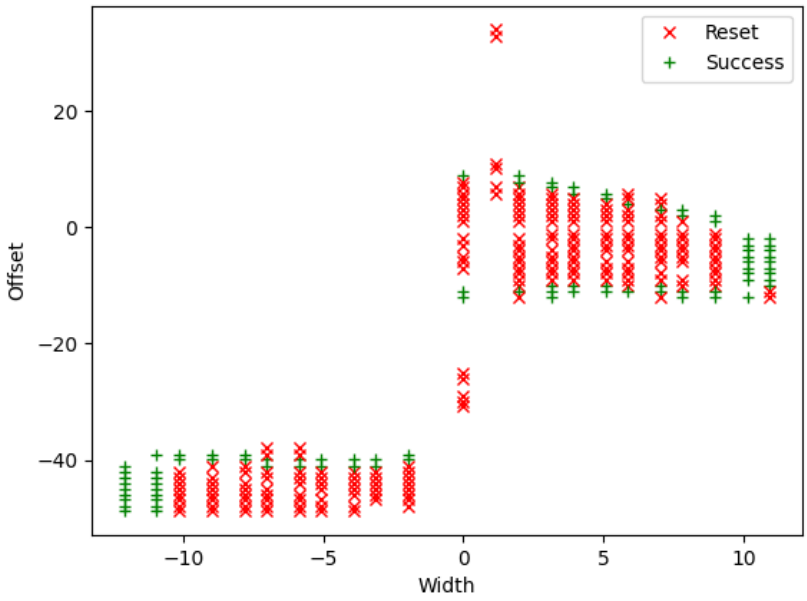

Figure 5.4: Clock Glitch Settings Overview

5.3.1 Clock Glitching

For clock glitching, we use the clock XOR mode, an example of the resulting clock pulses is given in Figure 5.3. Figure 5.4 gives an overview of the whole parameter range with successful and crashing settings. Already there we can see two major spots that seem to be a good choice. The first has the intervals (-12, -10.5) for width and (-49,-41) for offset, the second is located at (10,11.5) for width and (-10,-2) for offset. Even if those two spots will not work, several smaller spots delineate.

5.3.2 Voltage Glitching

For voltage glitching, it is harder to find good parameters, which gets obvious when we compare Figure 5.4 and Figure 5.5. In every quadrant of Figure 5.5, there are some successful glitches, though for the quadrants one and two, the upper half, they are very isolated or so seldom, that we missed them for quadrant one. Figure 5.6 highlights the more concentrated spots and from that we can derive two parameter sets with the following intervals: (i) width (-34,-32.5), offset (-47,-34) and (ii) width (33.5,34.5), offset (-30,-17).

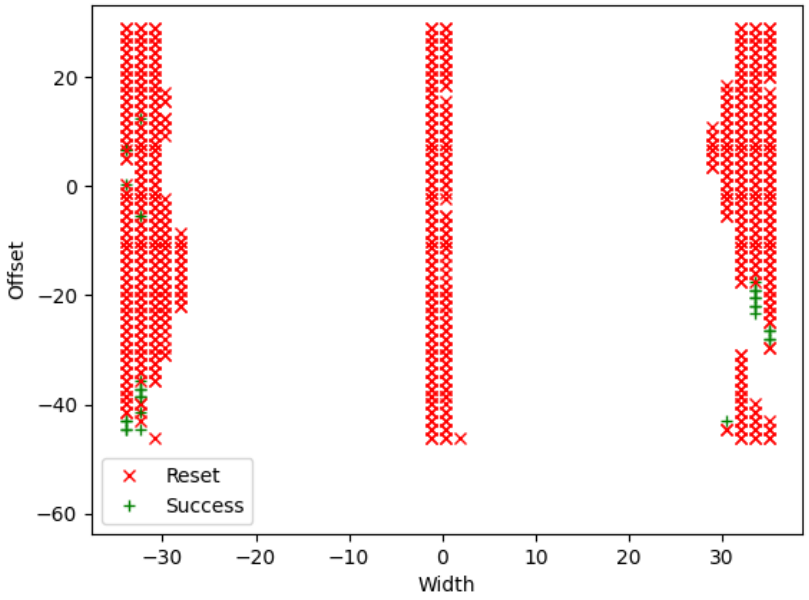

Figure 5.5: Voltage Glitch Settings Overview

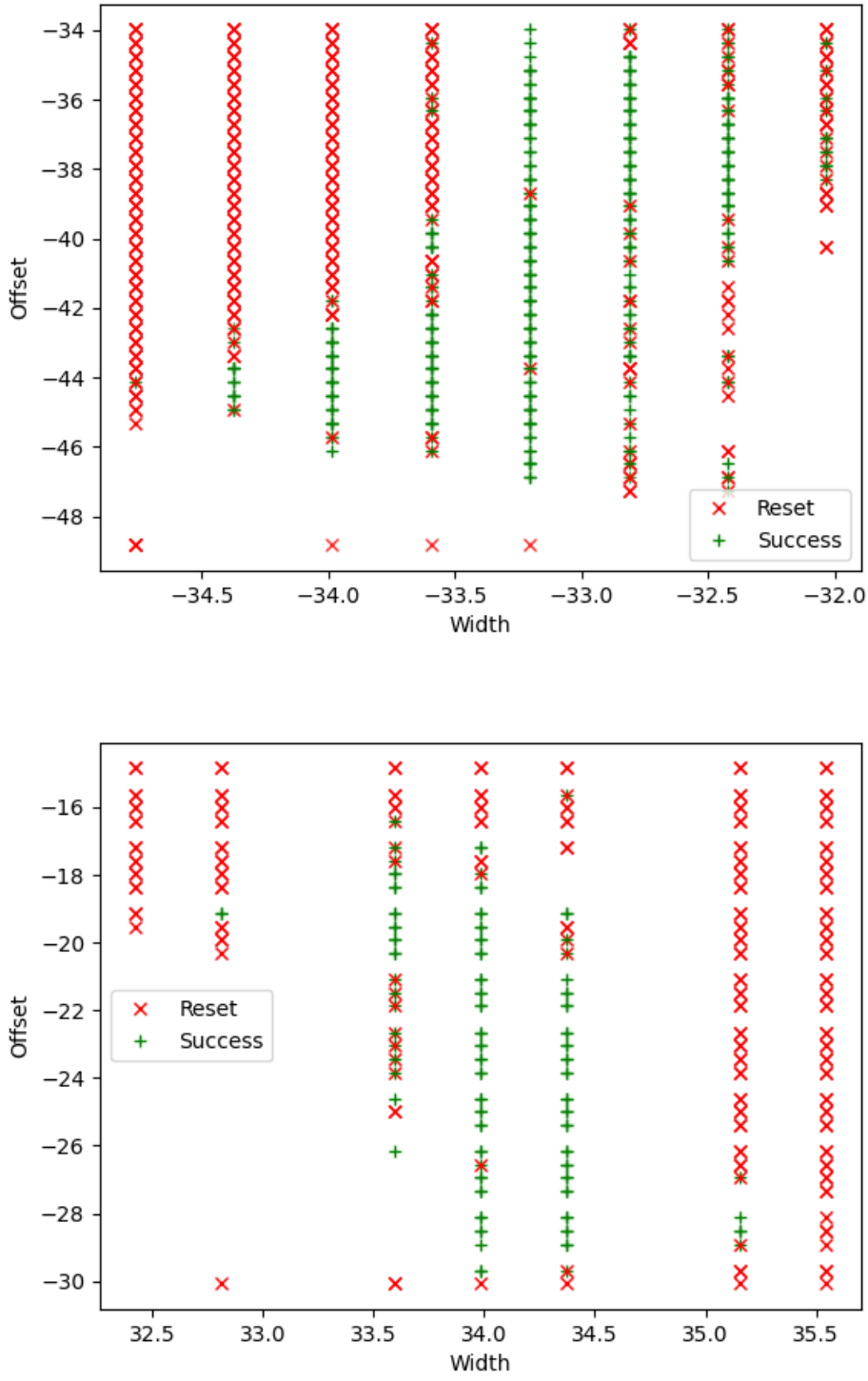

Figure 5.6: Voltage Glitch Settings Quadrants 3 and 4

6 Implementation

We will first introduce the concept of known answer tests, then discuss which implementation we chose as base and which functionality we had to add to the api and afterwards we will point out how we achieved this and which difficulties occurred. In the end of this chapter one can find some target board specifications of the targets we worked with and benchmarks we created for those boards.

6.1 Known Answer Tests

For the NIST standardization procedure it is obligatory to provide known answer tests (KAT). This is a concept to verify the correctness of one's implementation and/or setup. The idea is to generate an output with one implementation, a given pseudo random number generator (PRNG) and seed. This output can than be compared with an output from another implementation and the same PRNG. So for every submission, there is a reference implementation provided by the algorithm's authors to generate such KAT output files.
For the KAT creation NIST provides source code which implements a deterministic PRNG.

6.2 Where to start from

There are multiple repositories and/or implementations that initially seemed to be a good base for this thesis. We had to try different options as every repository had a different goal which was already fulfilled and other aspects that were incomplete.
A brief overview of implementations which came into consideration as base for the research:

- reference implementation [Mea] from the BIKE team. This one already holds an api to run the KATs as they were generated from this version. There is no git repository available but a zip file at their homepage.

- pqm4 [KRSS] implementation. This is a quasi standard for testing and analyzing all candidates from the NIST post quantum standardization process. The repository includes build configurations for multiple boards. Natively there is no possibility to send data to the board.

- ChipWhisperer [Newb] project. There is already a handy communication implemented called *SimpleSerial*. As this repository is intended to be used by the manufacturer of the board, it holds a lot of code specialized for the board. Many board and ChipWhisperer specific functions are already implemented, such as handling triggers, LEDs, building board specific firmware, flashing firmware and so on. This repo does not include an implementation of BIKE.

As pqm4 is a quasi standard we preferred it over the reference implementation and tried to merge either pqm4 into ChipWhisperer (cw) or the other way round.
It soon turned out that the pqm4 repository is quite complex (recursively it consists of five repositories as shown in Listing 6.1) and due to all its optimizations and dependencies it is very hard to move the essential parts into the cw environment. So we decided to use it as base and reimplement the cw part inside of the pqm4 repository.
For this we used the pqm4 repository as is with commit *b78dc00* and copied all files into the thesis' repository without using it as git submodule but as part of this repository. To provide reproducibility we use the virtual machine image provided by NewAE at version 5.6.1 which includes the ChipWhisperer repository at commit *271e278b*.

6.3 pqm4

As base implementation we chose the *pqm4* [KRSS] project. It is meant to provide an up to date implementation of all the NIST candidates. While it is based on the *mupq* project and includes all implementations from this repository, there are as well optimized versions for the ARM Cortex M4 of a large variety of post quantum kem and signature algorithms. Every single implementation is included in this repository as a distinct scheme and to achieve the different security levels proposed by the NIST, there are multiple schemes of the same algorithm, only differing by the security parameters, i.e., key length. The m4 optimized schemes can be distinguished from the mupq schemes and the pqclean schemes by the location and, as the name is derived from the location, by the name. For BIKE there are initially the schemes *crypto_kem_bikel1_m4f*, *crypto_kem_bikel3_m4f*, *mupq_crypto_kem_bikel1_opt*, *mupq_crypto_kem_bikel3_opt* available. The pqm4 repository supports multiple ARM Cortex M4 boards, namely *cw308t-stm32f3*, *mps2-an386*, *stm32f4discovery*, *nucleo-l4r5zi* and *nucleo-l476rg*.

6.3.1 Repository Structure

To alter and add code inside the pqm4 project it is important to understand the structure, the interference and dependency between the different repositories this project is

composed of. Here we will point out the most significant aspects of the repository and makefile structure with the aim to ease the understanding of the following sections.

- For every platform, there is a makefile in *pqm4/mk*. One important thing here is the `EXCLUDED_SCHEMES` variable which defines schemes and scheme paths to exclude from the build process. As we are only focused on BIKE, we can exclude every other scheme to quicken the build process.
- For the physical targets (mps2-an386 will be virtualized) pqm4 uses libopencm3 [PETNH], which aims to provide a firmware library for various ARM Cortex-M microcontrollers. For our purpose, this repository can be left unchanged. It provides a good interface to the hardware to build further functions upon.
- In *pqm4/common*, there are two files which will be compiled to *aestest* and *boardtest* which happens once per board. Those build targets are defined in *pqm4/mk/tests.mk*.
- In contrast to the previous, the c files in *pqm4/mupq/crypto_{kem, sign}* are evaluated for every scheme. Those define the firmware which later can be flashed to the target boards. The build targets of those are defined in *pqm4/mupq/mk/schemes.mk*.
- While most header files for the pqm4/mupq firmware are located in *pqm4/mupq/common*, their implementation can be found in *pqm4/common*.
- All available schemes are located in their own subfolder inside the folder *crypto_{kem, sign}* for the three modules pqm4, mupq and pqclean. The scheme name is composed of the absolute path. Considering the directory tree described in Listing 6.1, a valid scheme name is for example, *mupq_crypto_kem_bikel1_opt*. Because the pqclean repository does not contain a BIKE scheme it can mainly be ignored.
- In the root of pqm4, there are python scripts to automatically run tests and create benchmarks. These and their syntax are well described in *pqm4/README.md*. They even flash the firmware on the target.

Listing 6.1: pqm4 and submodules

```
1 pqm4
2 |-libopencm3
3 |-mupq
4   |-pqclean
5     |-test/pycparser
```

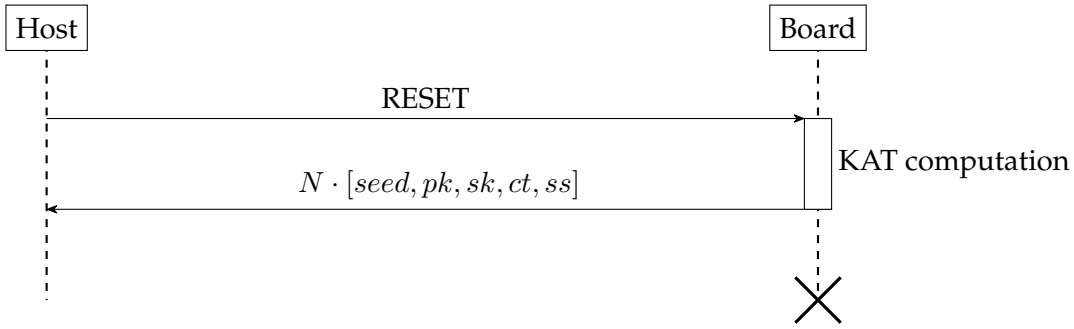

Figure 6.1: KAT Target Api

6.3.2 API

We can differ two use cases, the first is to run KATs as described in Section 6.1, to verify that the board is setup correctly and the second is to interactively control and communicate to the board. For fault injection, it is very useful to start specific subroutines, read the key/ciphertext outputted by the functions, set keys and PRNG for calculation and use other helpful minor functions.

1. Because the memory on the board is quite limited and other factors that will be discussed later in Section 6.4, it seems to be easiest to use the same computation as used for the KAT generation and constantly send the output to the host. Hence comparison can be done by the host. A sequence diagram is given in Figure 6.1.

2. For the interactive firmware the following commands should be available: **start** *keygen*, *encaps* and *decaps* separately, **read** separately either $key_b = ss$ decapsulated, $key_a = ss$ to encapsulate, pk, sk or ct, **write** either pk, sk or ct, reset and regenerate the **pseudo random number generator**. A sequence diagram is given in Figure 6.2

6.4 Adapting the Implementation

For the following passage, file paths assume to start at the root of the pqm4 repository except for paths starting with cw which refer to the path *hardware/victims/firmware/* inside the ChipWhisperer repository.
Inspired by the cw/simpleserial and due to the fact, that we hoped we could use the python SimpleSerial counterpart, we reimplemented some of its functions in common/-hal-opencm3.c and common/hal-mps2.c. Receiving data via UART/USART was not implemented in pqm4, for the physical targets libopencm3 implemented UART/USART and we could build upon that, but for mps2-an386 we even had to implement the very

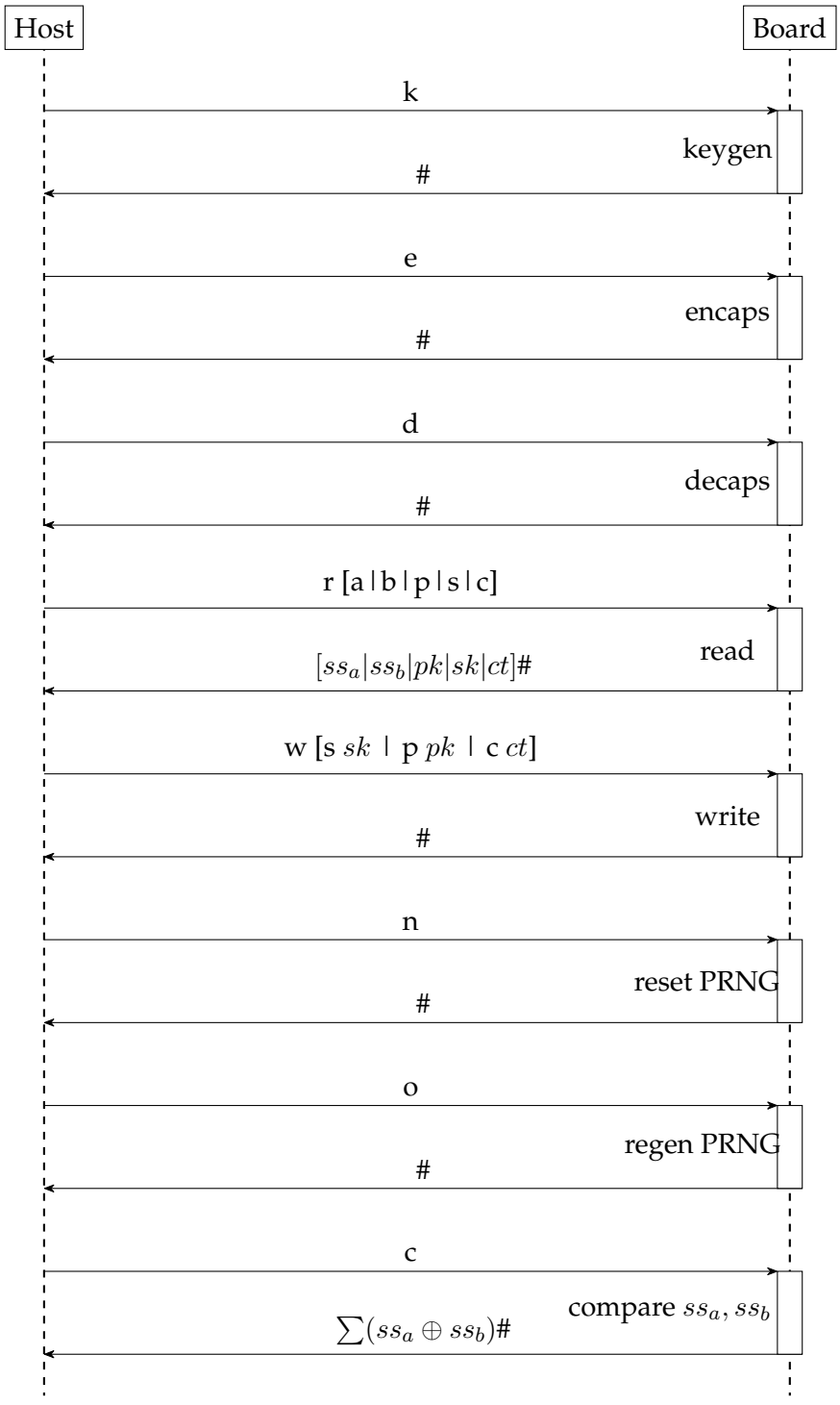

Figure 6.2: Interactive Target Api: [a | b] is meant as "one has to be chosen", italic names refer to the corresponding bytes (if sent from host) or hexadecimal string (if received by host), the shared secret check is computed bytewise.

basics before we could use the UART/USART Rx at all. The header file for this is mupq/-common/hal.h, to make the new functions available in the pqm4 project they need to be referenced there. The functions *getch* and *putch* are simple wrappers to either receive or send one byte via UART/USART, *hal_receive* enables the target to receive from UART/USART a defined amount of bytes, *hal_send* is the equivalent for sending (but this function was not used because sending hexadecimal strings at some point turned out to be easier to handle). The most important functions for fault injection in this setup are *trigger_setup, trigger_high, trigger_low*, as they signal the host when to start the fault injection. The *LED* related functions reside here as well; they were quite helpful for debugging. With them we could get the point of crashing more precise before there was a proper way of communication. Most board specific subroutines used there can be found in libopencm3/-include/libopencm3/stm32/ (headers) and libopencm3/lib/stm32/ (c files), while corresponding functions are mainly implemented in cw/hal/stm32f3/stm32f3-hal.c and cw/hal/stm32f3/stm32f3-hal_lowlevel.c.

The ChipWhisperer SimpleSerial utility provides an easy to use interface to control the target during run time and even send data to it. It is implemented in cw/simpleserial and works like a client server protocol. The target listens on the UART/USART interface and if a transmitted byte matches a known command, it parses the rest respectively to the command's expected payload, parses it and starts the defined subroutine. This seems to be a perfect interface to implement the API described in Figure 6.2. Unfortunately SimpleSerial's payload length is limited to 256 bytes of data. This is way too small for BIKE's key lengths, so we implemented a similar communication api in mupq/crypto_kem/fi.c and common/serial-com.c. The header file is common/serial-com.h. They can be included via the makefile mk/serial.mk. Using the function *sprintf* from <*stdio.h*> to format an integer to a string blows up the binary, because multiple functions from the library will be added. This is unintended as flashing the cw308t-stm32f3 is very slow and size dependent. So we wrote the function *nibble_to_hex* in serial-com.c to achieve the same result.

As mentioned earlier for KATs we need a deterministic pseudo random number generator (PRNG). The PRNG we need is already implemented in *pqclean* in the file mupq-/pqclean/test/common/nistkatrng.c. We use KATs only for the targets cw308t-stm32f3, stm32f4discovery and mps2-an386 so it is sufficient to change three makefiles. In mupq-/mk/schemes.mk we set a flag to distinguish the correct environment, this means in the *define schemelib* block we added the line `elf/$(2)_kat.elf: NIST_KAT_RANDOMBYTES=1` for the mupq/crypto_kem/kat.c target. Dependent on those flags in mk/opencm3.mk and mk/mps2-an386.mk the hal library is chosen. I.e. we added the definition for obj-/libpqm4hal-kat.a and extended the checkbacks to append the correct strings to `LDLIBS` and `LIBDEPS`. Those adaptions take place within the first 18 lines of mk/opencm3.mk.

The equivalent changes can be found in mk/mps2-an386.mk in lines 79 to 96. For being able to use the initialization function from mupq/pqclean/test/common/nistkatrng.c we added the mupq/common/nistkatrng.h header file.
Finally to generate new firmware files that can be flashed to the target and which use the above described alterations we added mupq/crypto_kem/kat.c and mupq/crypto_kem-/fi.c. The former implements the API visualized in Figure 6.1 the latter implements the API visualized in Figure 6.2. To add them to the build targets for every scheme we have to mention them in mupq/mk/schemes.mk in the lines starting with *tests: elf...* and *tests-bin: bin...*, to be sure we appended them to *tests-hex: bin...*, too.
On host side there are some python scripts integrated in the jupyter notebooks *KAT.ipynb*, *Interactive Target.ipynb* and *Fault Injection.ipynb* to handle the counterpart of the api.

6.4.1 Challenges

In the previous part some difficulties were already broached, now we highlight them and the consequences we derived from them. It was not a plug and play experience to run BIKE on the cw308t-stm32f3, though the pqm4 project claims to support this board and BIKE was not excluded for this target board in mk/cw308t-stm32f3.mk. There are benchmarks already signaling that the SRAM of cw308t-stm32f3 is not sufficient for the implementation of BIKE. Nevertheless we tried it, which caused a lot of trial-and-error of modifying the firmware, flashing it and facing that it does not work as expected. As flashing a firmware of 50 KiB in size takes almost 90 seconds (which is very slow) we tried to keep the firmware file as small as possible for debugging, e.g., by writing a function to print a hexadecimal string from an integer.
Another approach was to write the key material to the flash memory and read the keys and ciphertext from there, but even this did not work and the target crashed because of too little memory. The idea of using the flash memory as extended RAM seems to be out of scope, though flash storage can be written to during run time and the term rom does not absolutely fit. This is because we need a special operation for writing. In the submodule libopencm3 it is called `void flash_program_half_word(uint32_t address, uint16_t data)` (defined in the file libopencm3/include/libopencm3/stm32/f3/flash.h) and as the name indicates it always writes a half word, i.e., 16 bit. This would cause a total reimplementation if one would like the keys natively to be saved in flash memory.
So at some point we switched to another target. Cw308t-stm32f3 has only 40 KiB of SRAM, stm32f4discovery has already 192 KiB RAM and mps2-an386 is a FPGA that can be virtualized with QEMU, it holds 4 MiB of RAM so there is comparatively no memory limit. On mps2-an386, every algorithm can run without any difficulties. BIKE level 1 ran out of the box on stm32f4discovery, but BIKE level 3 was too large even for this target. But with the

discovery board we had a second target which has a quite similar communication interface like cw308t-stm32f3 and we could use it for debugging and only had to make minor changes to use the previous adaptions on the new target. Here flashing a 50 KiB firmware takes less than 5 seconds and due to the additional memory the target did not crash that often.
But still it was not possible to run a BIKE implementation on cw308t-stm32f3. For this, we had to add new BIKE security levels, their parameters are described in the Section 2.3.
For adaption we had to alter three files in the *opt* implementation, namely bike_defs.h, decode.c and gf2x_inv.c. While in `bike_defs.h` block length r, key weight $w/2 = d$, error weight t, threshold coefficient 0, threshold coefficient 1, threshold minimum $\frac{d+1}{2}$ and block bits $2^{\lceil \log r \rceil}$ are defined. In `decode.c` we have to set decoder parameters for the new levels (it is essential for `BGF_DECODER`). In `gf2x_inv.c` are some hard coded values and in the comments above is *sage* code to generate those for different values of r, so the hard coded values can easily be expanded to the new levels.
The *m4f* implementation requires much more adjustment, in at least `types.h`, `ring_ops.c` and `decode.c` changes have to be made. In `decode.c` a lot of values seem to be hard coded; this caused us to skip the adjustment for the *m4f* implementation. But finally we had a BIKE implementation running on our target board of choice cw308t-stm32f3.

KAT verification

After this first success the next step was to verify the implementation's correctness and at the same time, that the new parameter sets are working. For this we took the reference implementation from [Mea] in the version 4.1, added the parameter sets there and generated KAT files for the new levels. We had to choose version 4.1 because the pqm4 implements this version, too. The successing versions replace AES and SHA2 by SHA3 and use a data oblivious sampling method to generate fixed weight vectors. These changes should not effect the overall behavior, but cause a different output.
Because the BIKE implementation included in pqm4 is based on the constant time implementation from Amazon and they have a different secret key representation than the Reference Implementation, we wrote a tool to extract the corresponding key material. This tool can be found at kat_bike.py and is used in the KAT notebook. From the `types.h` file, we can reconstruct the key composition. For pqm4 it is: $sk_{pqm4} = (\textit{weight list}, (h_0, h_1), pk, \sigma)$, this makes a total length of $w \cdot 4 + 2 \cdot R + R + l/8$ bytes with $R = \lceil r/8 \rceil$ the number of bytes needed to sotre r bits. The *weight list* is an alternative repersentation of (h_0, h_1), only indexing the set bits. Because we are working with 32 bit processors one index has the length of $32\ bits = 4\ bytes$. For the Reference Implementation it is exactly like specified, $sk_{ref} = ((h_0, h_1), \sigma)$, which makes a total length of $2 \cdot R + l/8$

bytes. For every level $l = 256$, so we could replace $l/8$ with 32.
We could verify that the levels we work with create the same output on cw308t-stm32f3, stm32f4discovery, mps2-an386 and the Reference Implementation. There is the jupyter notebook KAT.ipynb which implements KAT automation. With this we found a single decoding failure took place for level 11. This might be because the parameters are so small that parameter inaccuracies have more impact. At the same time we considered the advantage of the speedup higher than the loss due to the DFR with the correct parameters.

Testvectors

For the sake of completeness we mention, that the pqm4 project provides a test which is similar to KAT. This is realized by mupq/crypto_kem/testvectors.c. There is a deterministic randombyte generator implemented and it can be run on the host and on the target board to compare the output. This test is passed by the targets for the new and native security levels.
Nevertheless it is obvious, that both calculations use the same implementation, so we did not pay too much attention to this feature, but focused on KAT. As KAT can be used over different implementations they are a way more reliable and important test.

6.5 Benchmarks

With BIKE running on the target we created benchmarks from the three targets we were working with in the meantime. Some of the hardware specifications of the different boards can be found in Table 6.1, they are all taken from the manuals [Lim, STMb, STMa]. All the benchmarks were generated with the python scripts provided by the pqm4 project, they are documented in the corresponding readme.
Table 6.3 shows the cycles needed for computation. For cw308t-stm32f3 we only show one value, because cw308t-stm32f3 has no hardware random number generator, but uses a pseudo random number generator. Because the benchmarks were taken after a board restart, every of the 10 iterations calculates with the exact same random bytes, so it always takes the exact same number of cycles. For mps2-an386 the large variety of clock cycles of the *opt* implementation is remarkable, but the cycle count for the virtualized board is considered meaningless according to the pqm4 readme. Thats why we only show one scheme for mps2-an386.
In Table 6.2 the memory consumption is captured. As one can see even bikel01 (adapted from [Ket22]) takes more than the available memory on cw308t-stm32f3. Even the keygen method from bikel01 crashes cw308t-stm32f3, because the keys are allocated before the

cw308t-stm32f3	Flash	256 KB
	Pages	128 á 2 KiB
	SRAM	40 KiB
	CCM RAM	8 KiB
	max Clock	32 MHz
	random	pseudo
stm32f4discovery	Flash	1 MiB
	RAM	192 KiB
	max Clock	168 MHz
	random	hardware
mps2-an386	Flash	$\geq$ 64KiB
	RAM	$\geq$ 4MiB
	sys Clock	25 MHz

Table 6.1: Target Hardware Specifications

function is called. Listing 6.2 shows the key allocation in *pqm4/mupq/crypto_kem/stack.c*. More benchmarks can be found in the folder *benchmarks*.

Listing 6.2: Stack Test

```
1 unsigned char key_a[MUPQ_CRYPTO_BYTES], key_b[MUPQ_CRYPTO_BYTES];
2 unsigned char pk[MUPQ_CRYPTO_PUBLICKEYBYTES];
3 unsigned char sendb[MUPQ_CRYPTO_CIPHERTEXTBYTES];
4 unsigned char sk_a[MUPQ_CRYPTO_SECRETKEYBYTES];
5 unsigned int stack_key_gen, stack_encaps, stack_decaps;
```

Level	impl	key gen	encaps	decaps
		mps2-an386		
bikel1	opt	3,720	2,184	5,520
bikel2	opt	3,840	2,268	4,092
bikel3	opt	5,856	3,552	9,368
bikel4	opt	6,184	3,728	11,120
bikel5	opt	9,904	6,136	17,024
bikel6	opt	9,904	6,136	17,048
bikel7	opt	10,728	6,624	21,748
bikel8	opt	18,032	11,344	33,592
bikel1	m4f	43,744	28,632	96,040
bikel1	opt	35,952	22,824	79,212
bikel3	m4f	85,512	56,288	195,064
bikel3	opt	69,624	44,416	157,364
		stm32f4discovery		
bikel1	opt	3,720	2,184	5,520
bikel2	opt	3,840	2,268	4,092
bikel3	opt	5,856	3,552	9,480
bikel4	opt	6,296	3,728	11,120
bikel5	opt	10,016	6,136	17,024
bikel6	opt	10,016	6,136	17,048
bikel7	opt	10,840	6,624	21,748
bikel8	opt	18,144	11,344	33,704
bikel01	opt	19,320	12,096	40,252
bikel1	m4f	43,744	28,632	95,928
bikel1	opt	35,840	22,712	79,212
		cw308t-stm32f3		
bikel1	opt	3,720	2,184	5,520
bikel2	opt	3,840	2,268	4,092
bikel3	opt	5,856	3,552	9,368
bikel4	opt	6,184	3,728	11,120
bikel5	opt	10,016	6,136	17,024
bikel6	opt	10,016	6,136	17,048
bikel7	opt	10,840	6,624	21,748

Table 6.2: Memory Evaluation in Bytes

Level		key gen	encaps	decaps
		mps2-an386	50 executions	
bikel1	MAX:	18,446,744,073,696,421,082	644,764	7,231,480
opt	AVG:	368,934,881,477,444,352	380,649	5,219,579
	MIN:	2,924,814	305,037	3,842,998
		stm32f4discovery	10 executions	
bikel11	AVG:	3,438,044	229,887	2,298,661
opt	MIN:	3,432,255	226,826	2,295,603
	MAX:	3,444,546	235,174	2,303,946
bikel12	AVG:	6,223,140	248,327	2,885,846
opt	MIN:	6,223,125	247,500	2,884,951
	MAX:	6,223,279	251,443	2,889,009
bikel13	AVG:	6,765,799	460,761	5,922,580
opt	MIN:	6,759,605	449,612	5,911,442
	MAX:	6,771,896	466,123	5,927,953
bikel14	AVG:	13,728,440	566,454	8,973,846
opt	MIN:	13,728,320	565,829	8,973,220
	MAX:	13,728,853	570,842	8,978,235
bikel15	AVG:	14,792,388	1,023,656	14,558,950
opt	MIN:	14,773,981	1,012,902	14,548,187
	MAX:	14,815,170	1,037,638	14,572,903
bikel16	AVG:	16,198,107	1,015,483	14,552,296
opt	MIN:	16,183,770	996,659	14,533,554
	MAX:	16,208,867	1,037,552	14,574,424
bikel17	AVG:	35,714,009	1,426,892	28,624,711
opt	MIN:	35,713,466	1,423,306	28,621,120
	MAX:	35,717,639	1,427,785	28,625,591
bikel1	MAX:	27,523,266	3,425,016	61,765,975
m4f	AVG:	27,512,672	3,409,224	61,750,117
	MIN:	27,503,210	3,403,922	61,744,771
bikel1	MAX:	173,697,956	9,883,167	219,697,637
opt	AVG:	173,676,423	9,869,754	219,684,159
	MIN:	173,661,232	9,854,642	219,669,112
		cw308t-stm32f3	10 executions	opt
bikel11		3,385,825	231,357	2,238,475
bikel12		6,119,291	251,604	2,804,906
bikel13		6,648,505	454,585	5,754,618
bikel14		13,488,329	565,648	8,703,531
bikel15		14,528,889	1,030,551	14,220,227
bikel16		15,900,551	1,030,463	14,221,931
bikel17		35,069,628	1,417,797	27,928,081

Table 6.3: Speed Evaluation in clock cycles

7 Fault Injection Attack

First, we give an overview of the tools we developed in python. With those, we were able to generate interesting data, such as DFR, supporting further analysis. To determine promising spots for fault injection, we will have a look at the source code and analyze the resulting assembly dump. The previous steps finally enable us to begin with fault injection and we will discuss successful settings.

7.1 Tools and Preparation

In the previous chapter we focused on the software for our targets, now we will take a look at the counter part of this, the tools needed on the host side. All the implementations can be found in either *scripts* (for python/sage scripts) or *jupyter* (for jupyter notebooks). Those obviously include the counter part of the serial communication. While this is implemented in *target_com.py* and it is only a wrapper to send and receive data via UART/-USART, there are other tools to handle the data.

7.1.1 Key Representation

When implementing KAT we faced that the key representation in pqm4 differs to KAT files generated with the Reference Implementation. As shown in Section 6.4.1, the pqm4 key is composed of $(\textit{weight list}, (h_0, h_1), pk, \sigma) = sk_{pqm4}$, whereas in the Reference Implementation the secret key is stored as described in the specification, $((h_0, h_1), \sigma) = sk_{ref}$. This obviously results in different lengths. To handle these differences, we calculate offsets and lengths depending on the block length r and the row weight d and store them in *kat_bike.Level* objects. Besides this, *kat_bike* implements a KAT parser and comparator for both pqm4 and reference keys. While reading the data from the target, *kat_bike* checks whether Bob's and Alice's shared secret match. If they do not, a decoding failure happened. If the Reference Implementation behaves the same way decoding failures do *not* cause the KAT to fail.
Building upon *kat_bike*, there is *bike_key* which implements most of BIKE key handling. It can either read a sk_{pqm4} key or a sk_{ref} key and calculate the missing parts from the given information. Additionally, we can create new random keys. All of those keys are accepted by the target firmware and can be transferred to and read from the target.

If we compare Definition 3.4 with the key representation above, we see there are multiple parts of the key which could be faulted. In the sk_{pqm4} key the secret key is saved in redundant forms, as binary vector (h_0, h_1) and as index list of the set bits, *weight list*. From this various alternation options arise. A combination of these options forms a *fault mode*.

7.1.2 Fault Modes

A *fault mode* defines how a faulted key was altered or will be generated. We found four properties that have to be considered:

1. *Faulty Key Kind*: according to Definition 3.4 we can distinguish type *one*, type *two.a* and type *two.b*.

2. *Weight List Kind*: For weights lower than level's row weight d the question arises: how to deal with the additional index entries? Either an existing entry could be referenced *multiple* times, they could point to *unset* bits or to an *invalid* index, i.e. an index i is invalid if $i < 0$ or $i > r$. One can think of a total *mismatch* between the vector representation and the index list.

3. *Public Key Kind*: Due to the redundant representation we can derive the public key either from the *secret key* or from the *weight list*.

4. *Fault Target*: Either the *secret key*, *weight list* or *both* were/are supposed to be faulted.

Fault modes are implemented in *bike_key* as well. Because some information is lost if the faulted weight is higher than the level's row weight, we state *undefined* for *Faulty Key Kind* and *Weight List Kind* to mark information loss during the analysis.
With these fault modes, we were able to generate intentionally faulted keys on the host, send them to the target and calculate the decoding failure rate for the different fault modes separately. Besides generating faulted keys, we implemented an analyzer for faulted keys to determine the achieved fault mode when faulting keys on the target.
In addition to the previously mentioned tools, we also developed a tool to calculate parameters defining a BIKE security level. This is implemented in the files *BIKE_params.sage* and *threshold.sage*, while the latter is mainly adopted from [Ket22]. An explanation was already provided in Section 2.3.

7.1.3 Jupyter Notebooks

All those tools form the foundation for the jupyter notebooks. There are three notebooks meant for direct interaction, *KAT.ipynb, Interactive Target.ipynb* and *Fault Injection.ipynb*,

while the others are invoked by them. Those notebooks give examples of how to use the python tools.
The *KAT* notebook allows to run KATs on the targets cw308t-stm32f3, stm32f4discovery, mps2-an386 and compares the output with previously generated KAT files (those in the repository were generated with the Reference Implementation). With the *Interactive Target* notebook, one can test the communication with the targets, check the correctness of the python tools, capture power traces, test faulty key generation and analysis, compute and visualize decoding failure rate.
With the given hardware, we considered capturing power traces as insufficient, because the ChipWhisperer-Lite allows at maximum to capture 24400 samples per run. Most subroutines take way more clock cycles, thus we would have to take a capture over several runs (which is possible by setting the PRNG to the same initial state). But this causes spikes overtopping the rest of the trace, when the next capture segment starts. The Fault Injection notebook allows to fault the target with different settings (triggers have to be set in the pqm4 firmware explicitly). If the fault injection was successful and a faulted key was generated, we save the keys to later calculate the DFR for this key.

Lines of Code

To give an impression of the effort made, we show the source lines of code (SLOC). For python we used *radon* and for c we used *sloccount* to count the SLOC. The jupyter notebooks *Fault Injection, Flash_Disco_CWLITE, Interactive Target, Setup_DISCO_or_CWLITE* and *KAT* have a total of 464 SLOC. The python scripts *threshold, BIKE_params, bike_key, kat_bike,* and *target_com* have a total of 880 SLOC. Here we did not count the SLOC of the sage files of *threshold* and *BIKE_params* but the python files generated after preparsing, which makes a marginal difference. The c files *pqm4/common/serial.c, pqm4/mupq/crypto_kem/fi.c* and *pqm4/mupq/crypto_kem/kat.c* have a total of 323 SLOC. The mentioned c files are only those which were added to the pqm4 project. The overall effort made to adapt the pqm4 project is described in Section 6.4 in detail and can hardly be measured in SLOC. This makes a total of 1748 SLOC written for this thesis.

7.1.4 Actual Decoding Failure Rate

Given the tools described above, we calculated DFRs for all relevant fault modes. While calculating DFRs, we discovered some combinations to be insignificant, because they do not make any sense. These combinations are:

- Faulting only the secret key. In both implementations we worked with, the public key is derived from the vector representation. Hence equal behavior can be seen

with other fault modes and we can not distinguish the fault mode from analyzing the key structure.

- Deriving the public key from the weight list and the weight list pointing to invalid indices. Using invalid indices makes it impossible to properly apply the mathematics to calculate the public key.
- The weight list mismatches the secret key absolutely. A total mismatch corresponds to using two unrelated keys to first encapsulate the shared secret and later decapsulate the ciphertext. This leads to a DFR of almost 100 percent as shown in Figure 7.1.

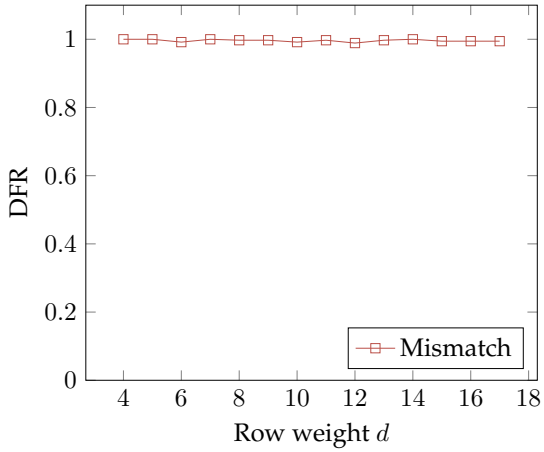

Figure 7.1: Weight List Kind *mismatch* can not be successfully decoded.

Sorting out all those combinations leads to 30 different interesting fault modes, which is even less than half of the originally $72 = 3 \cdot 4 \cdot 2 \cdot 3$ combinations.
We took relevant plots to highlight the effects of the different fault mode properties we studied. All of the plots were generated with level 11, as defined in Section 2.3, because we can compute more DFRs in the same time, compared to level 1. This is due to the faster computation of single samples while handling smaller keys and by the fact that the interval of row weights we have to observe is much smaller. For the given plots, we took 360 samples, 30 samples with 12 keys, for every point plotted. We verified that level 1 behaves the same way with the same fault mode by calculating DFRs with 50 samples per weight (10 samples per key with 5 keys). All targets, cw308t-stm32f3, stm32f4discovery and mps2-an386, generate the same output.

Weight List Kind In Figure 7.2 we can see that the weight list kind *multi* causes DFR to raise faster than the other options. The types *multi*, *unset* and *invalid* only have an effect if the row weight is lower than the level's weight, because they define how to

deal with potential empty indices. For level 11 the optimum weight is 9. Hence only DFR for values $d < 9$ are of interest. Besides the key generation, the weight lists are only used in the functions `find_err1` and `find_err2` to `rotate_right` the syndrome. These functions are called from `decode`. The exact cause of the different behavior has further to be researched.

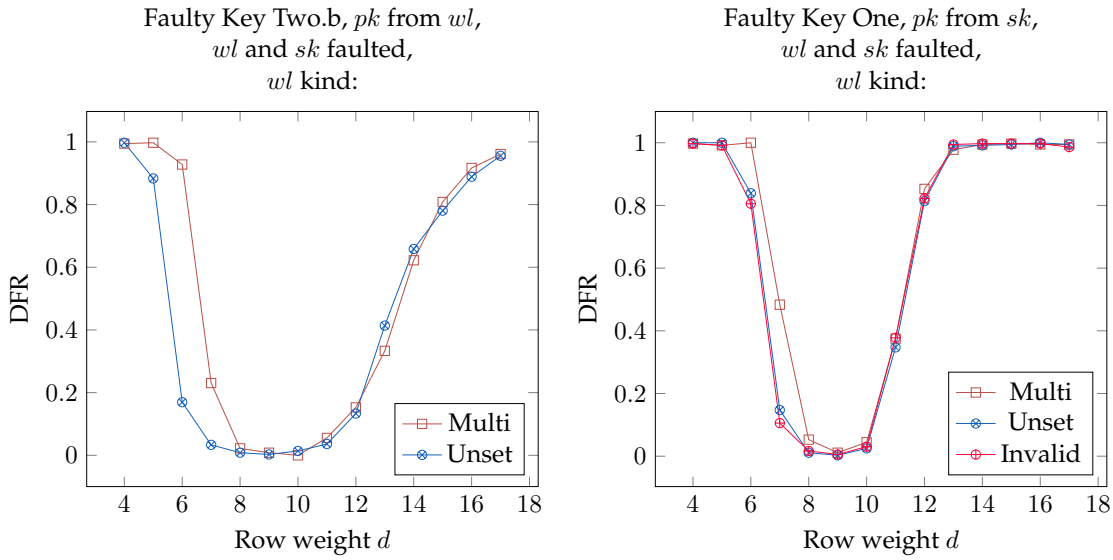

Figure 7.2: Impact of different Weight List Kinds.

Faulty Key Type In the upper left plot of Figure 7.3, we see that the DFR rises faster for Faulty Key *one* than for Faulty Key *two*. This holds for several combinations of the other properties.

In the upper right plot of the same figure, we can see that Faulty Key *two.a* has the same DFR as *two.b* for odd weights, while the DFR is very high for even weights. This is caused as h_0 has to be of odd weight to guarantee that it is invertible.

In the lower picture of Figure 7.3, we see the odd behavior of only faulting the weight list. Type *one* and type *two.a* might behave the same way because in both cases h_0 is faulted. At the same time type *two.b* shows a smooth behavior as seen with many other fault modes. This might be caused by the wider impact of faulting h_0 (because it will be inverted) and the combination with the other properties (deriving the public key from the weight list and only faulting the weight list but not the vector representation).

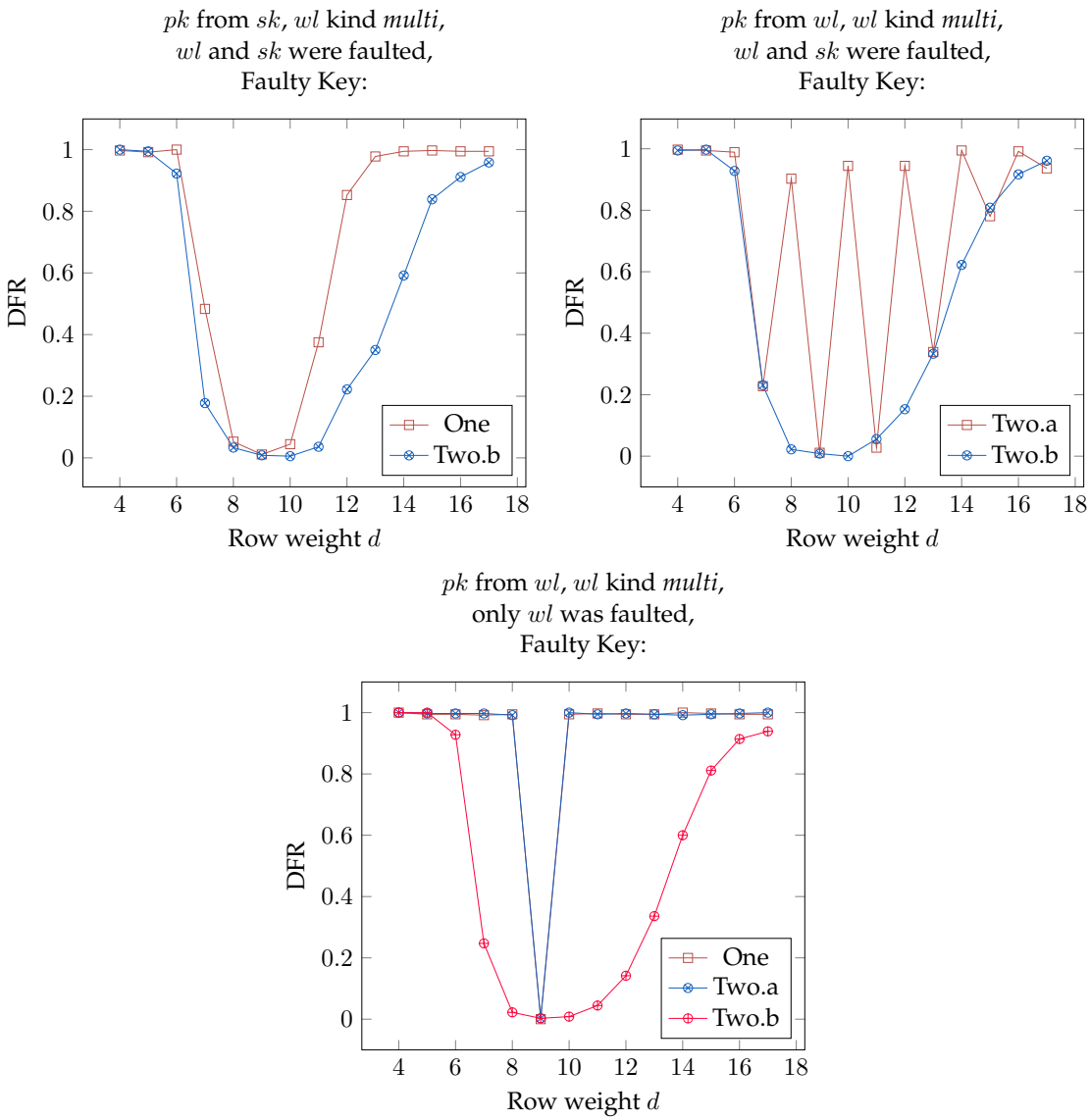

Figure 7.3: Impact of different Faulty Key Types.

Public Key Kind For the public key kind we observed multiple dependencies to other fault properties. The left plot of Figure 7.4 shows the large difference if chosen Faulty Key *two.a* and only the weight list is faulted. For Faulty Key *two.b* this has only little impact, as shown in the lower plot. In both plots we can see that deriving the public key pk from the secret key sk has no impact if only the weight list is faulted and we can remain with a low DFR. The upper right plot in Figure 7.4 shows that the different public key kinds have little effect if both wl and sk are faulted.

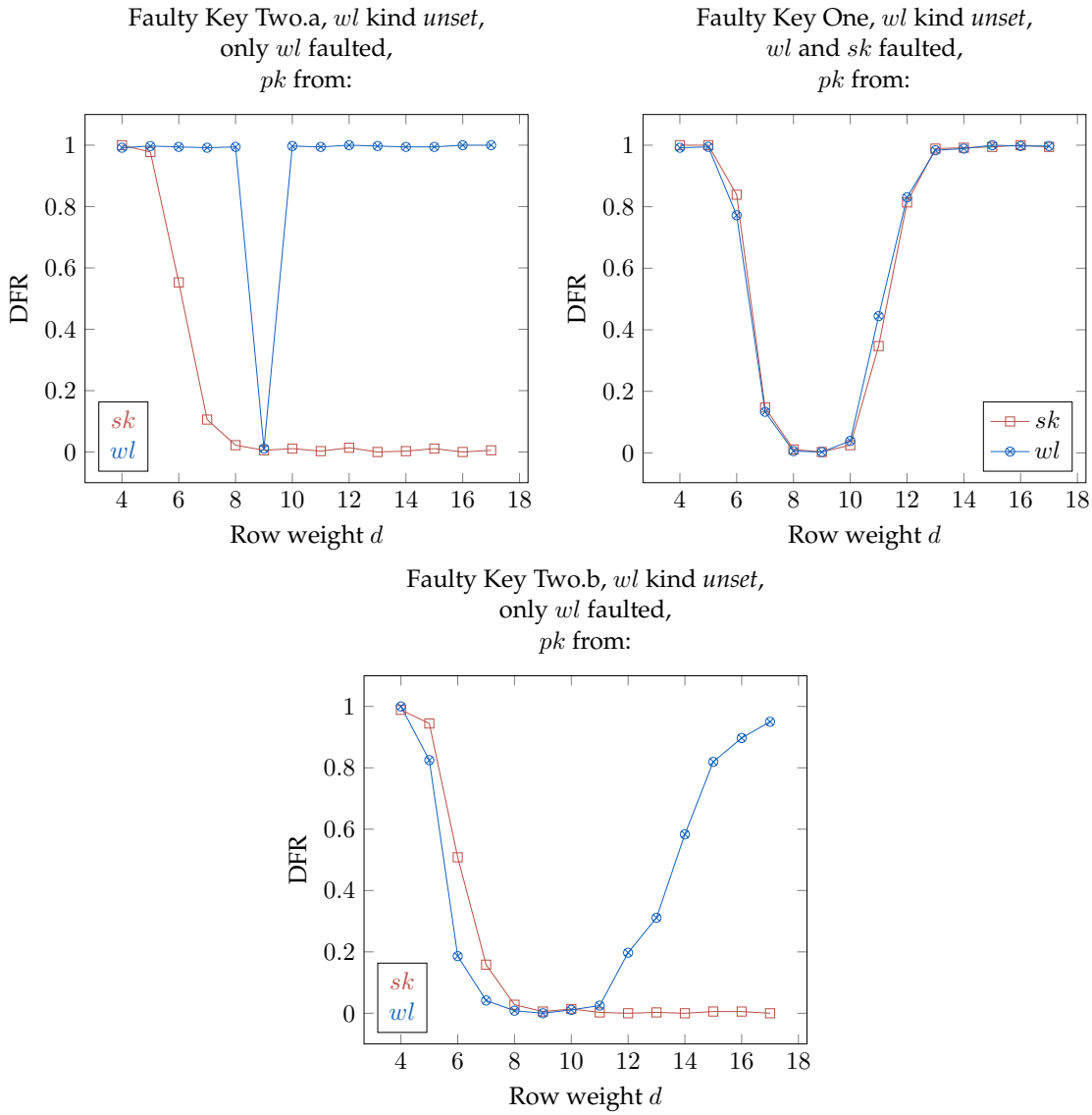

Figure 7.4: Impact of public key derivation. While only wl kind *Unset* is displayed, the same behavior can be observed for wl kind *multi*.

Fault Target For the different fault targets we can observe a similar behavior like for the different public key types. Figure 7.5 shows relatively little effect as long as we stick with the same Faulty Key Type.

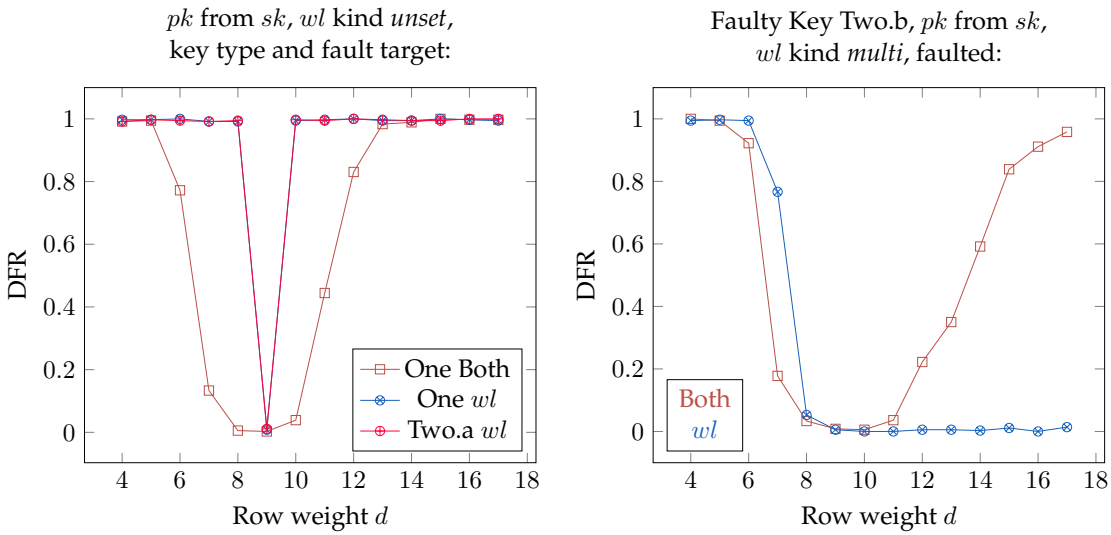

Figure 7.5: Impact of different Fault Targets.

The overall observation is, that we can verify the results presented by [Ket22]. In most cases we have an upside down hat shaped plot, which correlates with the DFR plots given by [Ket22]. Nevertheless we could show that different fault modes cause different behavior of the DFR. In the end we have to see which faulty keys we can cause to be generated with fault injection.

7.2 Analyzing BIKE

Before we start the actual attack, we need to figure out exactly which routines to attack. Knowing this, we analyze the pqm4 implementation and determine code sequences and/or processor instructions we want to fault.

7.2.1 Attack Outline

While Section 3.2 already describes the attack in general, here we focus on the fault injection to either generate faulty keys or fault the threshold calculation. This fault injection is supposed to enable the key recovery attack.
The keys need to be faulted already at generation time, otherwise *encapsulation* and *decapsulation* would not match or one would have to reproduce the exact same errors at least

twice, which is almost impossible.
As a scenario for our attack, we assume **KeyGen** and **Decaps** are performed by our victim while we calculate **Encaps**, compare to Algorithm 3. So we only get information about the success or failure of decapsulation while we know (from our own calculation) the random message m and the error vector $e = (e_0, e_1)$. Algorithm 4 gives an overview of the attack described by [Ket22], highlighting the different steps and calculation parties.

Algorithm 4: *Attack scheme*: *steps in this style* are executed by the victim, **steps in this style** are supposed to be faulted, every other steps are executed by the attacker.

---

1. *crypto_kem_keypair*: **generate faulty keys**
2. crypto_kem_enc:
   - generate random message m
   - calculate error vectors $(e_0, e_1) = \mathbf{H}(m)$
   - encrypt error vectors $c = (e_0 + e_1 h, \quad m \oplus \mathbf{L}(e_0, e_1))$
3. *crypto_kem_dec*:
   - **decode c to get the error vectors $(e'_0, e'_1) = e'$**
   - *decapsulate the message $m' = c_1 \oplus \boldsymbol{L}(e')$*
   - *check whether e' equals $\boldsymbol{H}(m')$ or not*
4. *share the information whether decapsulation was successful or not*
5. calculate distance spectrum of e_0 and e_1
6. for every distance ρ in the error vectors increase the counter $b[\rho]$
7. for every erroneous decapsulation increase the counter $a[\rho]$ for every distance ρ in the error vectors
8. after a sufficient amount of repetitions of steps 2. to 7. with the same keypair the probabilistic distance spectrum s of the secret key can be calculated:
   for every distance ρ $s[\rho] = \frac{a[\rho]}{b[\rho]}$
9. from the probabilistic distance spectrum s the key can be recovered as described in [GJS16]

7.2.2 Analyze Implementation

First we focus on the key generation, then we will analyze the decapsulation.

Key Generation

In Listing 1, we can see that the secret key is generated by calling `generate_sparse_rep` once for h_0 and once for h_1. In `generate_sparse_rep` two methods are called, `generate_indices_mod_z` and `secure_set_bits`, see Listing 4. The former generates a list of distinct indices modular to a given integer z in a simple loop, the latter sets the bits accordingly to the list of indices.
For `generate_indices_mod_z`, Listing 5 shows the c code. Here we could try to fault the loop condition to leave the loop early. Repeating the loop more often would not have a foreseeable effect as it might either result in a buffer overflow and manipulate other data on the stack or overwrite an existing index.
The source of `secure_set_bits` is given in Listing 6. This method seems more complex compared to `generate_indices_mod_z`, as it contains one simple for loop and a nested for loop. Hence this results in multiple potential points of fault injections.

Decapsulation

In the decapsulation routine, we want to fault the threshold. It takes some function calls to the threshold calculation: `decapsulation` (Listing 3) calls `decode` (Listing 7) and `decode` calls `get_threshold`. Listing 8 shows the c code of `get_threshold`. In `get_threshold`, the syndrome weight is calculated by calling `r_bits_vector_weight`. Based on this and the threshold coefficients the threshold is computed. As shown in Listing 9, the weight is calculated in a loop iterating over the bytes of the syndrome. Thus, we could target the loop summing up the vector's weight or the instructions computing the final threshold.

This makes a total of four approaches to inject faults, at `generate_indices_mod_z` and `secure_set_bits` for key generation, while the latter offers several opportunities and `get_threshold` and `r_bits_vector_weight` for decapsulation.

7.3 Injecting Faults

Due to a matter of complexity of the preparations, we did not fully investigate all possible attack vectors we described earlier. However, we were successful with both, faulting key generation and faulting decapsulation, resulting in an increased DFR. For fault injection, we were working with security level 11.

In Section 5.3 we discussed the firmware independent glitch settings, now we give an overview of the firmware dependent settings. Afterwards, we will explain the settings leading to a desired behavior with our setup.

7.3.1 Firmware dependant Glitch Settings

For our approach, we focus on three settings, namely *glitch.repeat*, *external offset* and *glitch.trigger_src*.
As the name indicates, *repeat* states for how many clock cycles the glitch injection will be repeated. Here, the clock cycles depend on the analog digital converter (ADC) clock. The ADC can have various clock sources, we chose to use the same clock as the target.
The *external offset* allows to wait for the first glitch to be injected multiple clock cycles after the occurrence of the trigger event. Here, the clock cycles depend on the ADC clock, too.
For *trigger source* various modes can be chosen. With the available hardware, only two modes have to be considered. The first one, *ext_single*, is the default mode. It states that the capture board waits for an external signal (pulling the trigger pin high) and afterwards injecting glitches according to the previous settings once. For this to happen, the capture board has be armed first with `scope.arm()`. The other mode is *ext_continuous*. Here, the capture board does not need to be armed and glitches will be injected each time the trigger condition is satisfied. According to the documentation, among others, *external offset* and *repeat* have no effect for this mode, but in practice we found they still do. This enables us to inject faults multiple times within a single run of the routine, which is very useful for loops.

Obviously these settings depend on the triggers which are set by the target's firmware. To allow reproduction of these fault injections, we have to give both, the glitch settings and the position of the triggers.

Glitch Settings for Key Generation

We have had success with faulting the first loop inside the `secure_set_bits` routine, lines 20 to 24 in Listing 6. Listing 7.1 shows the loop after inserting the triggers, while Table 7.1 shows the related glitch settings.

Listing 7.1: Triggers in secure_set_bits

```
1   for(size_t i = 0; i < w_size; i++) {
2     trigger_high();
3     int32_t w  = wlist[i] - first_pos;
4     pos_qw[i]  = w >> 6;
5     pos_bit[i] = BIT(w & MASK(6));
6     trigger_low();
7     }
```

glitch kind	clock
width	12.5
offset	$(-49, -41)$
trigger source	*ext_contiunous*
external offset	(34, 35)
repeat	2

Table 7.1: Glitch Settings for *secure_set_bits*

Glitch Settings for Decapsulation

For the decapsulation, we have had success with faulting the decoder. While detecting promising settings for the spots mentioned above in `get_threshold` and `r_bits_vector_weight`, we found very reliable settings for another spot of the decoder. We placed a `trigger_high()` right before the threshold calculation in line 8 in Listing 8. With the glitch settings in Table 7.2, we achieve a DFR in a desired range.

glitch kind	clock
width	12.5
offset	$(-49, -45)$
trigger source	*ext_single*
external offset	(637)
repeat	1

Table 7.2: Glitch Settings for *decode*

7.3.2 Results

The impact on *key generation* is quite intuitive, as we fault setting the bits in the vector representation, the weight list is not affected. As the implementation does not provide any other way, the public key is always derived from the vector representation of the secret key.
While we get some faulted keys with a weight for h_0 and h_1 of about 30, we get keys with a weight for h_0 and h_1 in the range of 6 to 8, as well. The former is too high and we end up with a DFR of 100 percent, which is useless in the matter of our attack. Whereas the Latter is exactly what we want, as it causes DFRs in between 1 and 99 percent.
Because we fault both h_0 and h_1 and the final weight often differs between them, we seldom generate straight *two.a* or *two.b* keys. There are some type *one* keys, but most of them do neither match Definition 3.3 nor Definition 3.4. The analyzer correctly recognizes that the *fault target* never is *both*. For the *weight list kind* some times the analysis states *unset* as expected, but often it is categorized as *undef*. This reveals that there is potential to improve the key analysis.

As stated in Section 7.3.1, in *decapsulation* we faulted a part of the algorithm that previously did not catch our attention. Which effect is actually caused has to be studied in further work. What we can claim here is that this fault injection causes reliably a DFR in the range of 10 to 90 percent for different keys. This is perfect for the key recovery attack described in Chapter 3.
One guess is that we cause the decoder to make extra iterations. The cycle count while the trigger pin is high increases by a factor of about 1.4 with fault injection compared to without fault injection. While the decoder makes at most five iterations in the given implementation, this could indicate that there are two extra iterations. But there has to be an additional fault, because more decoding iterations should not cause the DFR to raise but to fall. The maximum of iterations is set to five, as higher values would cause additional computation with little decoding improvement, not for security reasons.
The problem about faulting the decoder is to achieve a DFR which is neither to close too 100 percent nor too close 0 percent. In both cases it would be impossible to learn the distance spectrum of a key. Especially when setting the triggers inside `r_bits_vector_weight`, we were confronted with a DFR of either 100 percent or 0 percent. This is caused by the use of `r_bits_vector_weight` in multiple places. Hence, the glitches unintendedly affect other subroutines, too. To answer this, one could elaborate setting triggers, e.g, by implementing the triggers to be set conditionally.

8 Conclusions

In this thesis we studied the feasibility of the key recovery attack described by [Ket22] with BIKE running on a physical target. This attack is either based on faulted keys or on a faulted threshold. As a matter of achieving these faults, we chose the ChipWhisperer-Lite ARM board as hardware and the pqm4 implementation of the BIKE algorithm as software.

8.1 Summary

In Chapter 2, we give an overview of the BIKE algorithm and highlight the aspects of it with the highest relevance to the key recovery attack described in Chapter 3. As fault injection is the underlying idea of this key recovery attack, we give a definition of fault injection in Chapter 4. Additionally, we show properties of fault injection that help us to distinguish fault injections and classify our own approach.
As hardware we use the ChipWhisperer-Lite with an ARM target to inject faults. We chose this board because it is widespread, well documented, offers low level fault injection, is quite cheap and provides an easy to handle python interface to control it. While there is almost no limit of effort about injecting faults, choosing this hardware is a decision to a less powerful attack. Hence, our approach reveals a larger threat. In Chapter 5 we describe the hardware in detail and give an introduction to another target board, which was very useful for debugging. We analyze the firmware independent settings and state the most promising intervals there, too.
At the beginning of the research, we thought the pqm4 project provides a plug and play experience with our hardware and the BIKE algorithm, as pqm4 states to support our target board. This was absolutely not the case, as we point out in Chapter 6. To deal with the repository's complexity and interdependence was a milestone for this thesis and achieved in several steps. Besides the challenges we did not see coming, it was clear that we would have to extend the api for the communication with the target. Namely, we did append two firmware source files, to run KATs and an interactive firmware. For the latter, we have had to implement receiving data on the targets. This was easy for the hardware targets cw308t-stm32f3 and stm32f4discovery and we extended it to include the virtualized mps2-an386, too. The target mps2-an386 was very helpful for generating data, such as DFR for different fault modes, as it computes way faster than the hardware targets and at the same time their behavior does not differ. This might be helpful for related work,

too.
Besides extending the target's api, we did develop the counter part of the api on the host side and tools to handle the data in python. Chapter 7 explains the tools we did develop. Here, we emphasize two aspects. The first one is that we can distinguish at least 30 fault modes, resulting from faulting different parts of the secret key. In Section 7.1.4, we did show that different fault modes cause different behavior of the DFR when faulting keys according to the fault mode. We did discuss the cause of the different behavior we observed, but this could further be researched.
The second aspect is that we did develop a tool to analyze previously faulted keys. This leads us straight to the core of the thesis. We were able to inject faults in booth routines, *key generation* and *decapsulation*. The former could be faulted, such that the generated keys lead to a suitable DFR. The keys we caused to be generated did marginally match Definition 3.3 and Definition 3.4. To achieve this, one could further advance setting triggers in the given setup to improve fault injection. The latter faulted the decoder, leading to a suitable DFR, too, but this is key independent. In Section 7.3 the details for the fault injections are presented and the outcome is discussed.

All in all we can state that it is possible to enable the key recovery attacks introduced by [Ket22] via fault injection. For key generation, a successful attack seemed realistic. Whereas the difficulty about faulting the threshold is, to reliably achieve a DFR in between 1 and 0, as the extreme values are useless. Nevertheless we did show that it is possible to successfully fault both.

8.2 Further Research

While we could proof the concept of Faulty Keys and a faulted threshold with the ChipWhisperer-Lite ARM, this method depends on access to alter the target's firmware. We have to set triggers, to mark when to start the glitches. With more advanced hardware like the ChipWhisperer Pro or the ChipWhisperer Husky, the need to set triggers in the firmware gets obsolete, while it is still possible. Those capture boards allow to send the oscilloscope's output stream straight to the host. This enables the attacker to analyze power consumption in realtime and define consumption patterns as triggers.
Another aspect would be to use other targets for fault injection. For the cw308t-stm32f3, we had to massively lower the security level's parameters to get it working. One could use the stm32f4discovery to run BIKE level 1 and connect it to the CW-Lite (or other capture hardware). It might be tricky to set it up for fault injection, as stm32f4discovery is powered with 5V while CW-Lite is only powered with 3.3V. For clock glitching some solder bridges of stm32f4discovery have to be altered in advance to use an external clock (from the capture board). At the same time stm32f4discovery has a mechanism to detect irregularity on the external clock and switch back to the internal one. This mechanism might be turned off by disabling an interrupt in the targets firmware.
In general the introduced fault injection methods could further be advanced and analyzed. While we only took advantage of clock glitches it might be possible to use voltage glitches as well. This would decrease the power needed by an attacker and by that increase the threat. As for most devices clock glitching is only possible with access to the chip's pins, voltage glitching can be realized by a manipulated power supply.

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. Status report on the third round of the nist post-quantum cryptography standardization process. Technical report, NIST, 2022. https://csrc.nist.gov/publications/detail/nistir/8413/final.

[AGB+21] Nicolas Aragon, Tim Güneysu, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. Bike: Bit flipping key encapsulation, 2021. https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf.

[Bal15] Josep Balasch. introduction to fault attacks. lecture script, 2015. https://www.cs.ucr.edu/~nael/ee260/lectures/fault-injection.pdf.

[Ber21] Mladen Berekovic. Zuverlässigkeit von rechensystemen, kapitel 2. lecture script, 2021. Universität zu Lübeck.

[DGKP20] Nir Drucker, Shay Gueron, Dusan Kostic, and Edoardo Persichetti. On the applicability of the fujisaki-okamoto transformation to the bike kem. Cryptology ePrint Archive, Paper 2020/510, 2020. https://eprint.iacr.org/2020/510.

[EGVL+19] Luis Entrena, Mario García-Valderas, Almudena Lindoso, Marta Portela-Garcia, and Enrique San Millán. *Fault Injection Methodologies*, pages 127–144. Springer International Publishing, Cham, 2019. https://doi.org/10.1007/978-3-030-04660-6_6.

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on mdpc with cca security using decoding errors. In *Advances in Cryptology – ASIACRYPT 2016*, pages 789–815, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-53887-6_29.

[Inc] NewAE Technology Inc. *CW1173: ChipWhisperer-Lite*. https://media.newae.com/datasheets/NAE-CW1173_datasheet.pdf.

[Ket22] Sophie Ketelsen. Fault attacks on bike. Master's thesis, Universität zu Lübeck, 2022.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[Lim] ARM Limited. *Application Note AN386*. https://documentation-service.arm.com/static/5ed1094dca06a95ce53f8a9f?token=.

[Mea] Rafael Misoczki and et al. Bike suite. https://bikesuite.org/.

[Newa] NewAE. Chipwhisperer-lite 32-bit. https://www.newae.com/products/NAE-CWLITE-ARM.

[Newb] NewAE. Chipwhisperer software documentation. https://chipwhisperer.readthedocs.io/en/latest/.

[NIS] NIST. Nist round 4 submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[PETNH] Karl Palsson, Piotr Esden-Tempski, Fergus Noble, and Uwe Hermann. libopencm3: aims to create an open-source firmware library for various ARM Cortex-M microcontrollers. https://github.com/libopencm3/libopencm3.

[STMa] STMicroelectronics. *STM32F303xB STM32F303xC*. https://www.st.com/resource/en/datasheet/stm32f303rc.pdf.

[STMb] STMicroelectronics. *UM1472*. https://www.st.com/resource/en/user_manual/um1472-discovery-kit-with-stm32f407vg-mcu-stmicroelectronics.pdf.

[SV20] Nicolas Sendrier and Valentin Vasseur. On the existence of weak keys for qc-mdpc decoding. Cryptology ePrint Archive, Paper 2020/1232, 2020. https://eprint.iacr.org/2020/1232.

[Vas21] Valentin Vasseur. *Post-quantum cryptography: a study of the decoding of QC-MDPC codes*. PhD thesis, Université de Paris, 2021. https://theses.hal.science/tel-03254461.

Listings

In this chapter one can find listings of source code and assembly dump of functions that are interesting in the matter of fault injection for this thesis.

1 C Source Code

The source code files are split up in the KEM Api and the internal functions 1.2. Besides the KEM api functions, every function listed is underlined.

1.1 KEM Api

Listing 1: KEM Key Pair

```
1  //
       ////////////////////////////////////////////////////////////////////////////
2  int crypto_kem_keypair(OUT unsigned char *pk, OUT unsigned char *sk)
3  {
4    DEFER_CLEANUP(aligned_sk_t l_sk = {0}, sk_cleanup);
5
6    // The secret key is (h0, h1),
7    // and the public key h=(h0^-1 * h1).
8    // Padded structures are used internally, and are required by the
9    // decoder and the gf2x multiplication.
10   DEFER_CLEANUP(pad_r_t h0 = {0}, pad_r_cleanup);
11   DEFER_CLEANUP(pad_r_t h1 = {0}, pad_r_cleanup);
12   DEFER_CLEANUP(pad_r_t h0inv = {0}, pad_r_cleanup);
13   DEFER_CLEANUP(pad_r_t h = {0}, pad_r_cleanup);
14
15   // The randomness of the key generation
16   DEFER_CLEANUP(seeds_t seeds = {0}, seeds_cleanup);
17
18   get_seeds(&seeds);
19   // A SHAKE_PRF state for the secret key
20   DEFER_CLEANUP(prf_state_t h_prf_state = {0}, clean_shake256_prf_state);
21
22   GUARD(init_shake256_prf_state(&h_prf_state, MAX_PRF_INVOCATION, &seeds.seed
         [0]));
23
```

```
24    // Generate the secret key (h0, h1) with weight w/2
25    GUARD(generate_sparse_rep(&h0, l_sk.wlist[0].val, &h_prf_state));
26    GUARD(generate_sparse_rep(&h1, l_sk.wlist[1].val, &h_prf_state));
27
28    // // Generate sigma
29    // convert_seed_to_m_type(&l_sk.sigma, &seeds.seed[1]);
30
31    // // Calculate the public key
32    // gf2x_mod_inv(&h0inv, &h0);
33    // gf2x_mod_mul(&h, &h1, &h0inv);
34
35    // Fill the secret key data structure with contents - cancel the padding
36    l_sk.bin[0] = h0.val;
37    l_sk.bin[1] = h1.val;
38    l_sk.pk     = h.val;
39
40    (void) pk;
41    // Copy the data to the output buffers
42    bike_memcpy(sk, &l_sk, sizeof(l_sk));
43    // bike_memcpy(pk, &l_sk.pk, sizeof(l_sk.pk));
44
45    // print("h:  ", (uint64_t *)&l_sk.pk, R_BITS);
46    // print("h0: ", (uint64_t *)&l_sk.bin[0], R_BITS);
47    // print("h1: ", (uint64_t *)&l_sk.bin[1], R_BITS);
48    // print("h0 wlist:", (uint64_t *)&l_sk.wlist[0], SIZEOF_BITS(
          compressed_idx_d_t));
49    // print("h1 wlist:", (uint64_t *)&l_sk.wlist[1], SIZEOF_BITS(
          compressed_idx_d_t));
50    // print("sigma: ", (uint64_t *)l_sk.sigma.raw, M_BITS);
```

Listing 2: KEM Encapsulation

```
1  }
2
3  // Encapsulate - pk is the public key,
4  //               ct is a key encapsulation message (ciphertext),
5  //               ss is the shared secret.
6  int crypto_kem_enc(OUT unsigned char *     ct,
7                     OUT unsigned char *     ss,
8                     IN const unsigned char *pk)
9  {
10   // Public values (they do not require cleanup on exit).
11   pk_t l_pk;
12   ct_t l_ct;
13
14   DEFER_CLEANUP(m_t m, m_cleanup);
```

```
15   DEFER_CLEANUP(ss_t l_ss, ss_cleanup);
16   DEFER_CLEANUP(seeds_t seeds = {0}, seeds_cleanup);
17   DEFER_CLEANUP(pad_e_t e, pad_e_cleanup);
18
19   // Copy the data from the input buffer. This is required in order to avoid
20   // alignment issues on non x86_64 processors.
21   bike_memcpy(&l_pk, pk, sizeof(l_pk));
22
23   get_seeds(&seeds);
24
25   // e = H(m) = H(seed[0])
26   convert_seed_to_m_type(&m, &seeds.seed[0]);
27   GUARD(function_h(&e, &m));
28
29   // Calculate the ciphertext
30   GUARD(encrypt(&l_ct, &e, &l_pk, &m));
31
32   // Generate the shared secret
33   GUARD(function_k(&l_ss, &m, &l_ct));
34
35   print("ss:␣", (uint64_t *)l_ss.raw, SIZEOF_BITS(l_ss));
36
37   // Copy the data to the output buffers
38   bike_memcpy(ct, &l_ct, sizeof(l_ct));
39   bike_memcpy(ss, &l_ss, sizeof(l_ss));
```

Listing 3: KEM Decapsulation

```
1  }
2
3  // Decapsulate - ct is a key encapsulation message (ciphertext),
4  //               sk is the private key,
5  //               ss is the shared secret
6  int crypto_kem_dec(OUT unsigned char *     ss,
7                     IN const unsigned char *ct,
8                     IN const unsigned char *sk)
9  {
10   // Public values, does not require a cleanup on exit
11   ct_t l_ct;
12
13   DEFER_CLEANUP(seeds_t seeds = {0}, seeds_cleanup);
14
15   DEFER_CLEANUP(ss_t l_ss, ss_cleanup);
16   DEFER_CLEANUP(aligned_sk_t l_sk, sk_cleanup);
17   DEFER_CLEANUP(e_t e, e_cleanup);
18   DEFER_CLEANUP(m_t m_prime, m_cleanup);
```

```
19    DEFER_CLEANUP(pad_e_t e_tmp, pad_e_cleanup);
20    DEFER_CLEANUP(pad_e_t e_prime, pad_e_cleanup);
21
22    // Copy the data from the input buffers. This is required in order to avoid
23    // alignment issues on non x86_64 processors.
24    bike_memcpy(&l_ct, ct, sizeof(l_ct));
25    bike_memcpy(&l_sk, sk, sizeof(l_sk));
26
27    // Generate a random error vector to be used in case of decoding failure
28    // (Note: possibly, a "fixed" zeroed error vector could suffice too,
29    // and serve this generation)
30  //  get_seeds(&seeds);
31    //GUARD(generate_error_vector(&e_prime, &seeds.seed[0]));
32  //  generate_error_vector(&e_prime, &seeds.seed[0]);
33    memset( &e_prime, 0 , sizeof(e_prime) );
34
35    // Decode and on success check if |e|=T (all in constant-time)
36    volatile uint32_t success_cond = (decode(&e, &l_ct, &l_sk) == SUCCESS);
37  //  success_cond &= secure_cmp32(T, r_bits_vector_weight(&e.val[0]) +
38  //                                  r_bits_vector_weight(&e.val[1]));
39
40    // Set appropriate error based on the success condition
41    uint8_t mask; // = ~secure_l32_mask(0, success_cond);
42    for(size_t i = 0; i < R_BYTES; i++) {
43  //    PE0_RAW(&e_prime)[i] &= u8_barrier(~mask);
44  //    PE0_RAW(&e_prime)[i] |= (u8_barrier(mask) & E0_RAW(&e)[i]);
45  //    PE1_RAW(&e_prime)[i] &= u8_barrier(~mask);
46  //    PE1_RAW(&e_prime)[i] |= (u8_barrier(mask) & E1_RAW(&e)[i]);
47      PE0_RAW(&e_prime)[i] = E0_RAW(&e)[i];
48      PE1_RAW(&e_prime)[i] = E1_RAW(&e)[i];
49    }
50
51    GUARD(reencrypt(&m_prime, &e_prime, &l_ct));
52
53    // Check if H(m') is equal to (e0', e1')
54    // (in constant-time)
55    GUARD(function_h(&e_tmp, &m_prime));
56    success_cond = secure_cmp(PE0_RAW(&e_prime), PE0_RAW(&e_tmp), R_BYTES);
57    success_cond &= secure_cmp(PE1_RAW(&e_prime), PE1_RAW(&e_tmp), R_BYTES);
58
59    // Compute either K(m', C) or K(sigma, C) based on the success condition
60    mask = secure_l32_mask(0, success_cond);
61    for(size_t i = 0; i < M_BYTES; i++) {
62      m_prime.raw[i] &= u8_barrier(~mask);
63      m_prime.raw[i] |= (u8_barrier(mask) & l_sk.sigma.raw[i]);
64    }
```

```
65
66    // Generate the shared secret
67    GUARD(function_k(&l_ss, &m_prime, &l_ct));
68
69    // Copy the data into the output buffer
70    bike_memcpy(ss, &l_ss, sizeof(l_ss));
71
72    return SUCCESS;
73  }
```

1.2 BIKE Internal

First we can see the methods related to key generation, later those for decapsulation can be found.

Key Generation

Listing 4: generate_sparse_rep()

Listing 5: generate_indices_mod_z()

Listing 6: secure_set_bits()

```
1  void secure_set_bits(OUT pad_r_t *   r,
2                       IN const size_t first_pos,
3                       IN const idx_t *wlist,
4                       IN const size_t w_size)
5  {
6    assert(w_size <= MAX_WLIST_SIZE);
7
8    // Ideally we would like to cast r.val but it is not guaranteed to be
         aligned
9    // as the entire pad_r_t structure. Thus, we assert that the position of val
10   // is at the beginning of r.
11   bike_static_assert(offsetof(pad_r_t, val) == 0, val_wrong_pos_in_pad_r_t);
12   uint64_t *a64 = (uint64_t *)r;
13   uint64_t  val, mask;
14
15   // The size of wlist can be either D or T. So, we set it to max(D, T)
16   uint32_t pos_qw[MAX_WLIST_SIZE];
17   uint64_t pos_bit[MAX_WLIST_SIZE];
18
```

```
19    // Identify the QW position of every value, and the bit position inside this
          QW.
20    for(size_t i = 0; i < w_size; i++) {
21      int32_t w  = wlist[i] - first_pos;
22      pos_qw[i]  = w >> 6;
23      pos_bit[i] = BIT(w & MASK(6));
24    }
25
26    // Fill each QW in constant time
27    for(size_t i = 0; i < (sizeof(*r) / sizeof(uint64_t)); i++) {
28      val = 0;
29      for(size_t j = 0; j < w_size; j++) {
30        mask = (-1ULL) + (!secure_cmp32(pos_qw[j], i));
31        val |= (pos_bit[j] & mask);
32      }
33      a64[i] = val;
34    }
35  }
```

Decapsulation

Listing 7: decode()

```
1  ret_t decode(OUT e_t *e, IN const ct_t *ct, IN const sk_t *sk)
2  {
3    DEFER_CLEANUP(e_t black_e = {0}, e_cleanup);
4    DEFER_CLEANUP(e_t gray_e = {0}, e_cleanup);
5
6    DEFER_CLEANUP(pad_r_t c0 = {0}, pad_r_cleanup);
7    DEFER_CLEANUP(pad_r_t h0 = {0}, pad_r_cleanup);
8    pad_r_t pk = {0};
9
10   // Pad ciphertext (c0), secret key (h0), and public key (h)
11   c0.val = ct->c0;
12   h0.val = sk->bin[0];
13   pk.val = sk->pk;
14
15   DEFER_CLEANUP(syndrome_t s = {0}, syndrome_cleanup);
16   DMSG("  Computing s.\n");
17   GUARD(compute_syndrome(&s, &c0, &h0));
18   dup(&s);
19
20   // Reset (init) the error because it is xored in the find_err functions.
21   bike_memset(e, 0, sizeof(*e));
22
23   for(uint32_t iter = 0; iter < MAX_IT; iter++) {
24     const uint8_t threshold = get_threshold(&s);
25
26     DMSG("    Iteration: %d\n", iter);
27     DMSG("    Weight of e: %lu\n",
28          r_bits_vector_weight(&e->val[0]) + r_bits_vector_weight(&e->val[1]));
29     DMSG("    Weight of syndrome: %lu\n", r_bits_vector_weight((r_t *)s.qw));
30
31     find_err1(e, &black_e, &gray_e, &s, sk->wlist, threshold);
32     GUARD(recompute_syndrome(&s, &c0, &h0, &pk, e));
33 #if defined(BGF_DECODER)
34     if(iter >= 1) {
35       continue;
36     }
37 #endif
38     DMSG("    Weight of e: %lu\n",
39          r_bits_vector_weight(&e->val[0]) + r_bits_vector_weight(&e->val[1]));
40     DMSG("    Weight of syndrome: %lu\n", r_bits_vector_weight((r_t *)s.qw));
41
42     find_err2(e, &black_e, &s, sk->wlist, ((D + 1) / 2) + 1);
```

```
43      GUARD(recompute_syndrome(&s, &c0, &h0, &pk, e));
44
45      DMSG("    Weight of e: %lu\n",
46           r_bits_vector_weight(&e->val[0]) + r_bits_vector_weight(&e->val[1]));
47      DMSG("    Weight of syndrome: %lu\n", r_bits_vector_weight((r_t *)s.qw));
48
49      find_err2(e, &gray_e, &s, sk->wlist, ((D + 1) / 2) + 1);
50      GUARD(recompute_syndrome(&s, &c0, &h0, &pk, e));
51    }
52
53    if(r_bits_vector_weight((r_t *)s.qw) > 0) {
54      BIKE_ERROR(E_DECODING_FAILURE);
55    }
56
57    return SUCCESS;
58  }
```

Listing 8: get_threshold()

```
1   _INLINE_ uint8_t get_threshold(IN const syndrome_t *s)
2   {
3     bike_static_assert(sizeof(*s) >= sizeof(r_t), syndrome_is_large_enough);
4
5     const uint32_t syndrome_weight = r_bits_vector_weight((const r_t *)s->qw);
6
7     // The equations below are defined in BIKE's specification p. 16, Section
          5.2
8     uint32_t       thr  = THRESHOLD_COEFF0 + (THRESHOLD_COEFF1 * syndrome_weight
          );
9     const uint32_t mask = secure_l32_mask(thr, THRESHOLD_MIN);
10    thr = (u32_barrier(mask) & thr) | (u32_barrier(~mask) & THRESHOLD_MIN);
11
12    DMSG("    Threshold: %d\n", thr);
13    return thr;
14  }
```

Listing 9: r_bits_vector_weight()

```
1   uint64_t r_bits_vector_weight(IN const r_t *in)
2   {
3     uint64_t acc = 0;
4     for(size_t i = 0; i < (R_BYTES - 1); i++) {
5       acc += __builtin_popcount(in->raw[i]);
6     }
7
8     acc += __builtin_popcount(in->raw[R_BYTES - 1] & LAST_R_BYTE_MASK);
```

```
9   return acc;
10 }
```

2 Assembly Dump

Here we show the assembly dump of the internal functions, split in the same way as above. The dumps were taken from BIKE level 1 *opt*.

Key Generation

Listing 10: Assembly: generate_sparse_rep()

```
1  0000637c <generate_sparse_rep>:
2      637c: b570         push  {r4, r5, r6, lr}
3      637e: 460d         mov r5, r1
4      6380: 4613         mov r3, r2
5      6382: 4606         mov r6, r0
6      6384: f243 0223    movw  r2, #12323  ; 0x3023
7      6388: 2147         movs  r1, #71 ; 0x47
8      638a: 4628         mov r0, r5
9      638c: f7ff ff8c    bl  62a8 <generate_indices_mod_z>
10     6390: 4604         mov r4, r0
11     6392: b938         cbnz  r0, 63a4 <generate_sparse_rep+0x28>
12     6394: 4601         mov r1, r0
13     6396: 2347         movs  r3, #71 ; 0x47
14     6398: 462a         mov r2, r5
15     639a: 4630         mov r0, r6
16     639c: f000 f85a    bl  6454 <secure_set_bits>
17     63a0: 4620         mov r0, r4
18     63a2: bd70         pop {r4, r5, r6, pc}
19     63a4: f04f 34ff    mov.w r4, #4294967295 ; 0xffffffff
20     63a8: e7fa         b.n 63a0 <generate_sparse_rep+0x24>
```

Listing 11: Assembly: generate_indices_mod_z()

```
1  000062a8 <generate_indices_mod_z>:
2      62a8: e92d 4ff8    stmdb sp!, {r3, r4, r5, r6, r7, r8, r9, sl, fp, lr}
3      62ac: 4606         mov r6, r0
4      62ae: 4688         mov r8, r1
5      62b0: 4617         mov r7, r2
6      62b2: 469b         mov fp, r3
7      62b4: 2500         movs  r5, #0
8      62b6: f04f 0a01    mov.w sl, #1
9      62ba: 2300         movs  r3, #0
```

```
10      62bc: eb06 0985   add.w r9, r6, r5, lsl #2
11      62c0: 463a        mov r2, r7
12      62c2: 461c        mov r4, r3
13      62c4: ea52 0003   orrs.w  r0, r2, r3
14      62c8: f104 0101   add.w r1, r4, #1
15      62cc: d112        bne.n 62f4 <generate_indices_mod_z+0x4c>
16      62ce: b2e4        uxtb  r4, r4
17      62d0: fa0a f404   lsl.w r4, sl, r4
18      62d4: 3c01        subs  r4, #1
19      62d6: 2204        movs  r2, #4
20      62d8: 4659        mov r1, fp
21      62da: 4648        mov r0, r9
22      62dc: f7fe fd12   bl  4d04 <aes_ctr_prf>
23      62e0: b9e8        cbnz  r0, 631e <generate_indices_mod_z+0x76>
24      62e2: f8d9 2000   ldr.w r2, [r9]
25      62e6: 4022        ands  r2, r4
26      62e8: 4297        cmp r7, r2
27      62ea: f8c9 2000   str.w r2, [r9]
28      62ee: d9f2        bls.n 62d6 <generate_indices_mod_z+0x2e>
29      62f0: 4601        mov r1, r0
30      62f2: e00a        b.n 630a <generate_indices_mod_z+0x62>
31      62f4: 0852        lsrs  r2, r2, #1
32      62f6: ea42 72c3   orr.w r2, r2, r3, lsl #31
33      62fa: 460c        mov r4, r1
34      62fc: 085b        lsrs  r3, r3, #1
35      62fe: e7e1        b.n 62c4 <generate_indices_mod_z+0x1c>
36      6300: f856 3021   ldr.w r3, [r6, r1, lsl #2]
37      6304: 429a        cmp r2, r3
38      6306: d008        beq.n 631a <generate_indices_mod_z+0x72>
39      6308: 3101        adds  r1, #1
40      630a: 428d        cmp r5, r1
41      630c: d1f8        bne.n 6300 <generate_indices_mod_z+0x58>
42      630e: 2301        movs  r3, #1
43      6310: 441d        add r5, r3
44      6312: 4545        cmp r5, r8
45      6314: d3d1        bcc.n 62ba <generate_indices_mod_z+0x12>
46      6316: e8bd 8ff8   ldmia.w sp!, {r3, r4, r5, r6, r7, r8, r9, sl, fp, pc}
47      631a: 2300        movs  r3, #0
48      631c: e7f8        b.n 6310 <generate_indices_mod_z+0x68>
49      631e: f04f 30ff   mov.w r0, #4294967295 ; 0xffffffff
50      6322: e7f8        b.n 6316 <generate_indices_mod_z+0x6e>
```

Listing 12: Assembly: secure_set_bits()

```
1   00006454 <secure_set_bits>:
2       6454: e92d 43f0   stmdb sp!, {r4, r5, r6, r7, r8, r9, lr}
```

```
6458: 2b86        cmp r3, #134  ; 0x86
645a: f2ad 6d4c   subw  sp, sp, #1612 ; 0x64c
645e: d810        bhi.n 6482 <secure_set_bits+0x2e>
6460: 4614        mov r4, r2
6462: eb02 0e83   add.w lr, r2, r3, lsl #2
6466: ae86        add r6, sp, #536  ; 0x218
6468: 46ec        mov ip, sp
646a: 2701        movs  r7, #1
646c: 4574        cmp r4, lr
646e: d10e        bne.n 648e <secure_set_bits+0x3a>
6470: f1a0 0708   sub.w r7, r0, #8
6474: 2000        movs  r0, #0
6476: 2100        movs  r1, #0
6478: ac86        add r4, sp, #536  ; 0x218
647a: 46ee        mov lr, sp
647c: 460e        mov r6, r1
647e: 460d        mov r5, r1
6480: e031        b.n 64e6 <secure_set_bits+0x92>
6482: 4b1f        ldr r3, [pc, #124]  ; (6500 <secure_set_bits+0xac>)
6484: 4a1f        ldr r2, [pc, #124]  ; (6504 <secure_set_bits+0xb0>)
6486: 4820        ldr r0, [pc, #128]  ; (6508 <secure_set_bits+0xb4>)
6488: 2116        movs  r1, #22
648a: f000 fe07   bl  709c <__assert_func>
648e: f854 2b04   ldr.w r2, [r4], #4
6492: 1a52        subs  r2, r2, r1
6494: 1195        asrs  r5, r2, #6
6496: f002 023f   and.w r2, r2, #63 ; 0x3f
649a: f1c2 0820   rsb r8, r2, #32
649e: f84c 5b04   str.w r5, [ip], #4
64a2: f1a2 0520   sub.w r5, r2, #32
64a6: fa07 f505   lsl.w r5, r7, r5
64aa: fa27 f808   lsr.w r8, r7, r8
64ae: ea45 0508   orr.w r5, r5, r8
64b2: fa07 f202   lsl.w r2, r7, r2
64b6: e9c6 2500   strd  r2, r5, [r6]
64ba: 3608        adds  r6, #8
64bc: e7d6        b.n 646c <secure_set_bits+0x18>
64be: f85e 2b04   ldr.w r2, [lr], #4
64c2: e9d4 9800   ldrd  r9, r8, [r4]
64c6: 4282        cmp r2, r0
64c8: f04f 3cff   mov.w ip, #4294967295 ; 0xffffffff
64cc: bf18        it  ne
64ce: f04f 0c00   movne.w ip, #0
64d2: 4662        mov r2, ip
64d4: ea02 0208   and.w r2, r2, r8
64d8: ea0c 0c09   and.w ip, ip, r9
```

```
64dc: 3408        adds  r4, #8
64de: ea4c 0606   orr.w r6, ip, r6
64e2: 4315        orrs  r5, r2
64e4: 3101        adds  r1, #1
64e6: 4299        cmp r1, r3
64e8: d1e9        bne.n 64be <secure_set_bits+0x6a>
64ea: f847 6f08   str.w r6, [r7, #8]!
64ee: 3001        adds  r0, #1
64f0: f5b0 7f80   cmp.w r0, #256  ; 0x100
64f4: 607d        str r5, [r7, #4]
64f6: d1be        bne.n 6476 <secure_set_bits+0x22>
64f8: f20d 6d4c   addw  sp, sp, #1612 ; 0x64c
64fc: e8bd 83f0   ldmia.w sp!, {r4, r5, r6, r7, r8, r9, pc}
6500: 0000cda0  .word 0x0000cda0
6504: 0000cde8  .word 0x0000cde8
6508: 0000cdb9  .word 0x0000cdb9
```

Decapsulation

We did not include the whole assembly from `decode` because it is too long and we are only interested in a small part of it. Because `get_threshold` is only called from `decode` the compiler/linker does not create an extra function but includes the instructions from both in `decode`. We could determine that `get_threshold` results in the instructions in Listing 13. To give a rough orientation in the 272 lines of assembly dump from `decode`, this is around the first call of `r_bits_vector_weight`.

Listing 13: Assembly: get_threshold()

```
5226: 4630        mov r0, r6
5228: f001 f970   bl  650c <r_bits_vector_weight>
522c: f001 fe88   bl  6f40 <__aeabi_ui2d>
5230: a384        add r3, pc, #528  ; (adr r3, 5444 <decode+0x32c>)
5232: e9d3 2300   ldrd  r2, r3, [r3]
5236: f001 fc17   bl  6a68 <__aeabi_dmul>
523a: a384        add r3, pc, #528  ; (adr r3, 544c <decode+0x334>)
523c: e9d3 2300   ldrd  r2, r3, [r3]
5240: f001 fd42   bl  6cc8 <__adddf3>
5244: f001 fef6   bl  7034 <__aeabi_d2uiz>
5248: f04f 0800   mov.w r8, #0
524c: f1b0 0324   subs.w  r3, r0, #36 ; 0x24
5250: f148 33ff   adc.w r3, r8, #4294967295 ; 0xffffffff
5254: 43dd        mvns  r5, r3
5256: 4028        ands  r0, r5
5258: f003 0524   and.w r5, r3, #36 ; 0x24
525c: 4305        orrs  r5, r0
```

Listing 14: Assembly: r_bits_vector_weight()

```
0000650c <r_bits_vector_weight>:
    650c: e92d 41f0   stmdb sp!, {r4, r5, r6, r7, r8, lr}
    6510: 2400        movs  r4, #0
    6512: 4605        mov r5, r0
    6514: 1e47        subs  r7, r0, #1
    6516: f200 6803   addw  r8, r0, #1539 ; 0x603
    651a: 4626        mov r6, r4
    651c: f817 0f01   ldrb.w  r0, [r7, #1]!
    6520: f000 fda8   bl  7074 <__popcountsi2>
    6524: 1904        adds  r4, r0, r4
    6526: eb46 76e0   adc.w r6, r6, r0, asr #31
    652a: 4547        cmp r7, r8
    652c: d1f6        bne.n 651c <r_bits_vector_weight+0x10>
    652e: f895 0604   ldrb.w  r0, [r5, #1540] ; 0x604
    6532: f000 0007   and.w r0, r0, #7
    6536: f000 fd9d   bl  7074 <__popcountsi2>
    653a: 4601        mov r1, r0
    653c: 1900        adds  r0, r0, r4
    653e: eb46 71e1   adc.w r1, r6, r1, asr #31
    6542: e8bd 81f0   ldmia.w sp!, {r4, r5, r6, r7, r8, pc}
    6546: e7fe        b.n 6546 <r_bits_vector_weight+0x3a>
    6548: e7fe        b.n 6548 <r_bits_vector_weight+0x3c>
    654a: e7fe        b.n 654a <r_bits_vector_weight+0x3e>
    654c: e7fe        b.n 654c <r_bits_vector_weight+0x40>
    654e: e7fe        b.n 654e <r_bits_vector_weight+0x42>
    6550: e7fe        b.n 6550 <r_bits_vector_weight+0x44>
    6552: e7fe        b.n 6552 <r_bits_vector_weight+0x46>
    6554: e7fe        b.n 6554 <r_bits_vector_weight+0x48>
    6556: e7fe        b.n 6556 <r_bits_vector_weight+0x4a>
```