Federal Office
for Information Security

Laser Fault Injection Attack on the eXtended Merkle Signature Scheme (XMSS)

Document history

Version	***Date***	***Editor***	***Description***
1.0	06.12.2023		Final Version

Table 1: Document history

Federal Office for Information Security
P.O. Box 20 03 63
53133 Bonn
Internet: https://www.bsi.bund.de

Acknowledgement

This report was authored by the Fraunhofer Institute for Applied and Integrated Security (AISEC) on behalf of the German Federal Office for Information Security (BSI).

Table of Contents

List of Abbreviations

D-LMS	Double laser fault injection microscope
EM	Electromagnetic
GPIO	General purpose input/output
HBS	Hash-based signature
IR	Infrared
LFI	Laser fault injection
LMS	Leighton-Micali hash-based signature
MSS	Merkle signature scheme
MTS	Many-time signature
NIST	National Institute of Standards and Technology
OTS	One-time signature
PCB	Printed circuit board
PQC	Post-quantum cryptography
RST	Remaining substrate thickness
SCA	Side-channel analysis
SRAM	Static random-access memory
WOTS	Winternitz One-Time Signature
XMSS	EXtended Merkle Signature Scheme

1 Introduction

Hash-based signature (HBS) schemes are known for decades but they were not really considered for further research or practical applications in the past. This changed when the need for post-quantum cryptography (PQC) emerged that could withstand attacks by quantum computers.

The standardization of stateful hash-based signature (HBS) schemes started with the publications of the IETF RFCs for the eXtended Merkle Signature Scheme (XMSS) and Leighton-Micali hash-based signature (LMS) in 2018 and 2019, respectively [8],[11] . In 2020 the National Institute of Standards and Technology (NIST) published further recommended parameter sets [7]. The German Federal Office for Information Security (BSI) specifies both algorithms in their own publications [5]. Since their standardization, stateful HBS algorithms have been deployed in several products ranging from embedded devices up to servers [3],[6],[12]. Due to their inherent nature of statefulness, the number of signatures that can be created with a key pair is limited, which also limits the range of applications. In practice, they are most applicable to verify the integrity and authenticity of data that rarely changes, such as the firmware of embedded devices. The verification procedure then takes place during a secure boot or firmware update process. In past works, the research community has investigated hardware and software optimizations for this use case [9],[10],[15],[17] and vendors brought forward products [12].

These efforts demonstrate the need for a post-quantum secure boot and firmware update process. An adversary who can circumvent such a process can execute malicious firmware, which compromises the security of embedded devices completely. Over time, researchers have established, that fault attacks pose a considerable threat to exposed embedded devices, e.g. by allowing exactly such a circumvention of the secure boot process [4],[13]. Developers of secure boot libraries such as MCUboot[1] and microcontroller manufacturers have recognized this by introducing countermeasures against such attacks in the basic control flow [2]. The cryptographic implementations, however, often remain unprotected.

In this report, we practically evaluate a fault attack on the Winternitz One-Time Signature (WOTS) scheme published in [16]. The attack can be mounted on different HBS schemes, such as LMS, XMSS, and SPHINCS+. Both, the verification as well as the signing operation can be targeted. In the original publication, the attack is considered theoretically and evaluated with the help of simulations. In this report, we evaluate the attack practically on a microcontroller using laser fault injection (LFI) as method of attack. We focus on XMSS and the signature verification process with secure boot as an exemplary use case.

Attack setting. We target a signature verification operation in the secure boot setting. Hence, the attacker has access to the device containing the public key used for firmware verification. Instead of trying to entirely skip the secure boot process, the fault attack targets the internal structure of XMSS scheme. Hence, this enables an alternative attack path to other attacks on the execution flow on susceptible targets.

[1] https://github.com/mcu-tools/mcuboot

2 Background

In this section, we outline the WOTS and based on that introduce the basics of the stateful HBS scheme XMSS. The level of detail is kept so that the basic principle becomes clear and is sufficient to understand the attack.

2.1 Winternitz One-Time Signature

We briefly introduce the structure of WOTS and explain how it is used as a fundamental building block in the HBS algorithm XMSS [8]. Note, that this section does not consider any fault attacks, but focuses only on the cryptanalytic security.

To generate a WOTS signature, a message is hashed into an n-byte value m. The message digest m is split into l_1 chunks. Each chunk is interpreted as a value $m_i = \mathcal{N}(m, i)$, i.e. the function $\mathcal{N}$ maps the i-th chunk of m to m_i, where $m_i \in [0, w-1]$ and $i \in [0, l_1 - 1]$. The parameter w is the Winternitz parameter.

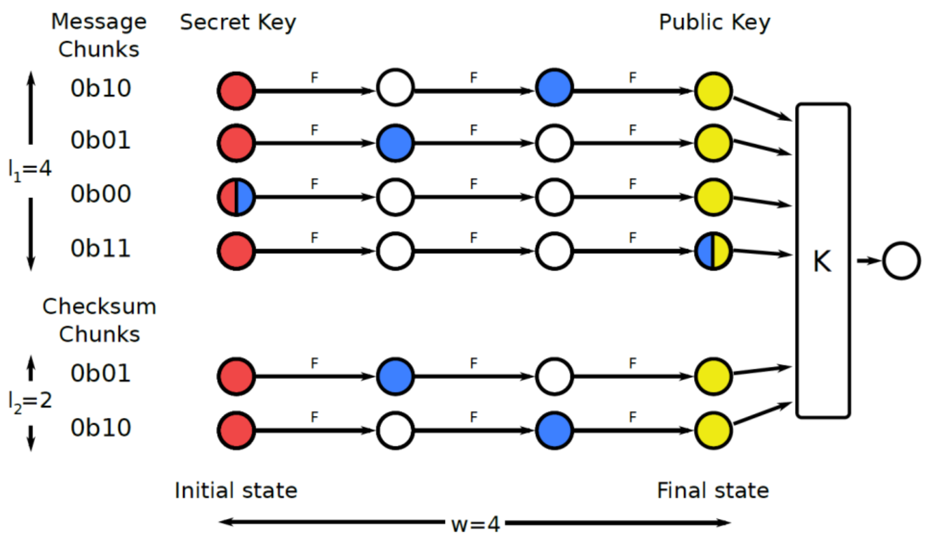


Figure 2.1: Simplified Winternitz One-Time Signature – w = 4 and n = 1 – with the nodes of the secret key (red), the public key (yellow), and the signature (blue) highlighted.

Each of the values m_i is assigned an individual hash chain consisting of w nodes, each represented by an n-byte value. The start node is the one-time signature (OTS) secret key (red), and the end node the OTS public key (yellow). Advancing from one node to another is realized by applying a function $\mathcal{F}$ to the current node. The output of $\mathcal{F}$ serves as the next node. The end nodes are combined by applying the function $\mathcal{K}$ to obtain the compressed OTS public key. Although the exact implementations of $\mathcal{F}$ and $\mathcal{K}$ may differ, we assume both to be single calls to a cryptographic hash function. In reality, before being hashed, the node data might be - depending on the scheme - pre-processed with masks and keys, which are also the output of a hash function.

To **sign** (red → blue) or **verify** (blue → yellow) a m_i the corresponding hash chain is advanced by applying $\mathcal{F}$. For signing, $\mathcal{F}$ is applied m_i times to the respective secret key node (red) and the resulting node (blue) is taken as part of the WOTS signature. For verifying, the signature node (blue) is taken as basis and advanced $w - 1 - m_i$ times. If this does not yield the public key (yellow), the verifier rejects the signature.

If the WOTS scheme were used just with the l_1 hash chains representing the message digest m, an adversary could trivially sign any message, where the digest r consists only of chunks r_i, where $r_i \geq m_i$, $\forall i \in [0, l_1 - 1]$. This is because the adversary gains information about intermediate hash chain nodes from the original signature. Information that was prior to the signing operation, is private. The adversary can simply advance all signature nodes by $r_i - m_i$ to forge a signature. To mitigate this, a checksum mechanism is part of the

WOTS scheme. In addition to the message digest and its corresponding signature nodes, each WOTS signature consists also of a checksum *c*, which has its own signature nodes (Figure 2.1). The calculation of the checksum *c* for a message digest *m* is shown in Equation (2.1).

$$c = \mathcal{C}(m) = \sum_{i=0}^{l_1-1} (w - 1 - m) \qquad (2.1)$$

Put in simple terms, c corresponds to the sum of "steps left" over all message hash chains. The value *c* is split into l_2 checksum chunks c_k, where $k \in [0, l_2 - 1]$ and l_2 is defined in Equation (2.2). The mapping between *c* and checksum chunks c_k is defined by the function $\mathcal{N}(c, k)$, similar to the mapping between message digest *m* and message chunks m_i.

$$l = l_1 + l_2, l_1 = \left\lceil \frac{8n}{log_2\,(w)} \right\rceil, l_2 = \left\lfloor \frac{log_2\,(l_1(w-1))}{log_2\,(w)} \right\rfloor + 1 \qquad (2.2)$$

For the final signature, message chunks and checksum chunks are appended, so that $m_0 \mid m_1 \mid \ldots \mid m_{l1-1} \mid c_0 \mid c_1 \mid \ldots \mid c_{l2-1}$. By doing an index transformation from $k \in [0, l_2 - 1]$ to $j \in [l_1, l - 1]$, we map $c_k = m_j$, so that we can simplify our signature to a continuous series of $m_0 \mid m_1 \mid \ldots \mid m_{l1-1}$, where m_i are nodes corresponding to message chunks and m_j are nodes corresponding to checksum chunks. With the checksum nodes, it is now guaranteed that for a malicious message digest *r* for which $r_i \geq m_i, \forall i \in [0, l_1 - 1]$ the checksum $c' < c$. Therefore, the adversary would have to get to a preceding node from the current node for at least one checksum chain. This is impossible from an algorithmic perspective, as these preceding nodes are neither public nor computable.

2.2 eXtended Merkle Signature Scheme

For most of today's applications of digital signatures, a one-time signature scheme like WOTS alone can hardly ever be used. Therefore, many-time signature (MTS) schemes like XMSS and LMS combine WOTS with one or multiple Merkle trees. Its structure is depicted in Figure 2.2. These schemes are stateful, i.e. the amount of signatures that can be created with one key pair is greater than one but still limited and the signer needs to keep track of the signatures that were already used (maintain a state). As in the previous section, WOTS is used to sign the initial message digest. The WOTS public key nodes (yellow) correspond to the leaf nodes of a Merkle tree. The root node of the tree (orange), in turn, corresponds to the XMSS public key. Therefore, a Merkle tree with a tree height of *h* can authenticate 2^h WOTS key pairs, each of which can be used once.

To sign a message, the signing entity publishes the WOTS signature (blue) and the so-called authentication path (purple). These nodes are used by the verifying entity to compute the root node and check whether it matches the Merkle signature scheme (MSS) public key (orange).

2.3 Fault Attack against XMSS

The attack of [16] enables an adversary to choose an arbitrary message and create a valid signature, which we refer to as forged signature throughout this report. This is possible, e.g. by altering the hashed content by appending a counter to the payload message. The forged signature can be generated if the adversary has at least one signature which was signed with the secret key. The fault attack targets the checksum mechanism of the WOTS scheme to render it ineffective. Before the fault injection attack a brute-force phase is necessary to generate a forged signature that fits the fault capabilities of the adversary.

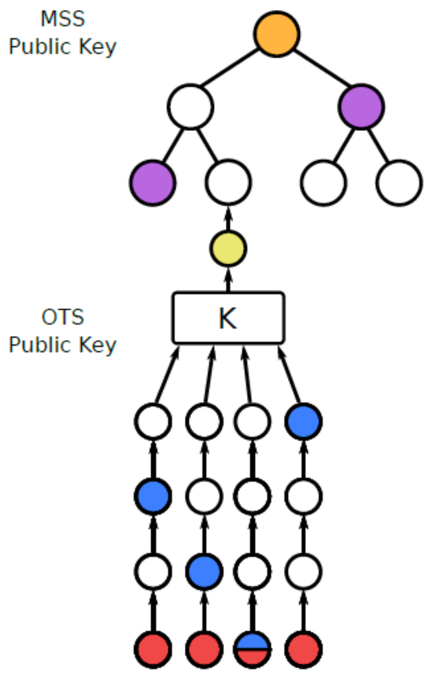


Figure 2.2: The Merkle signature scheme (MSS).

2.3.1 Brute-Force Forgery of WOTS

In the following, we first assume that the checksum mechanism is not part of the WOTS scheme, i.e. is ineffective due to the injected fault. Without the checksum, a WOTS of the message digest m created by an entity A can be used to sign other message digests, e.g. to forge a signature for a malicious payload. This is possible as the hash chains can be advanced by repeatedly hashing the signature chunks. To be able to exploit this, the adversary needs to be in possession of a message digest r, so that $r_i \geq m_i \; \forall i$ ($r_i = \mathcal{N}(\text{r, i})$ and $m_i = \mathcal{N}(m, i)$, see Section 2.1).

The forged signature behaves as if A had signed r using its secret key. Finding a message which maps to such a message digest r is only possible through brute-force search, due to the preimage resistance of the underlying hash function. Even if the adversary has a specific target message, e.g. in the form of a binary, an infinite number of potential forgery targets can be generated by e.g. appending a counter to the payload.

The runtime of the brute-force search depends on the number of signatures an adversary has, the parameters of the algorithm as well as the available hardware resources. A detailed cost estimation for different parameters and hardware can be found in the original publication of the attack [16].

2.3.2 Fault Attack on Checksum Chains

If the adversary is in possession of a malicious message digest r, so that $r_i \geq m_i \; \forall i$, the checksum of r will always be lower than that of m. The checksums cannot be equal as this would imply that all chunks of m and r are equal. We disregard the case where the adversary selects its malicious message to be equal to the original message, as this would be of no benefit. Further, if the digests r and m are equal but not the messages, this would resemble an highly unlikely second preimage attack.

However, for some checksum chunks $r_j = \mathcal{N}(\mathcal{C}(r), j)$ and $m_j = \mathcal{N}(\mathcal{C}(m), j)$, $r_j \geq m_j$ may still hold. For these, the adversary can simply reuse or advance chains of the signature of m for her forgery. But, if $r_j < m_j$, the adversary must know prior nodes of the OTS checksum hash chain. Recovering prior nodes by inverting $\mathcal{F}$

is impossible as it is based on a cryptographic hash function. To overcome this issue, a fault attack is used to force a node to a lower level on the chain than required by the respective checksum chunk.

Consider a value $v \in [0, w - 1]$ for checksum chunk $m_j = v$. Then, the corresponding signature node sig_j is advanced v or $w - v - 1$ times, i.e. $\mathcal{F}^{w-v-1}(sig_j)$. The fault attack forces the implementation to use values smaller than the actual v, or $w - v - 1$, respectively. If the verifying entity is attacked, a correct public WOTS key is derived from nodes too far progressed. With the fault, the adversary is able to forge a valid WOTS for r.

In this report, we do not go into the theoretical analysis of the attack and refer to the original publication [16].

2.3.3 Faulting WOTS to break XMSS

So far we have established how an adversary can forge a WOTS signature with fault injection. This section establishes that faulting WOTS is sufficient to break XMSS and describes the attack an adversary can mount on this scheme.

For the single tree variant of the XMSS algorithm, the adversary is limited to attacking the only WOTS instance that is signing the actual message digest. A successfully forged WOTS signature is also valid for XMSS as the Merkle tree in this scheme only authenticates the WOTS public key.

In case the XMSS verification is attacked, the adversary is able to collect a set of signatures. These signatures are used as an input for the brute-force phase. Depending on the faulting capabilities of the adversary, the success probability of the brute-force phase, and therefore also the computational cost, vary.

During the fault injection, the adversary tries to force the verifier to not advance a checksum hash chain as determined by the respective checksum chunk, i.e. manipulate $\mathcal{F}^{w-v-1}(sig_j)$ to $\mathcal{F}^{o}(sig_j)$, where $o < w - v - 1$. A straightforward approach for the adversary is to manipulate the victim, so that $o = 0$. In this case, the chain calculation of a checksum chunk is skipped entirely and the sig_j node of the forged signature is forwarded directly to the computation of the WOTS public key candidate. To achieve verification, the adversary sets sig_j to the top value of the respective chain, so that the correct public key is computed. In [16], the assumptions on the adversary, where setting $o = 0, \forall j$ is possible and more constrained assumptions, where only individual checksum hash chains are (partly) skipped is evaluated in detail. As described above, these scenarios imply different degrees of freedom for the brute-force phase.

The malicious payload and the forged signature are forwarded to the target device for verification. To trick the verifier into accepting the invalid signature containing an invalid OTS for the checksum, an adversary applies the fault attack as described above. The fault injection was not successful, if the verifier advances this hash chain too far and calculates an invalid compressed OTS public key, which fails verification. If the fault injection was successful, the verifier derives the correct WOTS public key, the signature is verified as valid, and the malicious payload is accepted by the target device.

3 Laboratory Evaluation

In this section, we describe the laboratory evaluation of the fault attack against XMSS. We start with the description of our experimental setup in Section 3.1. In Section 3.2, we analyze the target microcontroller that runs the XMSS signature verification. Finally, in Section 3.3, we evaluate the feasibility and practicability to circumvent the signature verification of XMSS with the help of LFI.

3.1 Experimental Setup

We use the double laser fault injection microscope (D-LMS) from Alphanov [1] as LFI setup. The laser setup is equipped with two laser beams with a wavelength of 1064 nm each. For a precise pulse generation and delay of the trigger signal to control the laser, we use the integrated Tombak pulse and delay generator.

As target device, we use a microcontroller of the STM32F4 series from STMicroelectronics, namely the STM32F401RBT6. This decision was based on the hardware requirements for the studied algorithm with respect to memory and computational power. This microcontroller was already analyzed in a comprehensive study [14] about the susceptibility of microcontrollers to LFI. The STM32F4 has shown to be susceptible to single-bit static random-access memory (SRAM) faults without side effects such as latch-ups.

The target microcontroller is opened from the backside and soldered bottom up on a printed circuit board (PCB) which is mounted on a CW308 board from NewAE Technology. The target board is depicted in Figure 3.1. The CW308 motherboard is mounted on a motorized tilt and rotation table inside the Alphanov D-LMS.

As additional target we chose a device of the STM32L4 series. On this device we were not able to precisely inject faults in SRAM. In Appendix B, we evaluate the whether remaining substrate thickness (RST) influences the susceptibility to LFI and single bit faults in the SRAM region in particular.

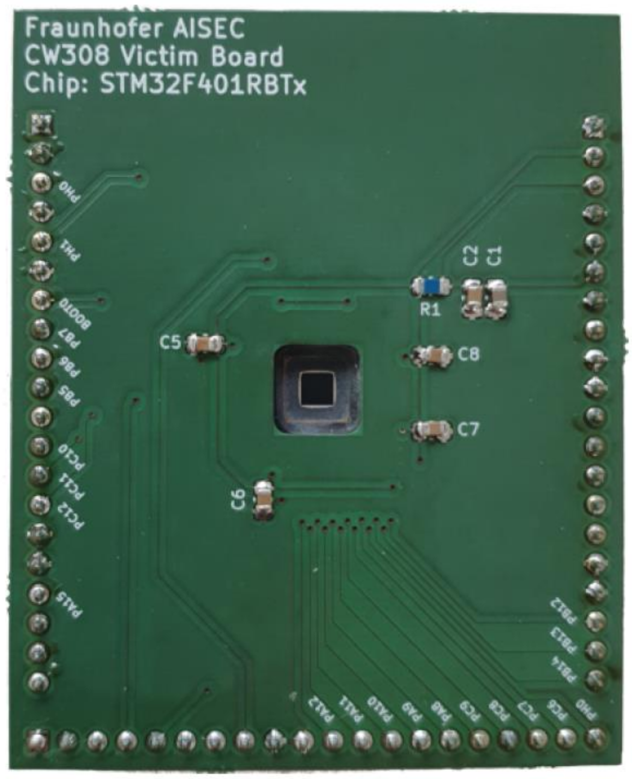


Figure 3.1: CW308 target board for the STM32F401RBT6 microcontroller.

3.2 Device Analysis

In this section, we perform an initial analysis of the microcontroller in order to determine the spatial location for a fault attack on the chip as well as suitable parameters for effective faults. This information is necessary for the practical attack on XMSS in Section 3.3.

In order to identify the locations of the different peripherals of the microcontroller, we take an IR picture of the backside of the chip. In Figure 3.2, an overview picture of the STM32F4 microcontroller is shown. Since the goal is to tamper with the data on the target microcontroller, we search for the SRAM. Based on this overview, we identify the location of the flash memory (red) and the SRAM (green) on the device. We identified these regions due to their regular structures. The flash memory can be distinguished from SRAM through its more complex read and write logic right next to it. The SRAM is organized in two blocks of equal size.

In a next step, we perform fault injection tests on the edges of both SRAM blocks. With that we determine the relation between the spatial location of the injected fault and the memory address of the SRAM. During this process we also narrow down the parameter space that is suitable to inject faults in the memory. Later, we will fine tune these parameters such that we can precisely inject faults on a certain memory word and bit position in the SRAM. To identify the memory organization we perform a grid scan of 12x12 positions.

In Figure 3.3, the mapping between spatial location and memory word indices for both SRAM blocks is depicted. The left plot shows the behavior on the first (upper) SRAM block and the right plot for the second (lower) block. In the first SRAM block, the words are organized horizontally whereas in the second block, the words are organized vertically. One can see that the indices increase from right to left and from top to bottom for the left and right plot, respectively. This reflects the 90 degree rotation of the two memory blocks. From the picture we infer that the highest address is located at the right side of the second SRAM block. In Figure 3.3 one can see that there are memory locations where there is no fault effect. Hence, only the memory organization is revealed by the experiment and not the location of certain memory addresses. In Section 3.3 we fine tune the parameters to target for a specific memory location.

Based on this evaluation we can deduce that the SRAM is split into two regions: the first block (upper) covers the address region from 0x20000000 to 0x20007fff. The second block (lower) covers the memory from 0x20008000 to 0x2000ffff.

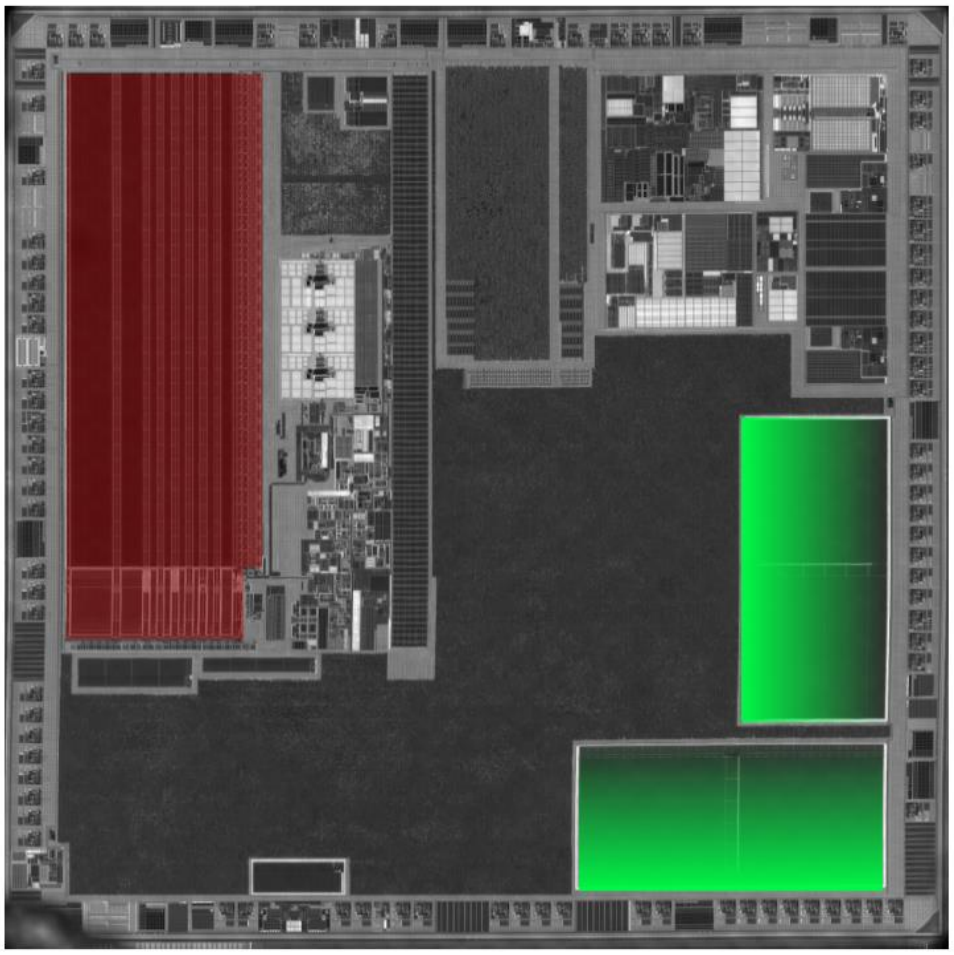
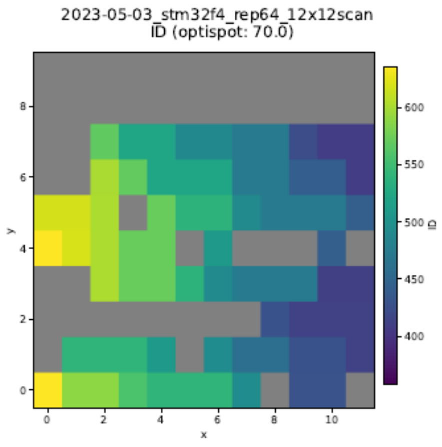
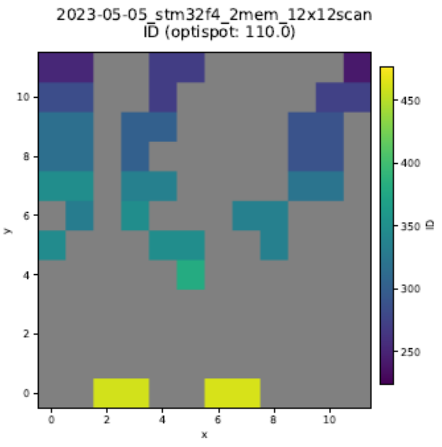


Figure 3.2: Backside infrared (IR) die shot of the STM32F401RBT6 microcontroller. The flash memory region is highlighted in red and the SRAM region in green. The gradient within the SRAM highlights the memory addresses deduced in Figure 3.3.



(a)
First (upper) SRAM block, showing increasing memory addresses from right to left.



(b)
Second (lower) SRAM block, showing increasing memory addresses from top to bottom.

Figure 3.3: Heatmaps of the memory address faulted within for a small region inside the first (a) and second (b) SRAM block in relation to position with step-size 1µm. The color indicates the offset of a faulted memory word.

3.3 Practical Attack on XMSS

In this section, we use the gained knowledge of the previous analysis to mount the actual attack on the XMSS verification operation.

To implement a verification application that shall represent a secure boot use case, we use the XMSS reference library[2]. For the hash operations required for XMSS, the tinycrypt[3] library is used. The STM32F4 microcontroller is configured for a core clock frequency of 16 MHz. As XMSS parameter set, we use XMSS-SHA2-10-192, being one of the recommended parameter sets by NIST [7]. With this parameter set we have 51 hash chains where the last three chains are used for the checksum.

We have made this decision in order to keep the brute-force search for a forged signature within a time frame that is practicable. The source code of the verification application is listed in Listing 5-1. The firmware first initializes the microcontroller and the necessary peripherals. Afterwards a message and its corresponding signature are verified. In order to keep the application as simple as possible, the message and the signature are hard-coded in memory.

After the device initialization, the firmware calls the *verify_signature()* function which calls the *xmss_sign_open()* function of the XMSS reference library to verify the signature. The result of the verification is stored in a variable passed as pointer to the function. To indicate whether the signature was successfully verified or not, we use the global variable *device_status*. This variable is read out via the debug interface by our evaluation tool after every experiment is executed. At the beginning this variable is set to an initial value, see line 52 of Listing 5-1. This allows us to identify if the injected laser fault crashed the firmware execution.

The firmware is executed from SRAM in order to prevent us from always having to write the firmware to the flash memory. Note that this does not influence the experiments and is just for the sake of convenience.

Signature Forgery. We generate a forged signature using the brute-force approach mentioned in Section 2.3.1. The required computational resources can be estimated with the results given in [16]. If the adversary possesses 100 valid signatures, a suitable signature is found with a probability of 50 % after around 104 GPU seconds. In our case the found signature requires a fault attack in the first checksum chain. In this chain, we need to force the second bit, changing the value from 0 to 2.

In Table 3-1, the hash chain length values before and after the fault injection are listed. The modified value in the 49th hash chain element is highlighted in boldface and red.

Table 3-1: Hash chain values of the forged signature before and after the fault attack.

	1	**2**	**3**	**4**	**5**	**...**	**48**	**49**	**50**	**51**
Before	7	10	7	10	12	...	8	0	14	13
After	7	10	7	10	12	...	8	**2**	14	13

Fault Location. The fault attack against the XMSS verification operation requires that the number of hash operations is tampered such that the public key generated with the forged signature matches the public key stored in the memory of the device.

The function in the XMSS reference library that calculates the public key from a signature is called *wots_pk_from_sig()*. In Listing 3-1, the source code of this function is listed. In line 9, the number of hash operations is calculated and stored in the array *lengths*. The *for* loop in line 11 to 17 then performs the hash operations on all chains.

[2] https://github.com/XMSS/xmss-reference
[3] https://github.com/intel/tinycrypt

For a successful attack, we need to induce the fault in the hash chain lengths array after the hash chain length is calculated (line 9). Otherwise, the injected fault would be overwritten by the software. The fault must also be injected before the hash operations for the first check sum is calculated within the 49th iteration of the *for* loop in line 11 to 17 (also see Table 3-1). This means that we have to inject the fault within the first 48 iterations of this *for* loop.

Listing 3-1: Source code of the XMSS reference implementation to calculate the public key from a signature

```
 1 void wots_pk_from_sig(const xmss_params *params,
 2     unsigned char *pk, const unsigned char *sig,
 3     const unsigned char *msg, const unsigned char *pub_seed,
 4     uint32_t addr[8])
 5 {
 6     int lengths[params->wots_len];
 7     uint32_t i;
 8
 9     chain_lengths(params, lengths, msg);
10
11     for (i = 0; i < params->wots_len; i++) {
12         set_chain_addr(addr, i);
13
14         gen_chain(params, pk + i * params->n,
15             sig + i * params->n, lengths[i],
16             params->wots_w - 1 - lengths[i], pub_seed, addr);
17     }
18 }
```

The lengths array resides in SRAM, more precisely on the stack. The stack usually resides at the end of the memory and grows from high to low addresses. Before we search for the exact position on the chip to inject the fault we need to determine the memory address. The lengths array is located at address 0x2000f150. Since we need to modify the 49th element of the array, the memory location we need to target is 0x2000f210. Note that the size of every element in the array is 32 bit..

Together with the information of the device analysis in Section 3.2 we have everything at hand to perform a scan on the SRAM to find the location where we need to inject the fault to bypass the verification operation. From the device analysis we know that the stack resides in the second (lower) SRAM block. The last memory addresses are located at the bottom right corner of the memory block.

In Figure 3.4 (b), a detailed view of this memory location is depicted. In this area we perform a grid scan to search the position of the address 0x2000f210. In order to reduce the search space, we use a pulse width of 1600 ns. This value has proven to be a good parameter choice during the previous device analysis.

Additionally to the spatial location, we sweep the pulse intensity as well as the focus using the Optispot technology of the Alphanov D-LMS for the parameter search. The grid scan is performed with a 20x optical zoom.

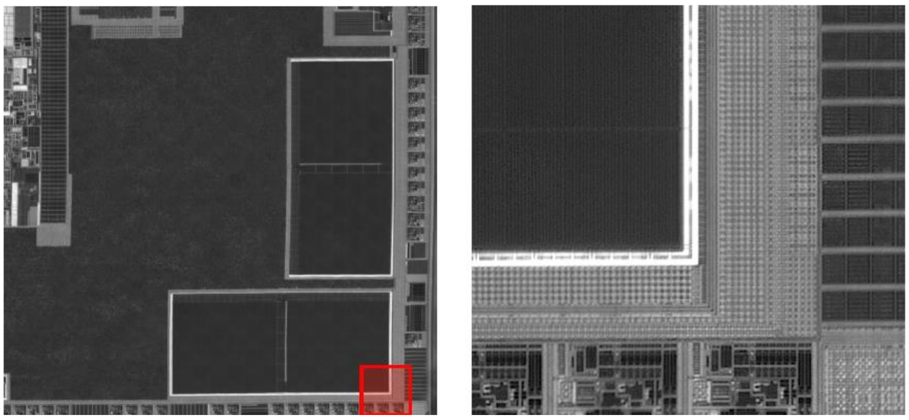


Figure 3.4: Backside IR die shot of both SRAM blocks (a) and a detailed view (b) into the red rectangle.

Trigger Preparation. Now that we determined the location in the SRAM, we need to setup a trigger to inject the fault at the right point in time. For our scenario, the right point in time is after the hash chain lengths are calculated and before the hash operations of the 49th hash chain are executed.

In order to identify the right point in time to inject the fault, we modify the *wots_pk_from_sig()* function such that a general purpose input/output (GPIO) is set and reset before and after the *gen_chain()* function is called. With that we can determine the point in time of each hash chain calculation relative to the beginning of the firmware execution. Additionally, we perform an electromagnetic (EM) side-channel analysis (SCA) in order to identify potential patterns in the EM signal that we can use as trigger. For the side-channel analysis (SCA), we place a Langer RF-U 2,5-2 near field probe on the shunt resistor (R1) on the target board (Figure 3.1).

In Figure 3.5, the EM signal (top) and the timing of the GPIO signal (bottom) is depicted. From the EM signal, we can see that there is no suitable pattern that can be used as a trigger signal. The peaks in the signal do not correlate with the hash chain operations nor the GPIO signal. We placed the EM probe also on other locations such as on the capacitors of the digital power domain C5 to C8 but without obtaining a suitable signal that can be used as trigger. We assume that the strong peaks stem from the integrated power regulator of the STM32F4.

Since we are not able to obtain a suitable trigger from the EM signal, we need a different trigger source. From the bottom plot in Figure 3.5 we can see that the hash operations take multiple seconds. For that reason, we decide to use the start of the firmware execution as initial trigger for the fault injection. The distance between two ticks shows the duration of a hash chain operation. The dark gray area represents the hash chain operations where a fault is effective whereas the light gray area represents the hash chain operations where a fault attack is not effective.

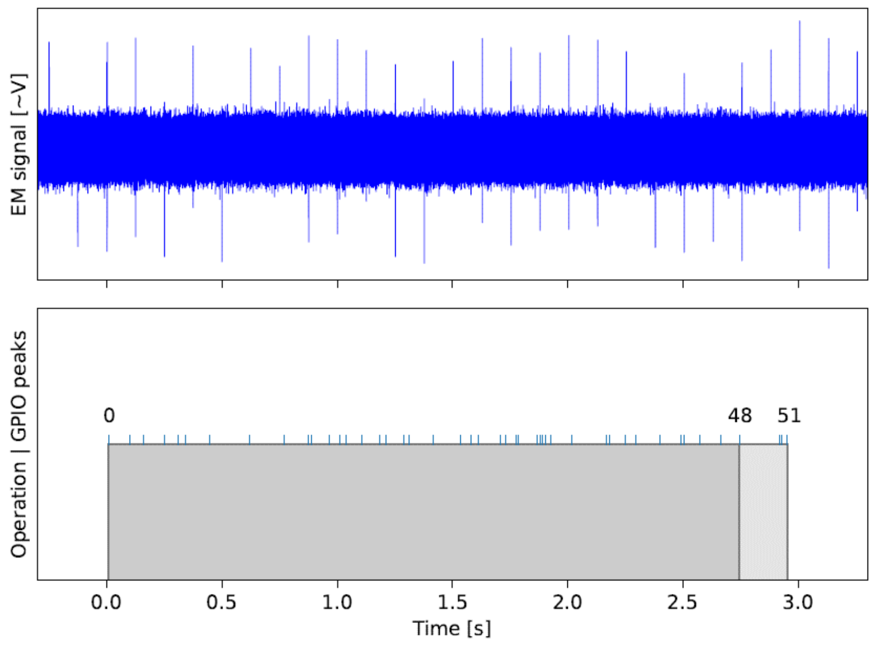


Figure 3.5: : Timing of the hash chain during computation. In the top the EM signal over time is depicted, showing no correlation with the GPIO signal depicted in the bottom. Within the dark area in the bottom graph, the initial hash chains are evaluated, whereas in the light area the checksum hash chains are evaluated.

From the plot one can see that a fault can be injected for more than 2.5 seconds after the firmware execution starts. After about 2.7 seconds, an injected fault has no effect anymore because the hash operations for the 49th hash chain is already processed. Note that we need at least a short delay of about 10 milliseconds relative to the start of the firmware until the *gen_chain()* function is executed.

Results. Different delays were evaluated based on a trigger using the start of the firmware execution. In order to evaluate the success rate, for each delay ranging between 0.0 s up to 2.5 s in steps of 0.5 s, 32 repeated measurements were performed. For delays within the above mentioned time frame (positive delay, smaller than 2.5 s), success rates between 50% and 80% were achieved.

For the two evaluations outside this time frame, a success rate of zero was reported. During these experiments the laser parameters were kept fixed to the most promising settings evaluated in the prior tests, see Table 3-2.

Note however, that any success rate larger than zero would allow an attacker to breach security. As in the secure boot use case, a practicable infinite amount of reboots and thus repeated attack tries are possible.

Table 3-2: AlphaNov settings for success rate evaluation.

Parameter	Value
Optical zoom	20x
Pulse width	1600 ns
Peak current	9.0% (max. 4000 mA)
Optispot	50

4 Conclusion

In this report, we experimentally evaluated a fault attack on the WOTS scheme published in [16] in the context of PQC. Note, that the attack can target both the verification as well as the signing operation of different HBS schemes. In this report the XMSS signature verification process was targeted using LFI in a secure boot setting.

Within the laboratory evaluation, two targets were tested, out of which a suitable target was selected and analyzed. The location and address-structure of the SRAM was identified. This allowed in a further step to map the word-placement to the targeted address in memory.

Attacking the verification step required the generation of a forged signature, which looks benign up to a single bit-flip in a single hash chain. In this report, a forged signature with a single forced bit was used, requiring the LFI attack to set a specific single bit.

Timing of the introduction of the fault was of minor concern within this setting, as the targeted hash chain allowed for a possible time-frame of a few seconds. This further reduced the requirements of a precise trigger signal, which was hard to obtain in the experimental setup.

We showed, that only minor timing constraints are necessary in order to achieve a successful breach of security. This is of specific interested in real-world application, where finding a suitable trigger signal shows a significant role in the experimental setup. The results of this report further highlight the necessity of single bit-faults for these kind of attacks, which require a specific attacker skill-set. Within this report, the selected target proved to be susceptible to these kind of faults.

In [16] also generic attacks containing invalid signatures were analyzed. These offer an alternative to circumvent signature verification in the considered use case. The difficulty of these attacks was considered of similar level. However, not all analyzed implementations were susceptible to the general attacks. For example, the XMSS reference implementation is only vulnerable to the WOTS-specific attack but not to the general. It should be possible to design mitigations for both types of attack, with the WOTS-specific attacks requiring knowledge of the underlying signature schemes.

As a possible countermeasure to this attack, [16] discusses the possibility of repeated calculation and comparison of the hash chain length. This way, a tampering of the hash chain length can be detected. From a performance perspective, this countermeasure only introduces a negligible overhead compared to the total cost of hash computations.

5 Appendices

5.1 Source Code

Listing 5-1: Source Source code of the firmware to verify the XMSS signature.

```c
#include <stdint.h>
#include <stdbool.h>

#include "board.h"

#include "xmss-reference/xmss.h"
#include "xmss-reference/params.h"

#include "trigger.h"
#include "public_key.h"
#include "signature.h"
#include "message.h"

enum {
  DEVICE_STATUS_INIT = 0x784ba06f,
  DEVICE_STATUS_VERIFY_OK = 0xa0b1c2d3,
  DEVICE_STATUS_VERIFY_FAIL = 0x4e5f6a7b,
};

static void verify_signature(bool *valid_signature)
{
  uint32_t xmss_oid;
  xmss_str_to_oid(&xmss_oid, "XMSS-SHA2_10_192");
  xmss_params params;
  xmss_parse_oid(&params, xmss_oid);

  *valid_signature = false;

  const unsigned char *sm = signature_bin;
  const unsigned long long smlen = signature_bin_len;
  const unsigned char *pk = ___xmss_files_public_key_bin;
  const unsigned int message_length =
    ___xmss_files_message_bin_len;

  unsigned long long mlen = 0;
  unsigned char m[params.sig_bytes + message_length];

  int ret = xmss_sign_open(m, &mlen, sm, smlen, pk);

  if (ret >= 0) {
    *valid_signature = true;
    return;
  }
}

int main(void)
{
  HAL_Init();

  trigger_init();
  trigger_set(false);

  device_status = DEVICE_STATUS_INIT;

  bool valid_signature;
  verify_signature(&valid_signature);

  if (valid_signature) {
    device_status = DEVICE_STATUS_VERIFY_OK;
  } else {
    device_status = DEVICE_STATUS_VERIFY_FAIL;
  }

  while (true);
}
```

5.2 STM32L4: Influence of Remaining Substrate Thickness (RST)

By thinning of the chip, we were able to analyze the influence of the RST. In Figure 5.1 the original un-thinned chip was measured, in Figure 5.2 the chip with an RST of 60µm and in Figure 5.3 with 30µm RST. We were not able to introduce single bit faults in the region of interest.

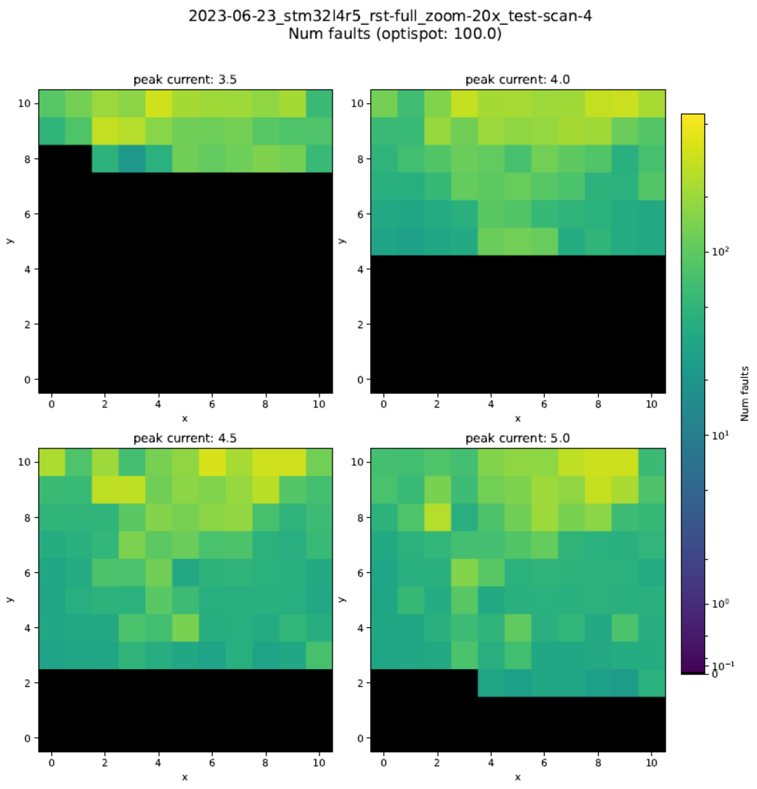


Figure 5.1: Heatmaps for the number of faults on the un-thinned STM32L4 with different peak currents.

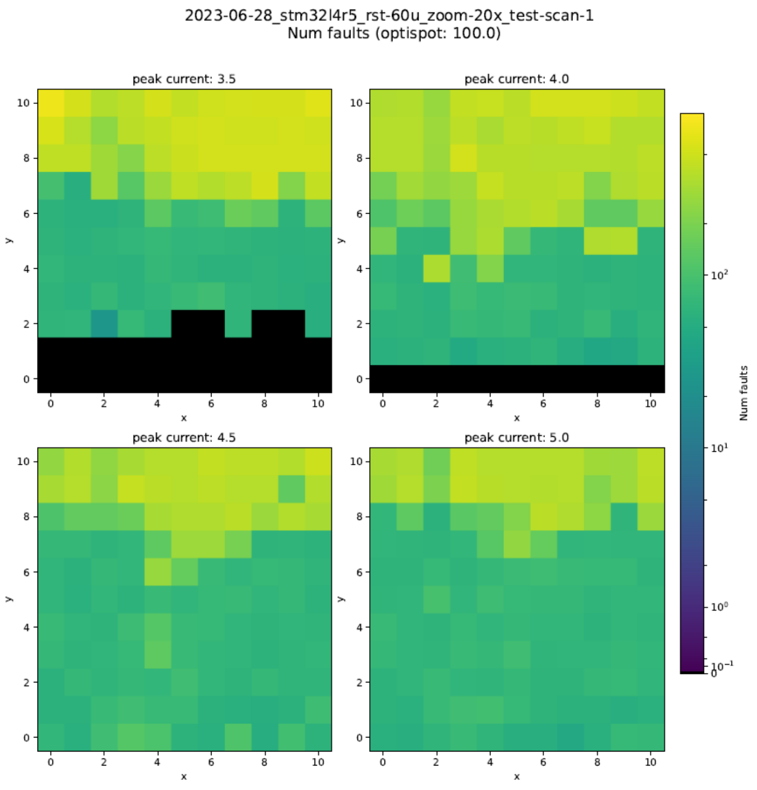


Figure 5.2: Heatmaps for the number of faults on the STM32L4 with 60µm RST with different peak currents. We did not observe any significant change to the un-thinned version.

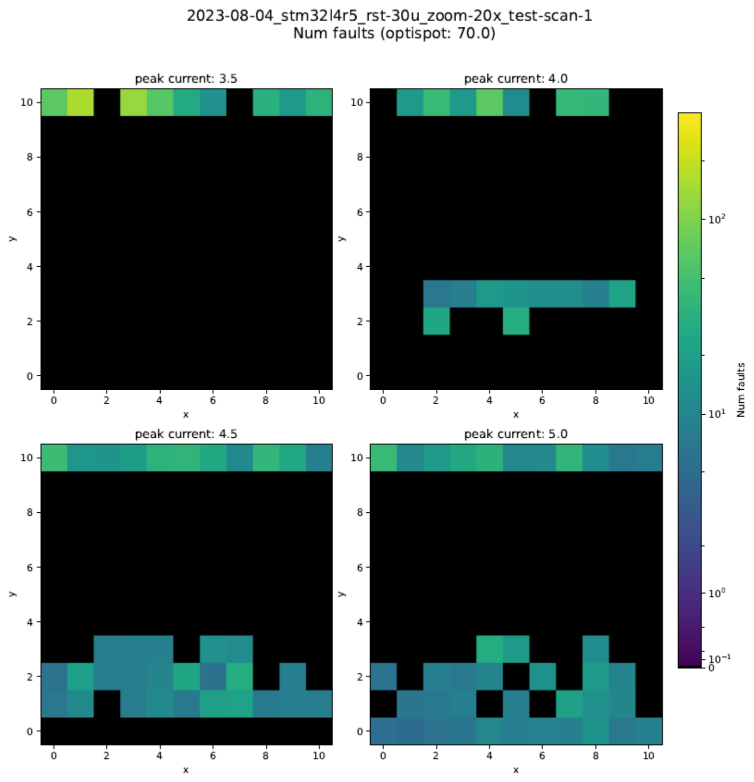


Figure 5.3: Heatmaps for the number of faults on the STM32L4 with 30μm RST with different peak currents. The number of faults was reduced, but no single bit faults were introduced.

Bibliography

[1] Alphanov: Optics & Lasers Technology Center https://www. alphanov.com/

[2] Assessing the effectiveness of mcuboot protections against fault injection attacks.

[3] IT security solutions from genua withstand attacks with quantum computers. https://www.genua.eu/knowledge-base/it-security-solutions-from-genua-withstand-attacks-with-quantum-computers , 2020

[4] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: Optimizing voltage fault injection attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):199–224, Feb. 2019

[5] BSI–Cryptographic Mechanisms: Recommendations and Key Lengths. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/ TG02102/BSI-TR-02102-1.html , January 2022

[6] Cisco. Post quantum trust anchors. https://www.cisco.com/c/dam/en_us/about/doing_business/trust-center/ docs/post-quantum-trust-anchors-wp.pdf, 2019

[7] David A. Cooper, Daniel C. Apon, Quynh H. Dang, Michael S. Davidson, Morris J. Dworkin, and Carl A. Miller. Recommendation for stateful hash-based signature schemes. Technical report, October 2020

[8] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen. XMSS: eXtended Merkle Signature Scheme. https://datatracker.ietf.org/doc/html/rfc8391, May 2018

[9] Panos Kampanakis, Peter Panburana, Michael Curcio, Chirag Shroff, and Mahbub Alam. Post-quantum LMS and SPHINCS+ hash-based signatures for UEFI secure boot.

[10] Vinay B. Y. Kumar, Naina Gupta, Anupam Chattopadhyay, Michael Kasper, Christoph Krauß, and Ruben Niederhagen. Post-quantum secure boot. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1582–1585, 2020

[11] D. McGrew and S. Fluhrer M. Curcio. Leighton-Micali Hash-Based Signatures. https://datatracker.ietf.org/doc/html/rfc8554, April 2019

[12] Guillaume Raimbault. Welcome to a new generation of future- proof TPMs: OPTIGA TPM SLB 9672. https://www.infineon. com/dgdl/Infineon-OPTIGA-TPM-SLB9672.pdf?fileId= 8ac78c8b7e7122d1017f071c3f6b00d2, February 2022

[13] Thomas Roth. TrustZone-M(eh): Breaking ARMv8-M's security, 2019

[14] Bodo Selmke, Kilian Zinnecker, Philipp Koppermann, Katja Miller, Johann Heyszl, and Georg Sigl. Locked out by latch-up? an empirical study on laser fault injection into arm cortex-m processors. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2018, Amsterdam, The Netherlands, September 13, 2018*, pages 7–14. IEEE Computer Society, 2018

[15] Alexander Wagner, Felix Oberhansl, and Marc Schink. To be, or not to be stateful: Post-quantum secure boot using hash-based signatures. In *Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security*, ASHES'22, page 85–94, New York, NY, USA, 2022. Association for Computing Machinery

[16] Alexander Wagner, Vera Wesselkamp, Felix Oberhansl, Marc Schink, and Emanuele Strieder. Faulting winternitz one-time signatures to forge lms, xmss, or sphincs+ signatures. In *Post-Quantum Cryptography - 14th International Workshop, PQCrypto 2023, College Park, MD, USA, August 16-18, 2022, Proceedings*, Lecture Notes in Computer Science. Springer, 2023

[17] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems: XMSS hardware accelerators for RISC-v. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography – SAC 2019*, volume 11959, pages 523–550. Springer International Publishing. Series Title: Lecture Notes in Computer Science