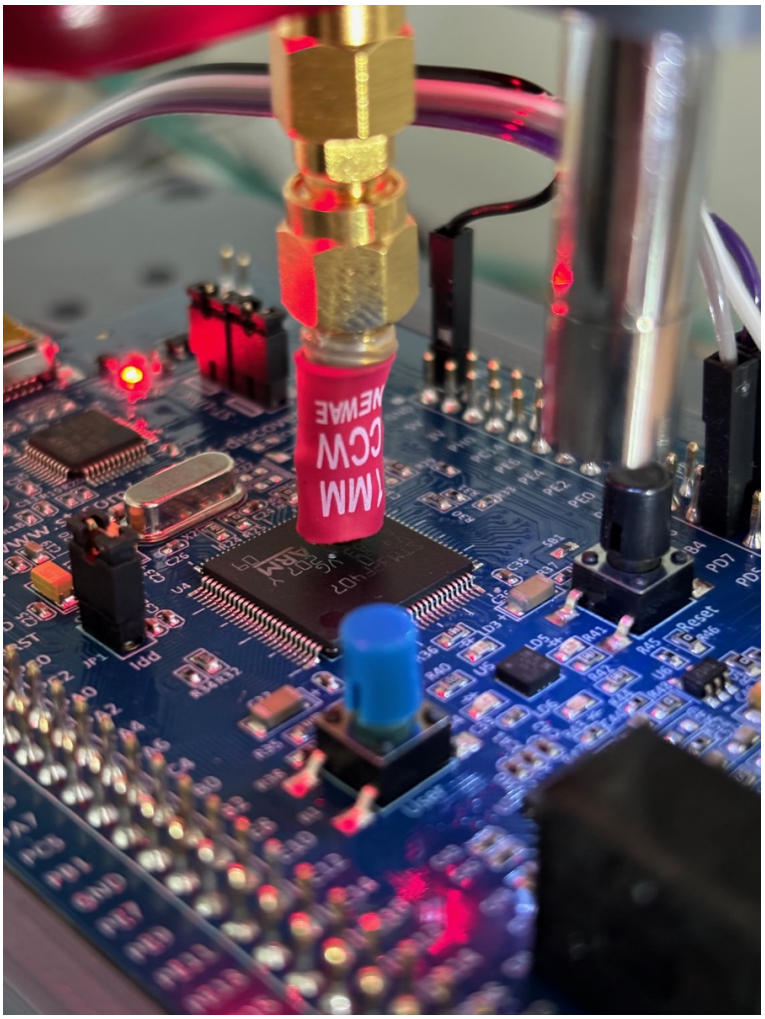
Case Study: I Want To Fault My BIKE

On the Feasibility of Electromagnetic Fault Injection Attacks Against The BIKE Cryptosystem

Jeremy Boy, Jack Mähl, Robin Sehm

2024-05-30



Contents

1 Introduction

This report documents the construction of an electromagnetic fault injection (EMFI) station, how to find suitable locations for injecting faults on the device-under-test (DUT) and a practical fault attack against the Bit Flipping Key Encapsulation (BIKE) framework. To achieve this, we first constructed an EMFI station that precisely moves a fault injection probe to the desired location on the chip.
To identify susceptible locations for fault injection (*probing*), a specialized firmware was developed to operate on the DUT. This firmware is designed to record register states both before and after a fault injection event. By analyzing the discrepancies between these states, the specific registers affected by the fault can be pinpointed.
Using the information which locations are most susceptible to faults, we slightly modified an implementation of the BIKE framework and used this for subsequent experiments. First, the information of the probing experiment was combined with the modified firmware to find susceptible locations specific to the targeted algorithm. After that, we started with the actual fault injection attack: The idea behind fault attacks on BIKE is to set bits during the secret key generation to reach a certain threshold. When this faulty key is then used for encapsulation, the security of the payload is compromised and the secret key can be recovered.
Experimental results show that, under certain conditions, faulting BIKE in a lab environment is possible, as we were able to effectively manipulate bits during secret key generation, nearly reaching the optimal threshold within a reasonable timeframe. These results underscore the vulnerability of the BIKE framework to fault injection attacks, highlighting potential security vulnerabilities that warrant further investigation and mitigation strategies.

1.1 Related Work

In [16], [14] the authors examined the influence of faults on the BIKE framework in a theoretical model. That is, it was assumed that fault attacks can reliably flip a large amount of bits, allowing the adversary to eventually recover the secret key.
In [17], the authors present the EMFI station and use their novel construction to successfully perform a fault injection attack against the AMD Secure Processor (AMD-SP). The EMFI station is based on components commonly found in 3D printers and uses motorized linear stages, which allow precise positioning of a fault injection probe. The system built during this case study is largely based on their original design of the EMFI station.
The authors of [15] implemented post-quantum cryptography (PQC) algorithms selected for standardization by NIST as part of the NIST PQC competition for several relevant ARM Cortex-M4 platforms.

1.2 Structure

This report discusses the feasibility of EMFI attack against the BIKE Key encapsulation mechanism running on the *STM32F4DISCOVERY* platform. Readers only interested in the getting started with the faulting station can skip until Section 4.
In Section 2, the post-quantum secure key encapsulation mechanism BIKE is introduced. The construction of BIKE is derived from quasi-cyclic linear codes and the Niederreiter cryptosystem.
After that, the Section 3 evaluates different approaches for key recovery attacks against the BIKE cryptosystem. Those attacks have been introduced in a previous work by Kettelsen [16]. A distinction is made between *weak* and *faulty* keys. Key recovery algorithms that abuse these key properties are also discussed. In Section 4, the general hardware setup and challenges we encountered are discussed. Section 5 presents our experiments, faulting station setup and build system. We introduce an experiment used to probe a DUT for vulnerable regions and an attack against the PQM4 [15] implementation of BIKE. The results of these experiments are described in detail in Section 6. We conclude this report in Section 7.

2 BIKE

The BIKE suite [5] is a code-based key-encapsulation mechanism designed to resist quantum attackers. BIKE was submitted to the *NIST Post-Quantum Cryptography Standardization Process* in November 2017[1], and its second round submission was accepted as a full submission in April 2019 [2]. In the third round, BIKE was listed as *alternate candidate* in the *public-key encryption and key-establishment algorithms* category [3]. BIKE was one of four entries to advance to the fourth round of the NIST competition [4].
BIKE implements the Niederreiter cryptosystem on quasi-cyclic moderate density parity check codes (*QC-MDPC codes*). The Niederreiter cryptosystem as well as the related McEliece cryptosystem are Indistinguishable for Chosen Plaintext Attacks (IND-CPA) secure and can be transformed into a Indistinguishable for Chosen Ciphertext Attacks (IND-CCA) secure cryptosystem using the Fujisaki-Okamoto transformation[10].
The following subsections will discuss the BIKE cryptosystem. We will start by briefly introducing cyclic codes, commencing with their generalization, quasi-cyclic codes.
Afterwards, we will discuss the McEliece and Niederreiter cryptosystems before finally introducing the BIKE. The following explanations are in part a synopsis of [7].

2.1 Linear Codes and (Quasi-)Cyclic Codes

Linear codes are a type of error-correcting code. These codes are called "linear" because they satisfy linearity. In a linear code, the sum of two codewords is also a valid codeword. For example, if we have two codewords 1010 and 0101 of a given code C, the bit-wise sum of these codewords 1111 is a code word as well. This property allows for efficient decoding of the received message.
Cyclic codes are a class of linear codes, where, given a code word $c = c_1c_2 \dots c_n$, then $\text{shift}(c) = c_nc_1c_2 \dots c_{n-1}$ is a code word as well. We can define a generator matrix G for a cyclic code C as

$$G = \begin{pmatrix} g_1 & g_2 & \dots & g_{r-1} & g_r & 0 & \dots & 0 & & \\ 0 & g_1 & g_2 & \dots & g_{r-1} & g_r & 0 & \dots & 0 & \\ 0 & 0 & g_1 & g_2 & \dots & g_{r-1} & g_r & 0 & \dots & 0 \\ & & \vdots & & & & & & & \\ 0 & 0 & \dots & & & 0 & g_1 & g_2 & \dots & g_r \end{pmatrix} \in \mathbb{F}^{k \times n}$$

with $k = n - r$. The linear code C is then defined as the product of vectors from $\mathbb{F}^k$ with the generator matrix:

$$C = \left\{ m \cdot G \mid m \in \mathbb{F}^k \right\} \subseteq \mathbb{F}^n.$$

Alternatively, we can define C by its parity check matrix H, which is a generator matrix of the dual code $C^\perp$:

$$C = \{ c \mid H \cdot c = 0 \} \subseteq \mathbb{F}^n.$$

Given a word $c' \in \mathbb{F}^n$, we can then decide if $c' \in C$ using *syndrome decoding*, i.e., check $H \cdot c' = 0$. With $c' = c + e$ with $c \in C$ and error e, we call $H \cdot c' = H \cdot (c' + e) = H \cdot c + H \cdot e = H \cdot e$ the *syndrome of* c. Observe that $c' \in C$ implies $H \cdot e = 0$.

Fact 1 *Given a generator matrix G (or parity check matrix H) and $c' = c + e$, finding e is NP-hard [6].*

Definition 1 (Quasi-Cyclic Codes) *Quasi-cyclic codes are a generalization of cyclic codes [13]. In contrast to cyclic codes, quasi-cyclic codes are invariant to shifts of length ℓ (note that, for cyclic codes, $\ell = 1$). The parity check matrix of a quasi-cyclic code can be expressed as*

$$H = \begin{pmatrix} H_0 & H_1 & \dots & H_{n-1} & H_n \end{pmatrix},$$

where the H_i are parity check matrices for cyclic codes C_i.

An interesting class of quasi-cyclic codes are those with moderately dense parity check matrices, which allow for fast syndrome decoding.

Definition 2 (QC-MDPC Codes) *Quasy-cyclic moderate density parity check codes (QC-MDPC codes) are quasi-cyclic codes with* moderate *density, i.e.*

$$\text{weight}(h_0) \in \mathcal{O}(\sqrt{n}),$$

$$\text{weight}(h_1) \in \mathcal{O}(\sqrt{n}),$$

where weight(h_i) *denotes the hamming weight of h_i and n is the code word length.*

2.2 McEliece Cryptosystem and Niederreiter Cryptosystem

In 1978, Robert McEliece developed the McEliece cryptosystem, an asymmetric encryption system that is inspired by the NP-hardness of the above-mentioned problem. For key generation, Alice samples a linear code ℓ able to recover t errors from a family of linear codes $\mathcal{L}$ with generator matrix $G \in \mathbb{F}^{k \times n}$ as well as permutation matrices $S \in \mathbb{F}^{k \times k}$ and $P \in \mathbb{F}^{n \times n}$. The public key then is

$$\mathsf{pk} = (\hat{G} = S \cdot G \cdot P)$$

and the private key is

$$\mathsf{sk} = (G, S, P).$$

After Alice sends pk to Bob, he can encrypt a message m by first calculating $c = m \cdot \hat{G}$ and sampling a random error e with $weight(e) \leq t$ to get the ciphertext $\hat{c} = c + e$. Because of the above-mentioned problem, finding e (and therefore c and m) given $c + e$ and $\hat{G}$ is hard. With knowledge of her private key sk though, Alice can calculate m given $\hat{c}$ by $c' = \hat{c} \cdot P^{-1}$, decoding c' w.r.t. G to get m', and calculate $m' \cdot S^{-1}$ to yield m.

Lemma 1 *The McEliece cryptosystem is secure against CPA attackers if*

- $\hat{G}$ *is indistinguishable from a random linear code and*
- $\hat{c} = c + e$ *is indistinguishable from a random vector.*

The Niederreiter cryptosystem works similarly to the McEliece cryptosystem. Instead of the generator matrix G, it uses the parity check matrix H with $\mathsf{pk} = (\hat{H} = S \cdot H \cdot P)$, $\mathsf{sk} = (S, H, P)$. With Niederreiter, we then encrypt an error e to a syndrome. The error e can then only be recovered if the secret key sk is known.

2.3 BIKE

BIKE is an implementation of a (modified) Niederreiter cryptosystem using QC-MDPC codes. In BIKE, Alice's private key consists of a matrix $H = \begin{pmatrix} h_0 & h_1 \end{pmatrix}$ where H is a $(2, 1, r, w)$-QC-MDPC code of length $n := 2r$, i.e., h_0, h_1 are $n \times n$ circular square block matrices. Additionally, a failure message σ is sampled at random. The private key then is

$$\mathsf{sk} = (h_0, h_1, \sigma).$$

Since the h_i are parity check matrices of cyclic codes, they are usually represented by their first row $h_i^{(1)} \in GF(2^r)$. We will use $h_i^{(1)}$ interchangeably with h_i. The public key is generated as $\mathsf{pk} = h = h_1 \cdot h_0^{-1}$.
Afterwards, using three hash functions $\mathbf{H}, \mathbf{L}, \mathbf{K}$, which are treated as random oracles in BIKE's security proofs, and public key $h = h_1 h_0^{-1}$, Bob can encapsulate a shared key k by first sampling a *random* message m from a message space $\mathcal{M}$. He then proceeds by calculating error vectors $(e_0, e_1) = \mathbf{H}(m)$ and ciphertext $c = (c_0, c_1) = (e_o + e_1 h, m \oplus \mathbf{L}(e_0, e_1)$. Bob then calculates the shared key by $k = \mathbf{K}(m, c)$ and sends $c = (c_0, c_1)$ to Alice. Here, c_0 is the encapsulation of error vectors e_0, e_1, which are pseudo-random due to hashing with the random oracle $\mathbf{L}$, while c_1 can later be used by Alice to verify her decoding result.
Alice, using her private key H and the ciphertext c, can recover the shared key k with high likelihood by first recovering an error vector

$$e' := (e_0, e_1)' = \mathit{decode}(c_0 h_0, (h_0, h_1))$$

from c using her knowledge of sk. For *decode*, c_0 is first mapped to

$$c_0 h_0 = (e_0 + e_1 h) h_0 = e_0 h_0 + e_1 h_1 = e'^T H.$$

The *decode* function then solves the syndrome decoding problem by finding a sparse error vector $e' = (e_0, e_1)'$ such that $e'^T H = e^T H$. After *decode* has recovered an error vector $e' = (e'_0, e'_1)$, Alice checks if she has the correct error vector by first calculating $m' = c_1 \oplus \mathbf{L}(e'_0, e'_1)$. If the decoding succeeded, $m' = m$ and subsequently $e' = \mathbf{H}(m')$ will hold. Alice then calculates the shared key $k = \mathbf{K}(m', (c_0, c_1))$. If not, Alice uses $k = \mathbf{K}(\sigma, c)$ as failure message in order to avoid leaking information about m. For a message sequence chart of the protocol, see Figure 1.

With high probability, the error vector generated by *decode* will be equal to e, compare [5]. The probability of *decode* recovering an error vector $e' \neq e$ is called the decoding failure rate (DFR). The DFR is what will be manipulated by a fault attack by forcing the use of weak or faulty keys in the protocol, see Section 3.

3 BIKE Key Recovery

We now explain how a fault injection can be used to recover the secret key that is used in BIKE. For the attack, we target h_0, h_1 from the secret key sk. The core idea is to manipulate h_0 or h_1 in such a way that the decoder fails every other round, i.e., to achieve an optimal DFR. This information whether the decoder failed together with a so-called distance spectrum of e can be used to recover the secret key. We show the top-level view of the attack in Figure 2. The function `Init` initializes the arrays used for the distance spectrum. The function `UpdateDistSpec` updates this spectrum based on bit distances in the generated error vector e. The other functions are explained in Section 2. The communication between the two parties is repeated N times until the spectrum is sufficient to recover the secret key. The probability of recovering sk for a given N has been studied in [14].

3.1 Key Properties and Faultability

There are two main types of secret keys that influence the error rate of the decoder, *faulty keys* and *weak keys*. Weak keys are in compliance to specification, i.e, they have the correct weight. The problem for the decoder stems from a bad roll of the random positions for the ones. A key is considered weak if large sets of ones are evenly spaced. However, the probability of such a distribution occurring for long secret keys when KGen is implemented according to the specification is very low. Faulty keys on the other hand are not according to specification. They have a lower or higher weight than the decoder expects.

Let us now discuss some ideas how we can fault KGen to return a weak or faulty key. Ketelsen[16] introduced the concept of efficiently forgeable weak keys (EFWK). It assumes a subset of the bits being stored in a register, and then all of those bits are faulted to one. We now have a block of f ones where f is the register size. In the reference implementation [8], one could target the `SET_BIT` function for this. Ketelsen assumed 32-bit registers, i.e., $f = 32$. However, as the current reference implementation of `SET_BIT` expects a `uint8*`, both Arm and x86 Compilers will instruct writing back 8 bit. In other words, every time the `SET_BIT` function is called, we can increase a block of ones in the secret key by a maximum of 8 bits (if the fault is successful). The big advantage over the previous approach is that we don't require a fault to generate a very specific value, however, we just try to set as many ones as possible. If we want to use the approach and get blocks larger than 8 bit, we still have to get lucky that the random positions lie close to each other. This approach seems to be the best for generating faulty keys, as we just need to increase (or decrease) the weight of sk. While easier than the previous approach,

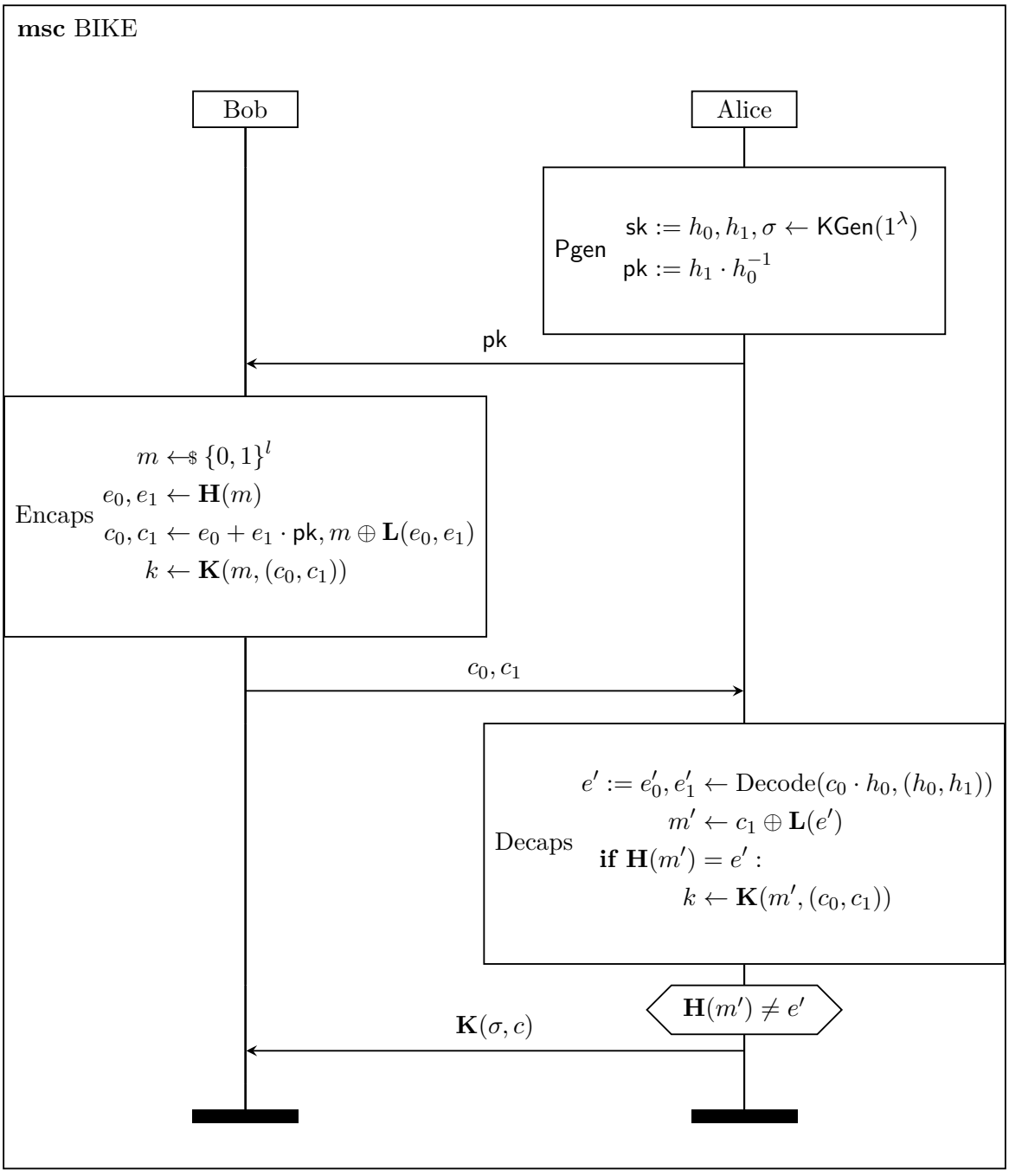


Figure 1: Message sequences of BIKE

it will still be hard to create EFWKs, as we need to also fault the termination condition of the for loop in `generate_sparse_rep_keccak`. This is because we now add more bits each iteration and still need to fulfill the weight requirement for the specification (by definition of a weak key). The rest of this document focuses on the PQM4 implementation. It handles key generation slightly differently. However, the basic idea that we inject faults within a subroutine of the KGen function similar to `SET_BIT` remains the same.

3.2 Key Recovery Algorithm

From Figure 2, it becomes clear that the adversary needs three helper methods to perform a key recovery. As explained in the introduction of this section, the secret key recovery algorithm requires the distances of ones in the error vector e_0, e_1 and whether the decoder was successful, i.e, if Alice does not send $\mathbf{K}(\sigma, c)$. If Alice used a non-faulty KGen, the DFR is almost negligible (as stated by Aragon et al. [5]). If we manage to increase the weight of h_0 or h_1, i.e, produce faulty keys, we can increase this DFR. Ketelsen [16] studied the influence of certain weights on the DFR. Their results indicate that there are certain weights that almost produce a 50% DFR. At such a decoding error rate, we need the least number of repetitions, i.e., N is then minimal. We now give a top-level explanation of the functions involved in the key recovery:

- `INIT` This function creates two helper arrays a and b of size r and sets all entries to 0. They are later needed to create the distance spectrum. Array a at position ρ contains how often a distance of ρ between the ones existed in e_0, e_1. We have $\rho \in \{1, \ldots, r-1\}$ where $\rho = 1$ indicates two ones directly next to each other. The array b at position ρ is incremented if decoding was unsuccessful and the distance ρ between ones was present at least once in the error vector e_0, e_1 of this round.

- `UpdateDistSpec` This function updates a and b as explained. The distance spectrum is a vector ds_i where i is the current round, i.e., $i \in \{1, \ldots, N\}$. Its entries are defined as

$$ds_i[\rho] = \begin{cases} \frac{a[\rho]}{b[\rho]} & \text{if } b[\rho] > 0 \\ 0, & \text{otherwise} \end{cases}$$

  for $\rho \in \{1, \ldots, r-1\}$

- `RecoverKey` Now, after N repetitions, we can use Algorithm 2 from [14] to recover a guess for sk by providing the distance spectrum ds_N. If N is sufficiently large, the algorithm will return the correct key shifted one to the right.

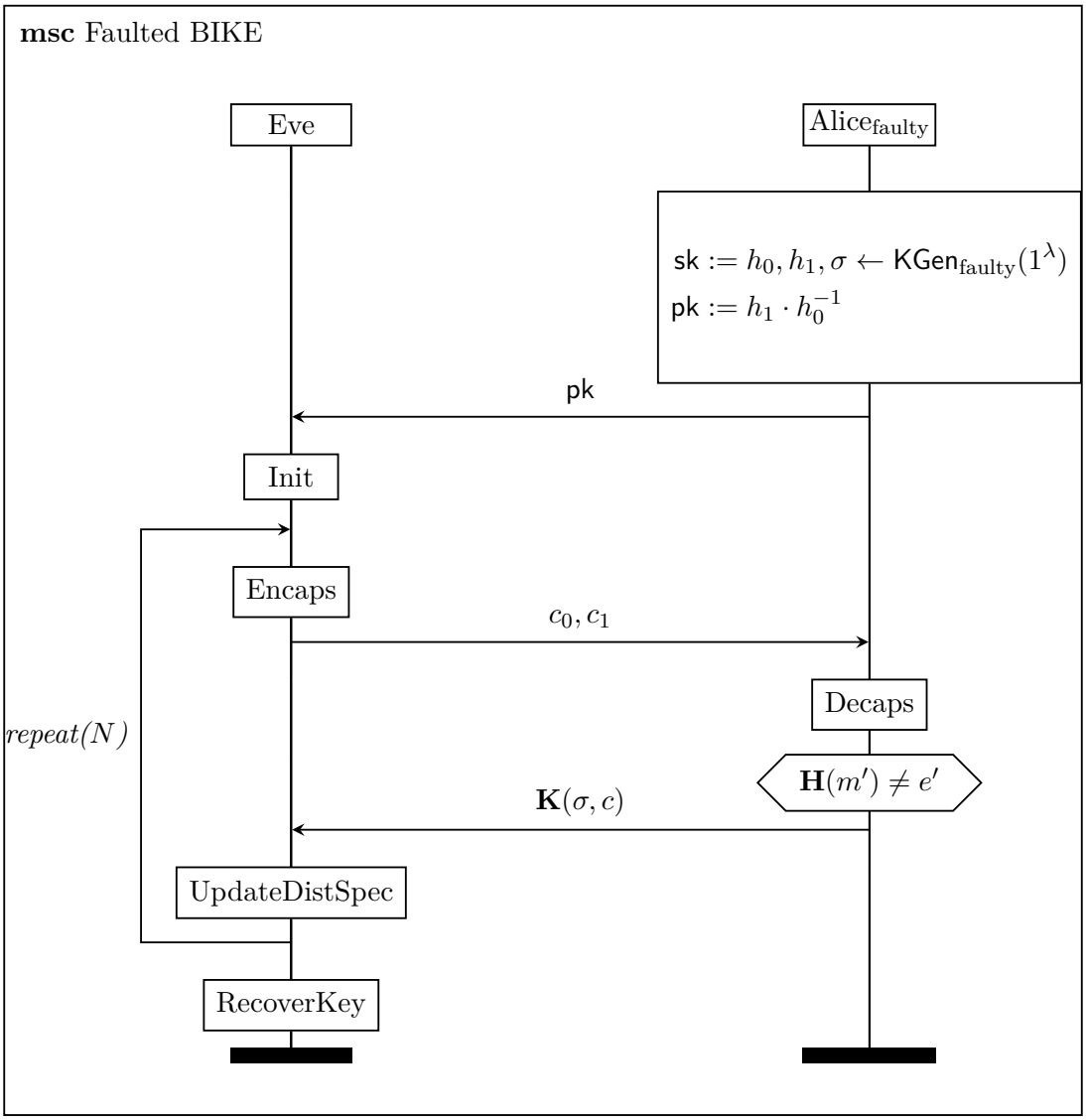


Figure 2: Top level view for key recovery of a faulty or weak secret key

4 Hardware

This section discusses the general setup of the EMFI station, the hardware used, and its purpose in the setup. We also discuss problems encountered during extensive probing runs and their solutions.
Our setup is based on the results of [17]. In their work, the authors build a EMFI station and evaluate it by conducting a fault attack against AMD-SP, an SoC found in current (as of 2021) AMD CPUs.
Our EMFI station consists of a XYZ stage attached to aluminum profiles, controlled by a motion control unit familiar from the 3D-printing community. The motion control unit is in turn attached to a main control unit (a Raspberry Pi 4) via USB. The XYZ stage is equipped with a ChipSHOUTER EMFI tool [19], that emits electromagnetic fields on demand via a text-based serial protocol and other hardware components used to monitor the probing process. The main control unit runs a Python application to control the motion control unit using GCode commands, which are common in 3D printing applications. It also controls the ChipSHOUTER, while monitoring the DUT's state for successful fault injections and crashes and resetting the DUT if needed. The EMFI station can be controlled and monitored via a web interface on the RPi 4. The web interface exposes two camera perspectives, one from above, as well as an overview camera and a thermal camera. The thermal camera is used to approximate the DUT's core temperature, allowing the user to pause the execution of an experiment or increase airflow by adapting the cooler fan RPM if the DUT's temperature exceeds a predefined threshold. This threshold prevents crashes due to core temperatures exceeding the device's permissible temperatures.

4.1 Construction

This section discusses the specific construction of the EMFI station, technical details and safety features regarding the usage of the station.

4.1.1 Target Mounting Plate

To ensure that one can achieve reproducible results, the EMFI station is equipped with a Target Mounting Plate, which ensures a repeatable location of the DUT. This plate has a pattern of 5.32mm holes spaced 15mm apart, and each row is offset 7.5mm from the previous row. For each DUT one has to construct a PCB holder, which ensures a correct and repeatable way of positioning the device.

4.1.2 Safe Construction of ChipSHOUTER Location

When moving the stages of the EMFI station, it must be ensured that they cannot hit the circuit board of the DUT or the aluminum frame construction. To ensure that the probe or the DUT cannot be damaged, the holder for the ChipSHOUTER can let the probe be pushed upwards and away from the DUT. Further, due to the construction of the EMFI station, the stages could crash into the main aluminum construction before

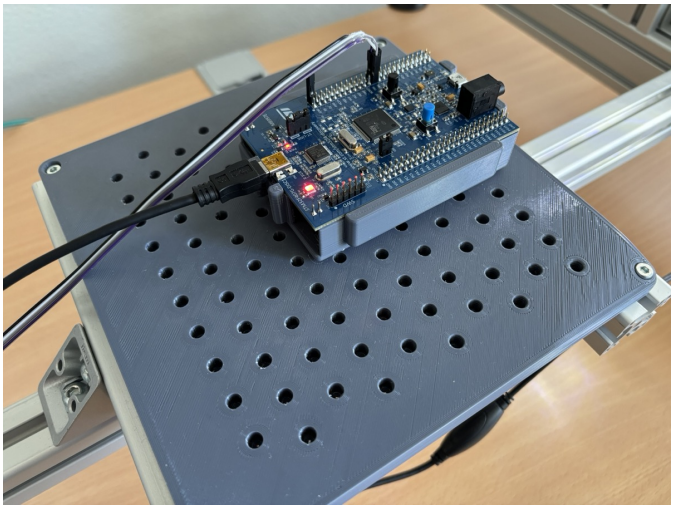


Figure 3: Target Mounting Plate carrying a PCB holder designed for STM32F4Discovery.

the end stops of the stages can trigger. This problem can be solved on both a software and hardware basis. Extended physical end stops were constructed to push the original end stops of the stages. This restricts the range of motion for the stage, but does not affect its ability to *home*, as it still utilizes the existing end stops.

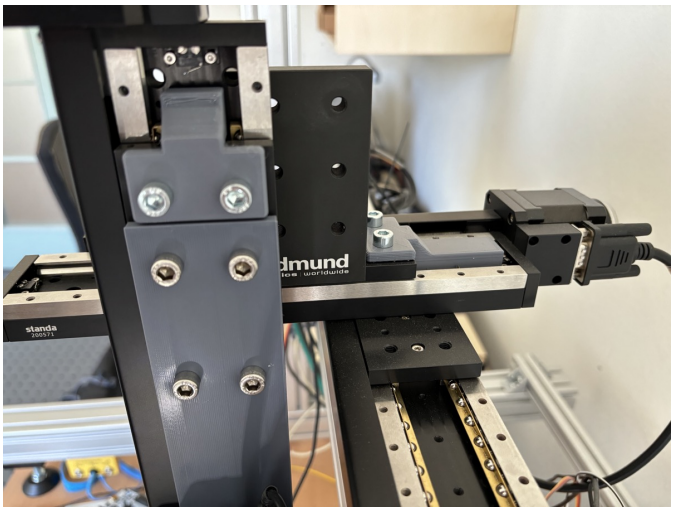


Figure 4: Extended physical end stops for Y and Z axes.

4.2 Challenges

During the project, we faced several issues related to the setup and construction. This section outlines the problems and our proposed solutions.

4.2.1 Cooling

When running faulting experiments over an extended period of time, the DUT can overheat due to the continuous injection of electromagnetic fields through the device. This can lead to unexpected behavior and potential crashes or damage to the device. To mitigate this problem, we installed a 24 V fan to cool the probe and the DUT during the experiments. This fan can be enabled or disabled via G-code commands in software. Due to the mounting of the fan at the holder for the ChipSHOUTER, it automatically moves with the probe and always cools it sufficiently.

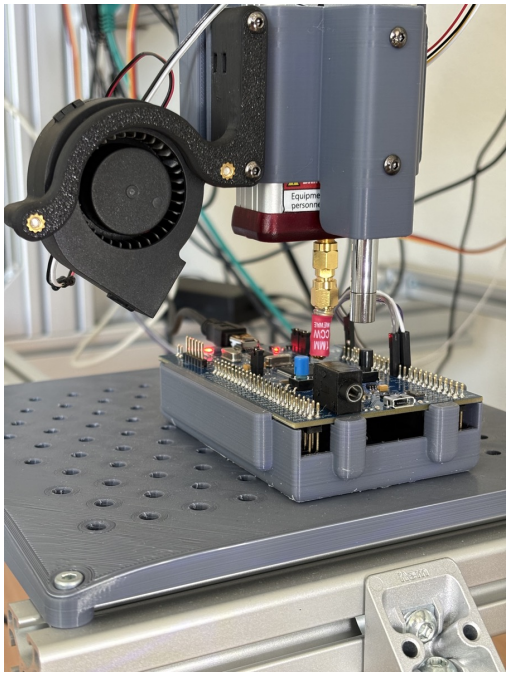


Figure 5: Improved cooling using a 24 V fan.

4.2.2 Controllable Relay

During longer experiments with the EMFI station, it may happen that the DUT does not respond to reset commands or does not process these reset commands as expected (Figure 12). In this case, human intervention is normally required to disconnect the power supply of the DUT and reset it. As a modification, we power the DUT via a 5V relay. This can be enabled or disabled via a GPIO pin, which is on the Raspberry Pi and seamlessly integrates in the software stack.
In practice, we always switch the device on and off to ensure a fresh device state and no errors or faults occur due to a bad reset.

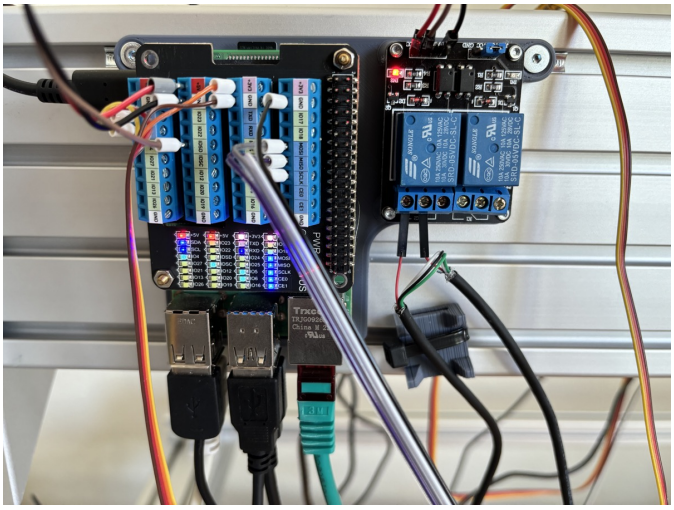


Figure 6: Raspberry Pi (left) running the EMFI controller software and controllable relay (right) used to reset the DUT.

4.3 Pinout

This subsection discusses the pinout of all devices used in the EMFI station. The pin assignment is not relevant and can be changed at will, which requires software and hardware changes.

4.3.1 Power and Data Layout

Figure 8 shows all data and power wires used for the connection between all devices of the EMFI station. All connections between the stages and the controller are exactly the same, so we only specify a general mapping to connect the stepper motor controller to the motors of the stages. Figure 7 describes the power layout of the faulting station in general. There are several power supply units that are required for the different voltages used by the various devices.

4.3.2 Connection between Stages and Controller

The stages used in this project are normally used with industrial CNC control boards, which do not offer the high software flexibility needed for this project. Further the cost of such control boards are way higher than the used solution from the 3D-printing market. Due to the different use cases of 3D printers and CNC machines, the connectors used between normal 3D-printing servo motors and the control board and linear stages for CNC applications differ wildly. Further, CNC stages are equipped with endstops, which run over the same cable as the servo motor voltage, which is different to the 3D printing world, where endstops are used standalone and run over an independent cable. Therefore a custom connection is needed to map the `DB9(M)` stage connector to a 5-PIN JST-XH power plug and a 3-PIN JST-XH plug for the endstop states on the controller.

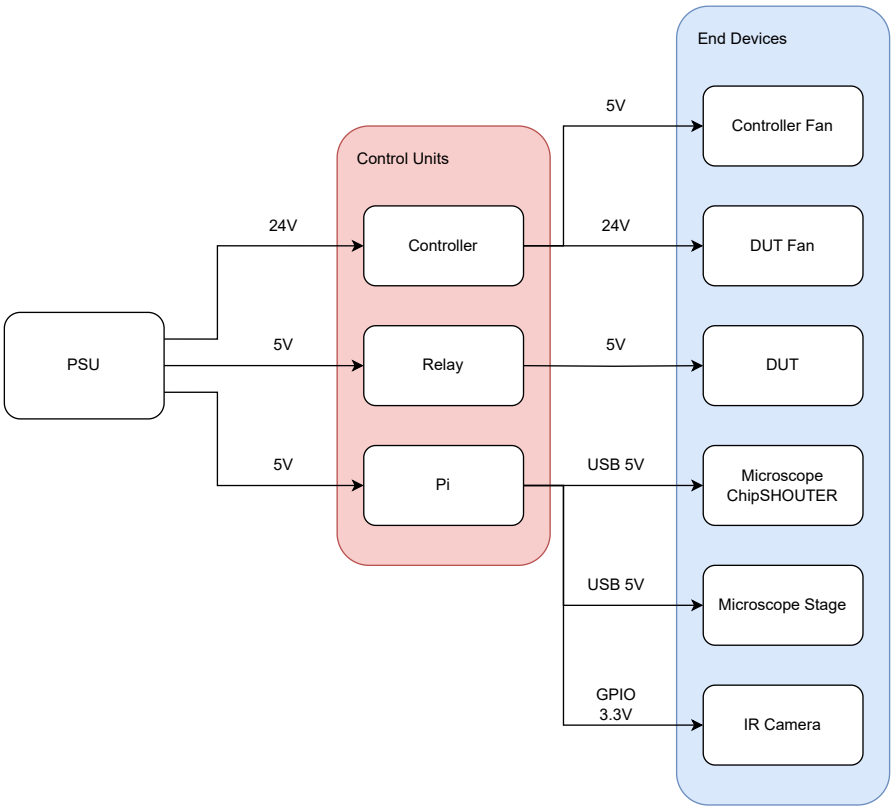


Figure 7: Power layout of the EMFI station.

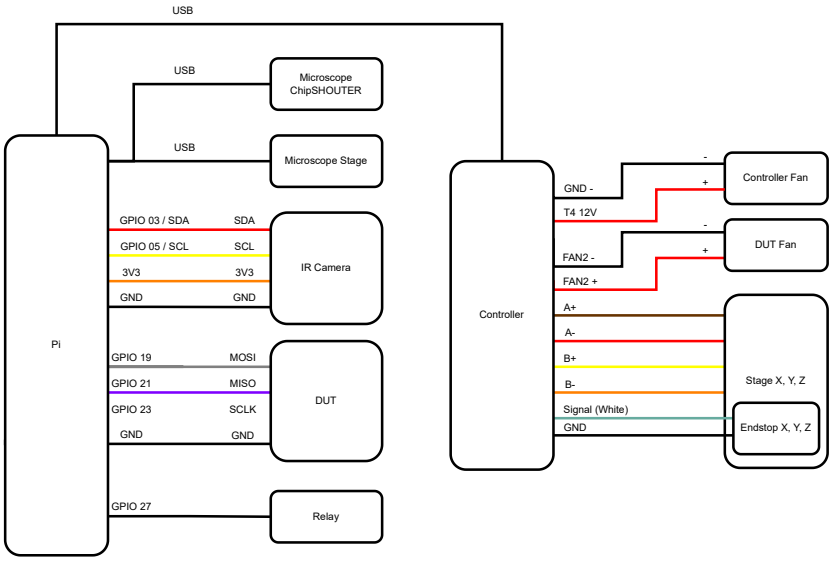


Figure 8: Data layout of the EMFI station.

We used a standard VGA connector, which has the same layout as the connector of the `DB9(M)` cable. This VGA connector is using the given mapping of the cable and via custom soldered wires we connect the stage to the power and control part on the control board and the endstops as denoted in Figure 10.
Each stage has two endstops, therefore we need to use two endstop pins on the controller per stage. The positive and negative wiring of the linear stage is not relevant, the stage is turning in the wrong direction the controller is capable of reversing the turn direction in software.

Figure 9: Stepper controller with DB9(M) stage connectors.

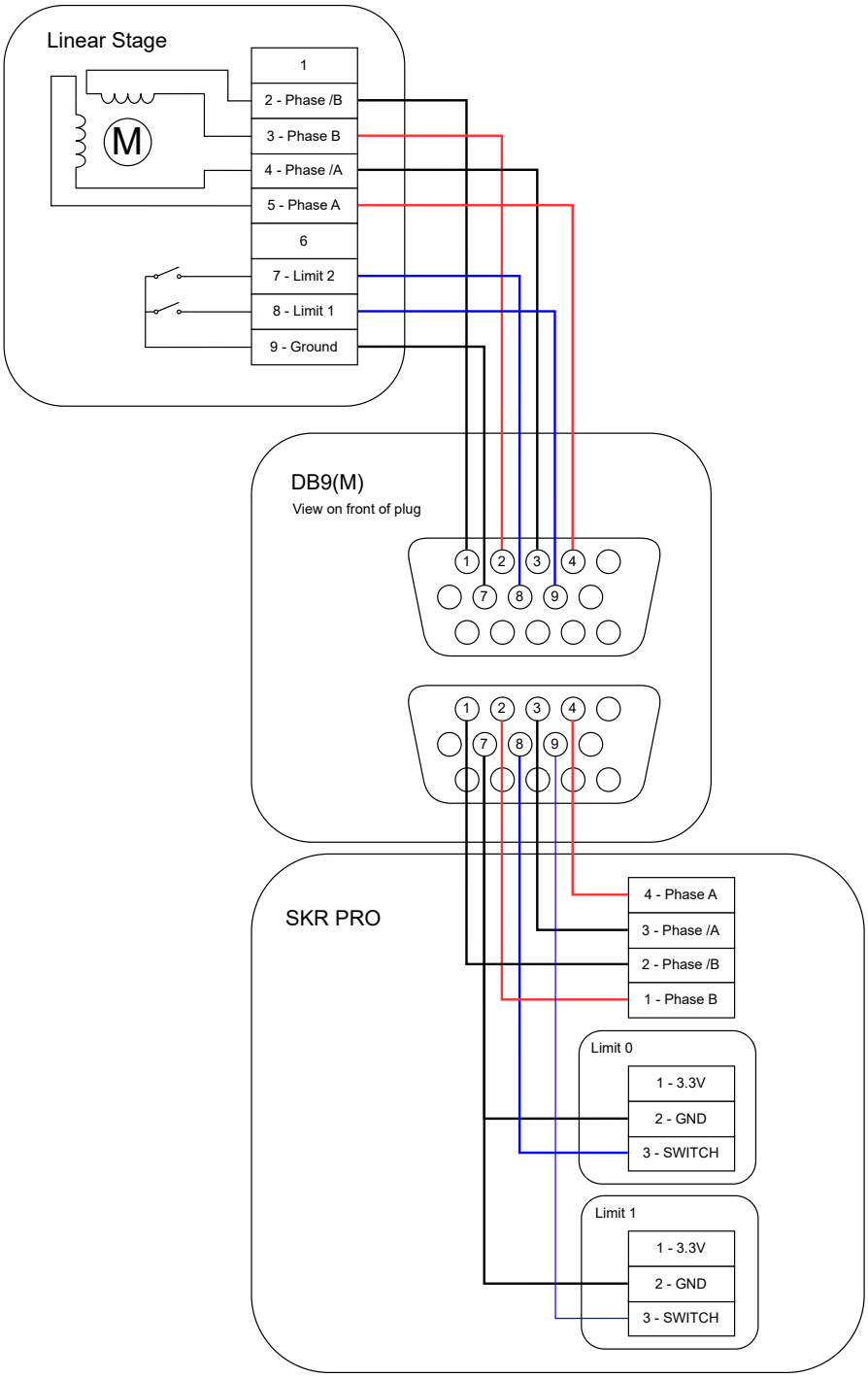


Figure 10: Example pinout between a stage and the controller (Denoted as SKR PRO). The DB9(M) cable has to be split up between the power connection and control of the linear stage and the endstop signal.

5 Experiments

In this section, we introduce and discuss the EMFI experiments we conducted against BIKE on a STM32F4DISCOVERY platform. We also explain the tools needed for using our setup and to reproduce the results. Our EMFI station's software setup is based on [17], which provides a Python framework for fault injection attacks against any kind of target. It is mainly designed for probing a DUT, i.e., finding vulnerable positions on the chip. For the general approach of the fault injection attack, we decided not to target the memory directly but rather the registers of the DUT. This decision resulted from uncertainties about the specific location on the memory of the chip and its potential caching mechanisms that could complicate the fault injection procedures. In addition, the BIKE firmware we target in the next experiment implements a function that is very suitable for faulting the registers during execution. Before faulting the actual firmware running on the DUT, we needed to find areas susceptible to fault attacks, i.e., areas on the chip where a fault would cause bits in a target register to flip. A fault should not only produce bit flips, but also be non-destructive – after the fault, operation should continue as before, ideally with some values in the registers changed. To find areas vulnerable to fault injection, we developed the *Probing firmware* (Subsection 5.2). This firmware sends all tested registers from the DUT while injecting electromagnetic fields using the ChipSHOUTER.
After finding areas susceptible to injecting a fault into the registers, we continue with the *Combined probing* attack against BIKE (Subsection 5.3. The idea is using the actual algorithm (BIKE) to find vulnerable positions on the chip. As running BIKE instead of a lightweight custom probing firmware requires a lot more time, we used the results of the probing run to limit the area to the most promising positions.
Now we can use these algorithm-specific vulnerable regions to run the actual fault injection. To know the internal state of the device and ease the attack, we instrumented the PQM4 [15] implementation of BIKE. The DUT reports its readiness to be faulted and the values of the targeted register using a protocol deduced from the probing experiment.

5.1 EMFI Station Overview

The EMFI station framework provides an abstract `Attack` base class the user can extend for their experiment's needs. It also provides several callbacks, which are called by an `Attack Worker` while conducting the experiment. The program first scans for implementations of the `Attack` base class in the `attacks/` directory on startup and allows the user to start an attack from a drop-down menu in the web interface. Implementations of the `Attack` base class must implement the following methods:

- `__init__(self, ...)`: the base class constructor, used to set important attack parameters. It takes the following arguments:
  - `start_pos`: the start position for the attack. The `Attack Worker` will move the injection probe to this position before starting the fault injection procedure.

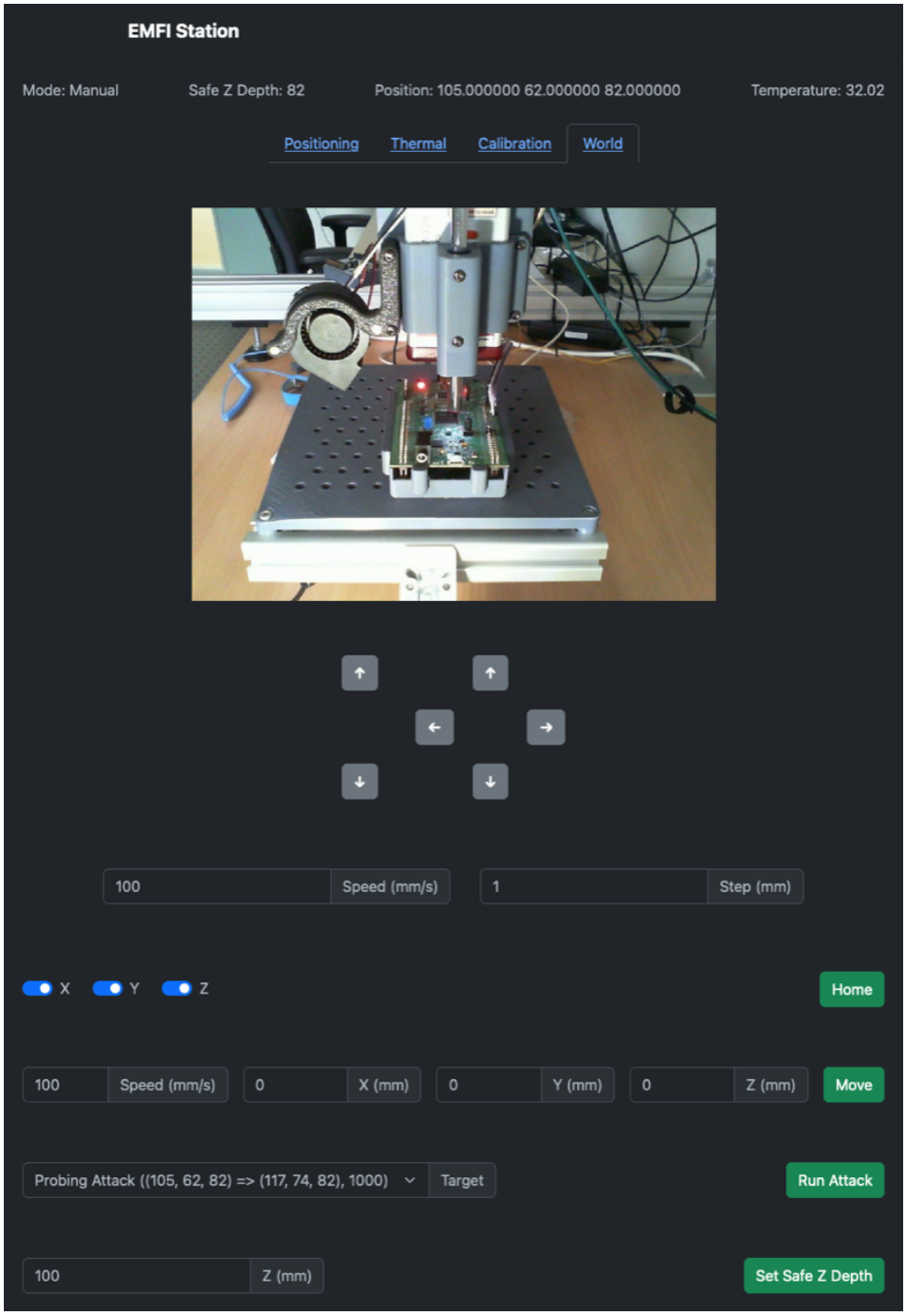


Figure 11: Screenshot of the EMFI station web interface. Various status data are shown at the top. The camera view and the digital joystick for moving the probe are visible in the middle. In the lower area there are control elements for homing and absolute positioning of the probe, starting an attack experiment and setting the safe-Z depth.

  - `end_pos`: the end position for the attack. The `Attack Worker` moves the injection probe in a grid pattern of `step_size` from `start_pos` to `end_pos`.
  - `step_size`: the step size as a float (in mm) the injection probe is moved between iterations.
  - `max_target_temp`: the maximum target temperature (in °C) deemed to be safe for the DUT. If the DUT exceeds `max_target_temp`, the attack is paused until the device cooled down. The target temperature is determined using the infrared camera attached to the injection probe.
  - `repetitions`: the number of attack repetitions for each position in the grid. If you want to target a specific position indefinitely (or until `was_successful` returns true), set this to the max integer limit, compare Subsection 5.3.

- `name()`: a static method returning a human-readable string describing the attack. This name is displayed on the web frontend before starting the attack.
- `shout(self)`: the shout method. It is called repeatedly by the `Attack Worker`. As our setup is based on the ChipSHOUTER framework, we use its API here to arm and fire the probe.
- `was_successful(self)`: check for a successful fault injection in this method. For example, one might try to conduct a key recovery attack in this method and return true iff the key recovery succeeded.
- `critical_check(self)`: conduct critical checks before a call to `shout`. For example, one might check whether the ChipSHOUTER is in a fault-ready state here.
- `shutdown(self)`: shut down the attack. The user might want to persist collected data and disarm the ChipSHOUTER here.

After implementing a simple attack against the *Ballistic Gel* target included in the ChipSHOUTER retail bundle, we proceeded to implement a probing attack as well as an attack against the BIKE cryptosystem. We will discuss both of these attacks in the subsequent sections.

5.2 Probing Experiment

The probing experiment aims to find regions susceptible to fault injection in a DUT. To find those areas, we developed the probing protocol (Figure 12) as well as a firmware and an attack script implementing this protocol.
After the device boots, all tested registers are initialized with known values, and all register values are sent to the host. Afterwards, a fixed *fault window start sequence* is transmitted to the host. The host then knows that the DUT entered the faulting state and starts to inject electromagnetic fields. After a certain time has elapsed, the DUT sends a *fault window end sequence*, which signals the host that it successfully left

the fault state. Afterwards, all registers are transmitted to the host again, and the host compares the register values with the expected values previously received from the DUT. The host side of the protocol is implemented using the `Attack` base class from the EMFI framework. The core of the attack implementation is the collection and processing of data in the `was_successful` callback. If a register value transferred after the fault injection differs from its expected state, a bit flip was successfully injected and logged to a JSON file for further analysis.
During the experiment, the host continuously logs `Datapoint` entities (Listing 1). This structure holds `Response` values for both before and after the fault attack has been conducted, as well as the attack location (X, Y, Z), register differences and register names. For each position (X, Y, Z), an array of N (number of repetitions) `Datapoints` is collected by the host.
Using the $X \times Y \times Z \times N$ dimensional array of `Datapoints` objects, we can then evaluate the probing run using one of the custom `Metrics` (currently, one of `ZeroOneFlipAnywhere`, `Crash`, and `ZeroOneFlipOnR4OrR5`[1]). The metric `ZeroOneFlipAnywhere` (Figure 15a) sums up all recorded bit flips in all registers or evaluates to -1 if the experiment caused the DUT to crash. Analogously, `ZeroOneFlipOnR4OrR5` (Figure 15b) sums up only zero-to-one flips in registers R4 and R5. Finally, the metric `Crash` (Figure 16) evaluates to 1 if the experiment caused the DUT to crash and evaluates to 0 if the device did not crash.

Listing 1: `Datapoint` python class used to hold register values

```python
@dataclass(eq=False)
class Datapoint:
    response_before_fault: Response
    response_after_fault: Response
    attack_location: Tuple[float, float, float]
    reg_diff: Dict[str, List[BitFlip]] = field(init=False)
    reg_names: Set[str] = field(init=False)
```

5.2.1 Probing Protocol

We now describe the host to DUT and DUT to host communication for probing. The protocol is implemented in the host script located in `attacks/probing.py`

1. **Initialization:**
    - The host initiates the communication by sending a `Reset` message. This message is implemented using a relay that is wired to physically toggle the power to the DUT instead of relying on *soft-reset* mechanics found in some STM32 models.

[1] This metric will become relevant for the attack against BIKE.

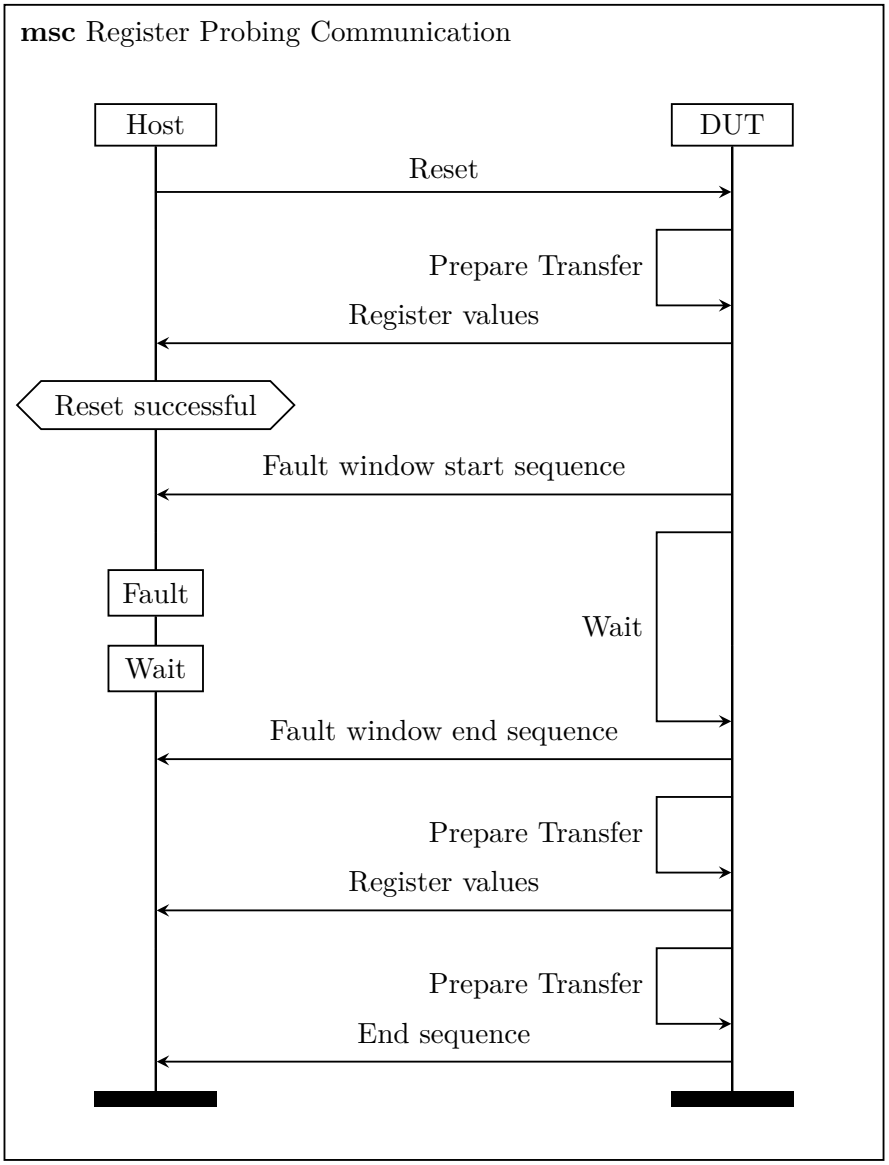


Figure 12: Register Probing Communication MSC.

- Following the reset, the DUT boots up and enters a `Prepare Transfer` phase. During this startup phase, the registers to be observed could, for example, be initialized with a fixed value `0xAAAAAAAA` (checkerboard pattern). This gives a 50% chance to "notice" a bit flip as ones and zeros are evenly distributed. For the attack on BIKE, a zero to one flip is much more relevant, thus we set the initial value of the targeted registers to zero, giving a 100% chance to catch these flips.

2. **Data Transfer:**
   - The DUT transmits the (now initialized) register values to the host.
   - Upon reception of the register values, the host can now verify that the reset was successful.
   - If the host does not receive the expected register values, it concludes that the reset was not successful, and the protocol restarts.

3. **Fault Detection Window:**
   - The DUT signals the start of a fault injection window with a `Fault window start sequence` message.
   - During this window, the DUT enters a wait loop to give the host time to conduct the fault injection experiment. It is important to not use a targeted register for counting down here, as this register does not contain a fixed known value for obvious reasons.

4. **Fault Injection:**
   - Once the host received the `Fault window start sequence` message, it can verify that the DUT has entered the fault window.
   - If no `Fault window start sequence` message was received after a timeout has elapsed, the protocol restarts. As use physical switches to turn off the board, this usually indicates serious problems with the DUT and warrants investigation.
   - Otherwise, the host begins with the fault injection by arming the ChipSHOUTER using the ChipWhisperer API via USB.
   - After injection, the host enters a wait phase, polling the serial interface to the DUT for a `Fault window end sequence` message.

5. **Conclusion of Fault Window:**
   - The DUT signals the end of the fault window with a `Fault window end sequence` message.
   - If the host successfully receives the `Fault window end sequence` message, it concludes that the fault injection did not cause the DUT to crash.

- If the host does not receive a `Fault window end sequence` message until a timeout period exceeds, it is assumed that the DUT crashed and the protocol restarts.

6. **Finalization:**
   - The DUT prepares for transfer and transmits the register values to the host.
   - The DUT concludes the communication with an `End sequence` message.
   - After successfully completing the protocol, the host can now compare the before and after values for each register. If a register holds a different *after* value, the host concludes that the fault injection caused a bit flip in the register. The host logs the current (X, Y, Z) position of the ChipSHOUTER as well as the values observed in this iteration.
   - Restart the protocol.

The Probing protocol is repeated for each position (X, Y, Z) in a predefined range with step width of W mm and a number N of iterations per position. Appropriate choices of W depend on the width of the injection probe in use. A width of $\frac{1}{2}$ to $\frac{1}{4}$ of the injection tip's width appears to be reasonable [26]. Additionally, the DUT wait loop duration as well as the host timeouts have to be adapted according to the DUT's processor frequency. The range to search for areas susceptible to fault injections depends on the DUT's dimension and architecture (see [24] for die-shots of MCUs).

5.2.2 Building the Probing Firmware

The probing firmware is based on the PlatformIO framework [22]. The project is located in `attacks/stm32_probing`. It contains the `platformio.ini` file. To build the firmware, attach an STM32F4DISCOVERY board to your computer via USB and run the command `pio run -t upload` (optionally specify the environment) to build and upload the firmware from within this directory.

5.3 BIKE Attack and Combined Probing

From the probing firmware, we obtained a coarse-grained view of which regions of the chip are susceptible to register fault injection. To get the final and optimal position to inject faults, we opted for a combined probing approach: We have the DUT running the final firmware and move the probe over the most promising region given by the probing run. The BIKE faulting firmware is one of the main contributions of this case study. To understand how to modify existing code to work with our setup, we will explain our changes to the BIKE firmware in detail. As we used an ARM Cortex M4 as the DUT, we forked the `mupq/pqm4` implementation of BIKE [15] and *slightly* modified it. This is necessary to (1) stop faulting once the target weight in the secret key is reached, (2) utilize our stateless serial communication (which proved to be the most fault resilient) to transfer data and (3) stall the device at the targeted function `secure_set_bits`. All of these modifications just ease analyzing the fault performance in a lab setup. For example,

a more capable attacker with more time and resources could perform a statistical analysis to determine the number of cycles after reset in which the DUT is most likely to be in the target instruction.
The modified BIKE code is located in `pqm4/mupq/crypto_kem/bikel1/opt` and the new main function `fault.c` is located in `pqm4/mupq/crypto_kem`. It initializes GPIO (`gpio_setup`) for communication with the host and runs the `MUPQ_crypto_kem_keypair` function that generates a key pair. The modification of the key generator is more involved. It mainly consists of modifying the `secure_set_bits` function (Listing 2) in `pqm4/mupq/crypto_kem/bikel1/opt/sampling_portable.c`. This file also contains the `_transfer` function that sends data via GPIO pins.
The second file that we need to make this experiment work is the `fault_util.S` in the same directory. It contains some helper functions that can only be written in assembly, like sending the register contents (`send_rx_ry`) and stalling the chip by counting down a specific, unused register (`delay_some_time`). The main idea behind this modification is to stall the device before writing the next 64 (on STM32F4DISCOVERY 2×32) key bits in variable `val` to memory (see Listing 2). During this time, the host script will inject the fault using the ChipSHOUTER API.

Listing 2: Modified `secure_set_bits` function modified for 32 bit device (with comments, relevant parts only).

```
void secure_set_bits(OUT pad_r_t * r,
                     IN const size_t first_pos, // is 0
                     IN const idx_t *wlist,     // contains the indices of the ones
                     IN const size_t w_size) {
  // r is a struct that contains R_BYTES (val) and padding (pad)
  // PQM4 is optimized for 64 bit thus we use uint64_t *
  uint64_t *a64 = (uint64_t *)r;
  uint64_t  val, mask;

  // The positions of the quad words and the positions of the bit inside the QW
  uint32_t pos_qw[MAX_WLIST_SIZE];
  uint64_t pos_bit[MAX_WLIST_SIZE];

  // [...]

  // Fill each QW in constant time
  for(size_t i = 0; i < (sizeof(*r) / sizeof(uint64_t)); i++) {
    val = 0;
    for(size_t j = 0; j < w_size; j++) {
      mask = (-1ULL) + (!secure_cmp32(pos_qw[j], i));
      // On a 32 bit DUT, rx and ry contain the 64 bits (val) that
      // contain the new key part
      val |= (pos_bit[j] & mask);
    }
    fault_window_start(); // send fault ready trigger
    // if MOSI PIN is high, delay. Else, immediately send fault window end
    if(gpio_get(GPIOB, GPIO4)) {
      // wait N seconds until fault is injected.
      delay_some_time();
    }
    fault_window_end(); // send fault window end sequence
    // Declared in fault_util.S which sends rx and ry via GPIO serial.
    // Which registers to send is explained in the corresponding section.
    send_rx_ry();
    // The partial key is written back to the stack
    a64[i] = val;
  }
}
```

5.3.1 Building the BIKE Firmware

We use the build process of PQM4 which is based on GNU make [12]. The setup of the authors is rather involved, so we only highlight the important parts of the build process. To build the faulting firmware using the optimized (constant-time) implementation of `BIKEL1` in `pqm4/mupq/crypto_kem/bikel1/opt/fault.c` (`fault.c` contains the main function), run `make -j4 bin/mupq_crypto_kem_bikel1_opt_fault.bin PLATFORM=stm32f4discovery` from the `PQM4` directory. The filename encodes the name of the file that contains the main function, i.e., `fault`. If you plan to build the firmware for a different platform, change the `PLATFORM` environment variable according to your DUT. For a list of supported platforms, refer to the README in `pqm4/README`.
After building the firmware, you should find the resulting .elf file in `elf/mupq_crypto_kem_bikel1_opt_fault.elf` (which is the firmware with debugging symbols), and the resulting .bin file in `bin/mupq_crypto_kem_bikel1_opt_fault.bin`. You can use the .elf file for analysis, for example in Ghidra, see next section. To upload the binary to an STM32F4DISCOVERY, attach a DUT to your computer using a USB cable and use the `st-flash` tool from the STLINK suite [20] to flash the firmware to your device. After successfully flashing the firmware, you might want to issue a reset command to restart the device:

```
st-flash write bin/mupq_crypto_kem_bikel1_opt_fault.bin 0x08000000
st-flash reset
```

5.3.2 Locating the Intermediate Key

We now discuss how to find the registers storing `val` (see Listing 2), which contains the partial key. While it is possible to consider only the key on the stack, analyzing the location of the partial keys stored in registers has the advantage that we can use the results of the probing attack and place the probe over the most promising location for said registers. The process of sending the intermediate key bits involves the `send_rx_ry` function in `fault_util.S` which sends the two registers containing `val`. Knowing the partial key allows for stopping the attack once a targeted weight is reached. When compiling the code for 64-bit machines, `val` is (most likely) stored in a single register, and the method for determining said register can easily be adapted. The procedure for adapting the firmware to a lab setup is as follows:

1. First, we implement all the changes to simplify fault injection. We use a generic function name `send_rx_ry`, which we modify for the actual registers after the analysis. See Listing 3 for the implementation.

2. Next, build the firmware for the DUT. After successful compilation, locate the `mupq_crypto_kem_bikel1_opt_fault.elf` file and analyze it to find which register(s) were used for the partial key(s). For this, we used Ghidra [18]. Figure 13 contains the relevant part from the analyzed firmware inside the `secure_set_bits`

function. We infer that (in this example), the registers r4 and r5 contain the two parts of the 64 bits.

3. We now need to adapt the `send_rx_ry` function (optionally rename it to `send_r4_r5`) to actually send those registers. Now we recompile and (optionally) analyze the firmware again to make sure the compiler kept the register assignments. The firmware is now properly adapted.

To the best of our knowledge, there is no easier way to be certain that a value is actually stored in a specific register. The `register` in C keyword can, and will usually be ignored by the compiler.

Figure 13: Using Ghidra [18] to find the register(s) that contain the partial key.

Listing 3: ARM Thumb mode compatible function that transfers register values located in `rx` and `ry`.

```
send_rx_ry:
    push {r0,r1,r4,r5,lr}
    // some register cannot be pushed to the stack in ARM thumb mode.
    // copy them to pushable registers (r4 and r5) first
    // mov r4, rx
    // mov r5, ry
    push {r4,r5}
    mov r0, sp  // mov sp (location of the r4, r5) to r0
    mov r1, #8  // r0 = begin of registers, r1 = size of two registers
    bl _transfer
    pop {r4,r5}
    ldr r0, =end_seq  // Call transfer with end_seq
    mov r1, #4
    bl _transfer
    pop {r0,r1,r4,r5,pc}
```

5.3.3 The BIKE Faulting Attack Script

The BIKE faulting attack script is a python script used to communicate with the DUT, control the fault injection, and collect data. The faulting attack script extends `EMFI`'s `Attack` base class. The protocol is similar to the probing protocol (Figure 14).

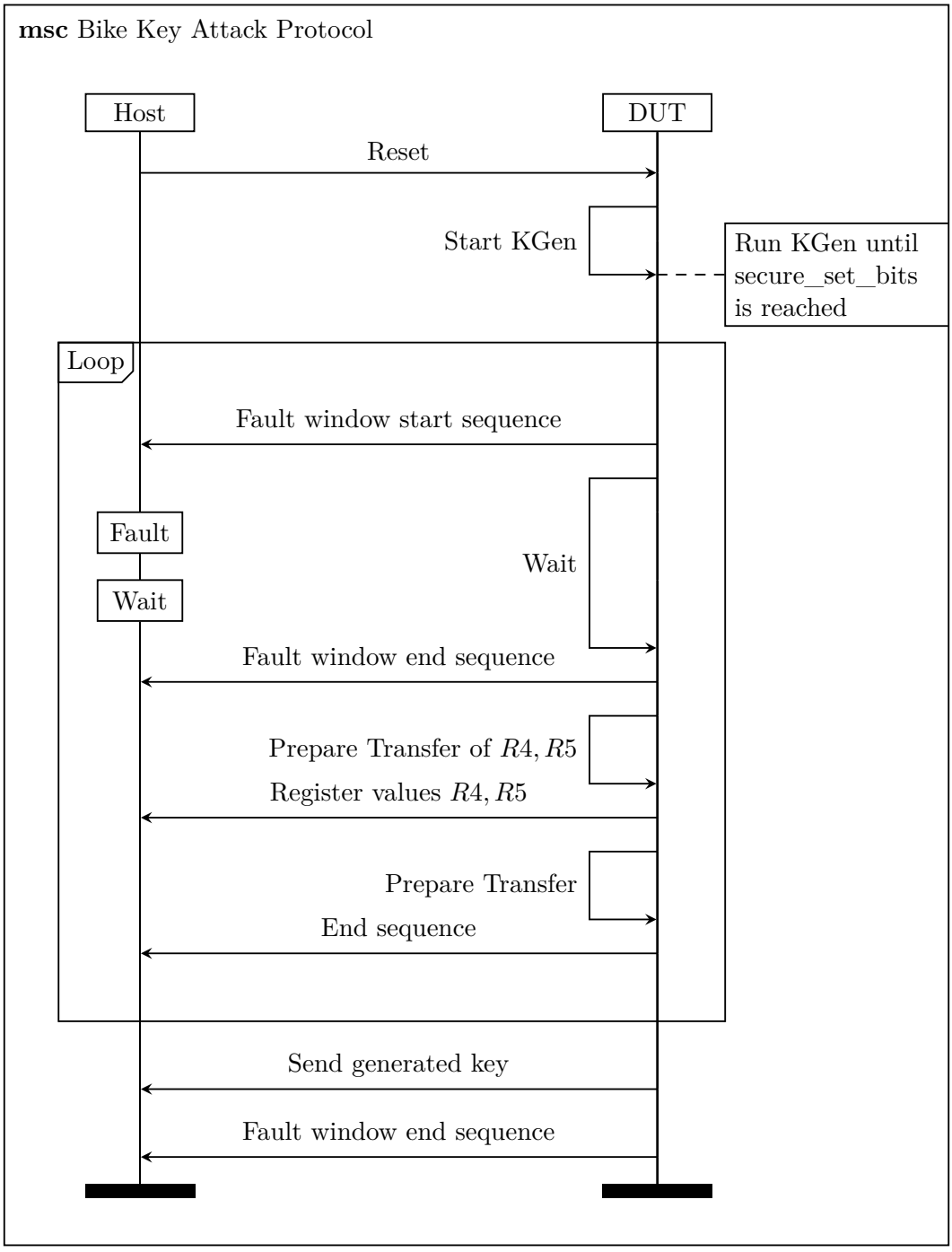


Figure 14: BIKE Attack Protocol MSC.

As we have already determined which area of the DUT is generally susceptible to fault injections (Figures 15a and 15b), we can limit the area of the BIKE attack. This enables

us to conduct an experiment with multiple repetitions while still maintaining a reasonable duration of the experiment – a single instrumented key generator run takes around four minutes, including online analysis.
In the `shout` callback, the attack blocks until a `fault_window_start` message is received. After receiving the message, implying that the DUT is now in a fault-ready state, the ChipSHOUTER is armed by the attack script. After injecting faults during the first three iterations of the loop (Figure 14), the attack script collects the remainder of the generated key. To skip the wait loop on the DUT, we set control pin MOSI to low.
In the `was_successful` callback, the attack script first collects the generated key from the DUT. This acts as a sanity check to ensure that the fault injection did not cause any side effects besides flipping bits in the targeted registers `R4` and `R5`. Afterwards, the weight of the key is calculated. If the weight exceeds a threshold of 95 or falls short of 30 (both corresponding to a DFR of approximately 50%), the attack is deemed successful.

6 Evaluation and Results

In this section, we explore the outcomes of two experiments in the context of this case study. The primary focus was on testing the feasibility of injecting faults during the key generation of the BIKE framework on an actual device. Guided by the work of [16], our goal was to set bits in the secret key stored in intermediate registers. This requires finding the physical location of the registers on the chip. To do this, we have developed a so-called probing firmware. It stores known values in the registers, waits for the fault and sends the register contents back. The more bitflips and the fewer crashes, the more promising the location. An instruction skip, which is a target in many other fault attacks [9] [11] [23] would not make sense in this case, as fewer bits would then be set in the key – the exact opposite of what we want to achieve.

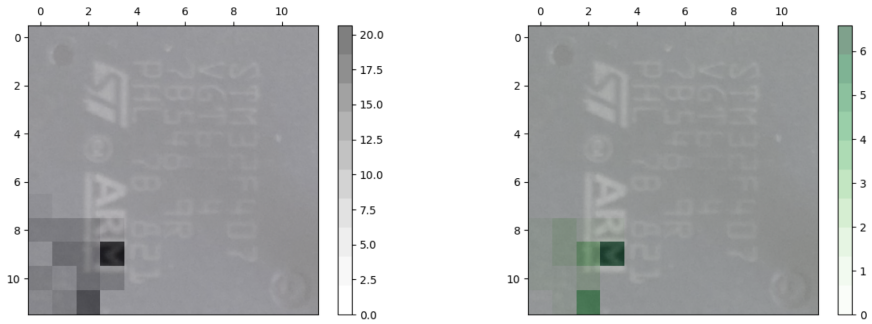
6.1 Probing

We conducted experiments to find EMFI susceptible regions on a STM32F4DISCOVERY microcontroller. For this purpose, we designed a probing firmware that reads the contents of predefined registers and sends them to the host via a custom serial protocol with minimal state.
As can be seen in Figure 15a and Figure 16, we can expect some amount of crashes at positions in proximity to regions where a successful bit flip is observed. This should come to no surprise, as we expect more crashes when the fault injection has an influence in the operations of the DUT. For the attack against BIKE, we are looking for a region where the flip probability of a target register is as high as possible while the crash probability in this region is sufficiently low (ideally negligible). From our evaluation (Figure 15b), we can deduce that the offset $\Delta_x = 3, \Delta_y = 9$ is the most promising for an attack against registers r4 and r5. As mentioned in previous sections, these registers contain parts of the secret key that we are targeting in the BIKE attack.

6.2 BIKE

Following the results of [16], setting or unsetting bits in the secret key of BIKE leads to a higher decoder failure rate, which can be exploited for key recovery. Thus, the main goal was reaching a specific weight threshold of the secret key. [16] established a weight between ≈ 93 and 97 leads to an optimal decoder failure rate at around 50%.
We have also conducted an experiment where we limited the attack to a single point[2] on the DUT, faulting every single iteration of the KGen loop. After around 6 hours of running this experiment, we generated a faulty key with a sufficient weight of 92 (Listing 4). The faulted key was reconstructed within the host script from the register values transmitted by the DUT. Unfortunately, even after intensive troubleshooting, we could not reproduce a fault where the key actually ended up in memory and could have been used in subsequent computations. Instead, when faulting the device, the `.data` section ended up on the stack. All other measures to verify that the DUT is in a known state

[2] determined using the data we collected during the probing experiment



(a) (b)

Figure 15: Heatmaps showing evaluation of (a) `AnyFlipAnywhere`, (b) `ZeroOneFlipOnR5OrR5` metric. The darker the color, the more bits were flipped on average at that position. We only consider data points where the system does not crash during fault injection ($N = 1000$ repetitions per position on a 1mm grid).

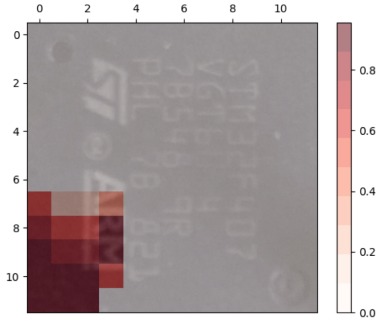


Figure 16: Heatmap showing evaluation of `Crash` metric. The darker the color, the higher the probability of crashing the device while faulting at that position. While some regions are very susceptible to crashing the device, most areas don't influence the device's operation at all. Note that position $(2, 9)$ has a somewhat increased probability of crashing the device and is close to the region we observed bit flips in ($N = 100$ repetitions per position).

have been met. Due to the complexity and number of moving parts, we cannot be certain that this is not a problem with the code. However, it is very unlikely, as running the code without inducing an electromagnetic field results in correct and expected behavior. The most likely explanation is an instruction skip in `delay_some_time` that causes the counter register, which contains a pointer to the `.data` section, to end up in the partial key which is then written back to memory.

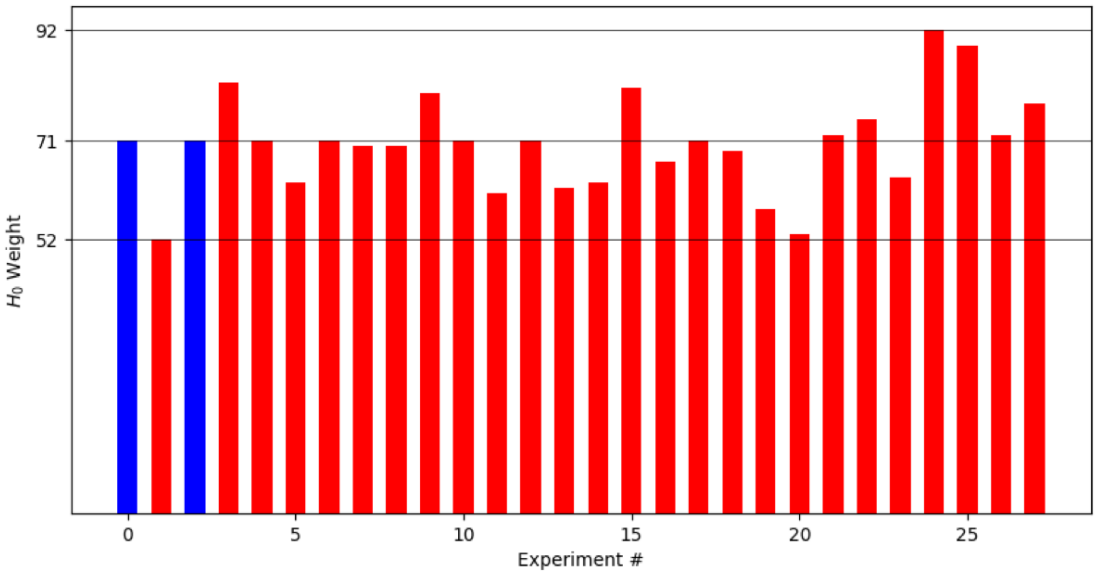


Figure 17: Bar chart showing weights of H_0 with a fixed probe position on an STM32F4 Discovery with ChipSHOUTER settings at 500 V and 3 pulses per *shout*. *Blue bars* indicate runs where the key returned by the KGen function match the register contents collected during the experiment. *Red bars* indicate runs where the key returned by the KGen function did not match the register contents collected during the experiment. These are probably due to side effects caused by fault injection. We observed that non-matching keys often contain parts of a data section inserted for instrumentation purposes: constants for fault window start/end sequence, timers, etc.

6.3 Additional Challenges and Findings

During the course of the case study that lead to this report, we encountered several obstacles, mainly caused by the nature of fault injection. Fault injection, by design, places an additional strain on the device and can completely prevent normal operation at certain locations. Furthermore, the additional energy induced by the fault injection process can lead to a temperature increase outside the device's specified range, which may even result in permanent damage.

6.3.1 Stateless Protocol

Initially, we tried using a serial protocol for communication. However, we quickly realized that the protocol seemed to be implemented in a stateful manner. This meant that faults had a large impact on the communication with the DUT, making using an "in-built" protocol unsuitable for our purposes. To overcome this, we turned our attention to the GPIO Controller, which seemed to be the most reliable option. We decided to implement a custom serial protocol using the MISO (Main In Sub Out) and MOSI (Main Out Sub In) pins of the GPIO Controller. This approach offered more flexibility and reduced dependencies that could introduce side effects. By implementing our own protocol, we

were able to control factors such as the size of the data section, the number of registers used, and the timing of data transmission. This increased the predictability of the system and allowed us to better manage the effects of fault injection.

6.3.2 Reliable Reset

The STM boards we used in our case study provide both soft and hard reset options, which can be invoked via Open On-Chip Debugger (OpenOCD) [21]. However, we encountered occasional issues where the board would stop resetting. This was due to a multitude of errors. For instance, the STM32F4 microcontroller has a lock flag that can lock the processor, preventing any further modifications to the flash memory. This feature is designed to protect the device from unintended programming, but in our case, it posed a challenge as it could interfere with the reset process. To overcome this, we found that the most reliable solution was to perform a physical reset via relays. A relay is an electrically operated switch that can be used to control a circuit by a separate low-power signal. By using such a relay, we were able to physically disconnect and reconnect the power supply, effectively resetting the board. This method proved to be the easiest and most reliable solution for ensuring a successful reset, even in the face of various errors and lock conditions. It allowed us to maintain control over the device and continue our testing process without significant interruptions.

6.3.3 Probe Position and Tip

During our case study, we found that the position of the probe, especially when it was placed at larger distances from the chip, significantly affected the communication. The communication was distorted, likely due to interference with the GPIO ports. This was a challenge as it affected the reliability of our data and the effectiveness of the probing experiment.
In one instance, we experienced an accident where the probe injected a fault not onto the chip, but next to it on the board. On the STM32F4DISCOVERY this appeared to be 1mm below the chip. This led to the board being permanently destroyed, i.e, it was unable to flash, and the data sent via GPIO was unusable. This incident underscored the importance of precise probe placement and the potential risks associated with fault injection. Following this, we took extra precautions to ensure that the probe only faults the chip and not other parts of the board. Regarding the tip of the probe, we experimented with both clockwise and counterclockwise tips. However, we did not observe any significant difference in the results based on the orientation of the EM tip. This suggests that the orientation of the EM tip does not significantly impact the fault injection process in our experimental setups.

6.3.4 Variability in EMFI Resistance

We have also observed that different devices of the same type have different levels of resistance to electromagnetic fault injection. While we conducted most of our early

experiments with high voltages (up to 500 V) and long-term induced magnetic fields with a single device, later on other devices failed during the very first experiment with the same parameters. This suggests that there are significant quality differences in resistance to fault injection attacks within a series. As a result of these observations, we reduced the probe voltage and as compensation moved the injection probe closer to the DUT to counteract failures. However, this also leads to reduced side effects and thus possibly to a lower probability of success of the experiment when repeated with different boards.

Listing 4: Faulty BIKE h_0 key with hamming weight 92 in hexadecimal

```
0000000000000000000000000000000000000000000000010000000000000000000000000000
0000000000000000000000000800000100000000000000000000000000000000000000000000
0000000002080000000000000000000000010000000002000000000000000000000000000000
0000000000000000000000000400000000000000000000000000000000000000000000000000
0000001000000001000000000000000020000000000000000000000000000000000000000000
0000000000000000000000000000000040000001000010000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000010000000000000000000000000000000000000000001
0000000008000000000000000000000000000020000000000000000000000000000000000004
0000000000000000000000000000000000000000000000000000000100000002000000
0000000080000000210000000000000200000000000400000000000000000000020000000000
0008000000004000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000040008000
0000000000000000000000000000000000000000000000000000020000000000080000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000400000000000000000000000100000020000000000000200000000080000
0000000000000000000000000000000000000100000000000000000000000000001000
0000000008000000880400000000000000000020000000000000000000000000000000000000
0000000000000000000000000000000000000100000000000002000000000000000000000000
0000000000001000000000000000000020000000200000000000200000000000000000200
0000000000000000802000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000040000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000040000000000000000000000200000000000000000000000
0000000000000000000000000000000000000100000000000000000000200000000000000000
0000004000000002000000000001000000000000000000000000000000000000000000000000
1000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000800000000000000000000000000000000000000
0000000000000000000000000000020000000000000000800000000000000000000000040000
0000008000000000000000000000000000000000000000000000000000000800000000000000
0004040000000000000000000008000000000000000000000000040000000000000000000020
0000000001000000020000000000000000000000000800000000000000000800000000000000
```

00000000000000000002000040000000000000000000000000000000800000200004010000000000
00000000000400000000000000000000000000000000020000000000000000000000000000000000
10000000000000000000000000000000000000000000000000000200000000000000000000000000
00000000000000000000000000000100000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000040000000000000000000000000000000
000000000000000000000000000000900000000000

7 Conclusion

In phase one of our case study, we built a working setup for an electromagnetic fault injection station (EMFI), based on previous results by Kühnapfel et al. [17].
In phase two, we conducted extensive experiments to examine the susceptibility to EMFI on the STM32F4DISCOVERY platform. We observed susceptible regions that matched the expected register positions derived from MCU die-shots [25].
In the last phase of our case study, we examined the feasibility of EMFI against the BIKE cryptosystem. Using our results from the probing experiment, we already determined which memory regions are most likely susceptible to fault injection. Following a 6-hour experiment, we successfully obtained a faulty key for the BIKE cryptosystem, potentially allowing key recovery. Nevertheless, even after extensive troubleshooting, we were not able to inject faults in a way such that the forged key is actually stored in memory for use in subsequent computations.
Although we assume that we found a reasonable attack against the BIKE cryptosystem, further research is required to overcome the obstacles most likely posed by loop skips or other problems with electromagnetic fault injection against this firmware. Another relevant field of research in the area of electromagnetic fault injections against the BIKE cryptosystem would be to improve tooling for key recovery against faulted keys.

Acronyms

BIKE Bit Flipping Key Encapsulation

DFR decoding failure rate

DUT device under test

EFWK efficiently forgeable weak keys

EMFI electromagnetic fault injection

IND-CCA Indistinguishable for Chosen Ciphertext Attacks

IND-CPA Indistinguishable for Chosen Plaintext Attacks

PQC post-quantum cryptography

References

[1] 2017. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions.

[2] 2019. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions.

[3] 2020. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[4] 2022. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-4-submissions.

[5] Nicolas Aragon et al. *BIKE: Bit Flipping Key Encapsulation*. URL: https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf (visited on 11/26/2022).

[6] E. Berlekamp, R. McEliece, and H. van Tilborg. "On the inherent intractability of certain coding problems (Corresp.)" In: *IEEE Transactions on Information Theory* 24.3 (May 1978), pp. 384–386. DOI: 10.1109/tit.1978.1055873. URL: https://doi.org/10.1109/tit.1978.1055873.

[7] Sebastian Berndt. Personal Conversation. Sept. 16, 2022.

[8] *Bike Reference Implementation*. Accessed: 2024-02-20 17:57. URL: https://bikesuite.org/files/v5.0/Reference_Implementation.2022.10.04.1.zip.

[9] Jean-Max Dutertre et al. "Experimental analysis of the electromagnetic instruction skip fault model and consequences for software countermeasures". In: *Microelectronics Reliability* 121 (2021), p. 114133.

[10] Eiichiro Fujisaki and Tatsuaki Okamoto. “Secure Integration of Asymmetric and Symmetric Encryption Schemes”. In: *Advances in Cryptology — CRYPTO’ 99*. Springer Berlin Heidelberg, 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34.

[11] Clement Gaine et al. “Fault Injection on Embedded Neural Networks: Impact of a Single Instruction Skip”. In: *arXiv preprint arXiv:2308.16665* (2023).

[12] *GNU Make*. Accessed: 2024-01-18 16:00. URL: https://www.gnu.org/software/make/.

[13] Cem Güneri, San Ling, and Buket Özkaya. *Quasi-Cyclic Codes*. 2020. DOI: 10.48550/ARXIV.2007.16029. URL: https://arxiv.org/abs/2007.16029.

[14] Qian Guo, Thomas Johansson, and Paul Stankovski. “A key recovery attack on MDPC with CCA security using decoding errors”. In: *International conference on the theory and application of cryptology and information security*. Springer. 2016, pp. 789–815.

[15] Matthias J. Kannwischer et al. *PQM4: Post-quantum crypto library for the ARM Cortex-M4*. https://github.com/mupq/pqm4.

[16] Sophie Ketelsen. “Fault attacks on BIKE”. MA thesis. Universität zu Lübeck, 2022.

[17] Niclas Kühnapfel et al. “EM-Fault It Yourself: Building a Replicable EMFI Setup for Desktop and Server Hardware”. In: *CoRR* abs/2209.09835 (2022).

[18] National Security Agency (NSA). *Ghidra: Software Reverse Engineering Framework*. Accessed: 2024-01-18 14:13. 2023. URL: https://ghidra-sre.org/.

[19] *NewAE ChipSHOUTER Kit*. Accessed: 2024-01-18 15:00. URL: https://www.newae.com/products/nae-cw520.

[20] *Open source version of the STMicroelectronics STLINK Tools*. Accessed: 2024-01-18 16:08. URL: https://github.com/stlink-org/stlink.

[21] *OpenOCD*. Accessed: 2024-03-07 14:55. URL: https://openocd.org/.

[22] *PlatformIO*. Accessed: 2024-01-18 14:13. URL: https://platformio.org/.

[23] Lionel Riviere et al. “High precision fault injections on the instruction cache of ARMv7-M architectures”. In: *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2015, pp. 62–67.

[24] *siliconpr0n website*. Accessed: 2024-02-20 16:14. URL: https://siliconpr0n.org.

[25] *STM32F407 Die Shot*. Accessed: 2024-02-20 16:14. URL: https://siliconpr0n.org/map/st/stm32f407vgt6/mcmaster_mz_mit20x/.

[26] Alexander Treff. Personal Conversation. Dec. 12, 2023.