Using Fault Injection to weaken RSA public key verification

Ivo van der Elzen
University of Amsterdam
Riscure

July 10, 2018

Abstract

Embedded devices can be susceptible to Voltage Fault Injection (V-FI) attacks, which can cause data corruption. Many embedded devices also rely on RSA for their security. For example, using a signature to verify the image they boot from. When fault injection is used to modify the public modulus of an RSA key, the resulting modulus can often be more easily factored. This factorization can be used to compute a private key corresponding to the modified modulus. We study the effect of V-FI on RSA by introducing voltage glitches to an embedded device, showing which types of faults can be expected to occur. A fault suitable for attacking RSA public moduli is then selected. We simulate the selected fault in randomly-generated RSA public keys and attempt to factor them. For every key that was tested, at least one fault yielded a successful factorization within only 60 seconds. Repeating the same fault on the target is possible within minutes, allowing use of the private key obtained from the factorization to sign messages which will verify on the target.

1 Introduction

Embedded systems often rely on the RSA cryptosystem for their security. For example, RSA signatures may be used to verify the authenticity of code being executed on the system. A common use case of this is secure boot. [1] [2] [3] The code, or firmware image, will be hashed with some cryptographic hash function by the author. The resulting hash will be encrypted using an RSA private key and stored on the device. The device will use the corresponding RSA public key to decrypt the hash, and will compare the decrypted hash to a digest it computes from the code it's about to execute. While the hash comparison function is often protected against external attacks, the RSA key may not be as well protected while in transit from the device's non-volatile storage to its RAM. [1] In such a case, the RSA public key is vulnerable to attack while it's being copied between these two locations.

One of the ways that the RSA cryptosystem may be attacked is by externally inducing faults in the public modulus. Corrupting a modulus changes its value, which often makes it easier to decompose into its prime factors. With this knowledge it becomes possible to create a valid private key for this new modulus. This private key can then be used to create valid signatures under the new modulus. [4] [5]

Using Fault Injection (FI) attacks it is possible to induce faults into the data being used by an embedded system [6] [7]. This makes the above outlined RSA attack feasible. However, when using FI it may be difficult to flip arbitrarily chosen bits. Thus, we explore the limitations of Voltage Fault Injection (V-FI) with regard to what types of faults can be expected, and how those faults can best be exploited when attempting to factor RSA public keys.

This paper describes a practical application of attacking RSA public moduli by using V-FI on an embedded target. We study the limitations and possibilities of faults that can be achieved using V-FI and how those faults can be used for weakening RSA by factoring modified public moduli.

1.1 Outline

This paper is structured in the following manner: Section 2 outlines related work on the subject of FI and of weakening RSA using FI. In Section 3 we discuss the process used to obtain a fault model. This includes characterizing the target's response curve to voltage glitches, and the types of faults that can be expected. Section 4 details the effects of weakening the public modulus of RSA, both on factoring weakened moduli, and on key construction using the resulting factorizations. Section 5 discusses selecting a suitable fault model, and presents the results of factoring glitched keys. Finally, the research is discussed in Section 6 and concluded in Section 7.

1.2 Approach

Our main research question is as follows:

- Is modifying the RSA public modulus using voltage fault injection a practical means of weakening RSA signature verification?

The following sub-questions have been formulated:

- Using voltage fault injection, how can an RSA public modulus be modified in a way that is beneficial to an attacker?
- Which types of modifications reliably yield factorable moduli?
- Can we create valid private keys from these factorizations?
- Is it practical to apply this attack against RSA?

To answer these questions, a target device is characterized for its response to voltage glitches, and a realistic fault model is derived from the results. This fault model describes the possibilities and limitations of fault injection on this target relating to the type of fault that may be expected to occur within data, such as the public modulus of an RSA key. To study the effect of faults expressed by this fault model in RSA public keys, faults are simulated in randomly generated RSA keys. In this way, perturbations of the RSA public keys are created. The perturbed keys are then attempted to be factored within a practical amount of time, in the order of minutes. If the factorization of this perturbed key is completed within the preset timeout, the induced fault yielded a modulus that is factorable. For each successfully factored modulus, a private key will be derived, which may then be used to sign a counterfeit message. This message should successfully verify against the perturbed modulus. If the exact same fault is repeated on the target device, the modulus used by the target is the same and the device should accept the counterfeit signature. This constitutes a successful attack.

2 Related Work

This section outlines some of the work done on the subject of fault injection on embedded systems, and on the weakening of RSA specifically.

The effects of fault injection on embedded systems pertaining to its effects on data and control flow have been extensively studied. Spruyt demonstrated in 2012 how to build a fault model for an XMEGA microcontroller. [6] In 2013 Roscian et al. used a laser to induce faults in Static RAM (SRAM) memory cells and their work shows how to build a fault model using this primitive. [8] In the same year Moro et al. built a fault model for a 32-bit microcontroller using Electromagnetic Fault Injection (EMFI). [9] The same technique was used in 2015 by Rivière et al. to modify the instruction cache on a ARMv7-M microprocessor. [10]

This section would not be complete without mentioning the seminal paper by Rivest, Shamir and Adleman, which introduces the RSA cryptosystem. [11] This public-key cryptosystem was the first of its kind to be published, and yet it remains widely used as a fundamental building block of systems security to this day. This illustrates the robustness of the fundamental principles of RSA, its security being based on the infeasibility of factoring large integers. While to date no practical attack has been demonstrated that breaks the fundamentals of RSA, there are many attacks that can be effective if improper care is taken when implementing RSA. [12]

Work has been done on targeting RSA using FI specifically. Already in 1996 Boneh et al. outlined a fault injection attack on RSA when the Chinese Remainder Theorem (CRT) optimization was being used in the implementation. [13] This is colloquially known as the "Bellcore" attack. This attack allows for the recovery of the private key. In 2002 Aumüller et al. demonstrated a practical application of the Bellcore attack. [14]

In 2005 Seifert was the first to show that RSA can also be weakened by inducing faults in the public modulus. [4] His research shows how one could transform the public modulus n of the RSA public key into a prime number n' by inducing data faults into the public key. This makes it trivial to compute the private exponent d' for this n'. In the same year, Muir improved on Seifert's work by relaxing the restrictions for the candidate n'. [5] In Muir's work, the candidate n' does not have to be a prime number, rather it can be a composite number that is (much) easier to factor than the original n. In 2006 Brier et al. showed it is also possible to recover the private elements of the original key by inducing faults into the public modulus. [15] This further cements the need to also protect the public elements of a key before and during their use.

Bahrengi et al. demonstrated low voltage fault injection attacks against AES, RSA and other cryptosystems in 2009 and 2010. [16] [17] In their attacks they used undervolting to attack RSA to perform the Bellcore attack and the "e-th root" attack. This shows that voltage-based FI is a viable means of attacking RSA in general, but the attacks demonstrated focus on obtaining secret values (key material). In our attack, no secret material is obtained, rather we subvert signature verification.

Other primitives may also be used. In 2016 Razavi et al. demonstrated a practical attack on RSA using DRAM rowhammer to induce bit flips in public keys. Their attack was shown to be successful against both OpenSSH and GnuPG. [18] While they used rowhammer instead of more common fault injection techniques, the effect is similar and the method of obtaining a corresponding private exponent d' is the same.

At USENIX 2017 Tang et al. presented a practical attack against ARM TrustZone's usage of RSA by leveraging the embedded platform's power management features. This attack was also used to induce faults in the public modulus n. [19]

This is an area of active research. It is known that there exist various ways of attacking RSA by inducing faults, either during computation, in private values, or in the public values. It is also known that perturbing the public modulus of an RSA key often makes it easier to factor. However, it remains unclear if perturbing the public modulus while it's being copied in memory is a practical means of attack under restrictions imposed by voltage fault injection. This paper attempts to clarify this issue.

3 Obtaining a Fault Model

In this section we describe a method of obtaining a realistic fault model for our target. We induce faults into the target using V-FI in an automated way, and study the effects on three different memory copy routines running on a target device. By interpreting the resulting data we determine the types of data corruption that can be expected to occur in the public modulus of a given RSA key.

3.1 Experimental setup

The experimental setup consists of three major components:

1. A target device running test code,
2. a device that induces voltage glitches into the device, and
3. a PC controlling the voltage glitcher and recording the device's responses.

A schematic diagram of the setup is shown in Figure 1 below. The target device is a Riscure Piñata S using an ARM Cortex-M4F processor. [20] The glitching device is a Riscure Spider, coupled with a Glitch amplifier. [21]

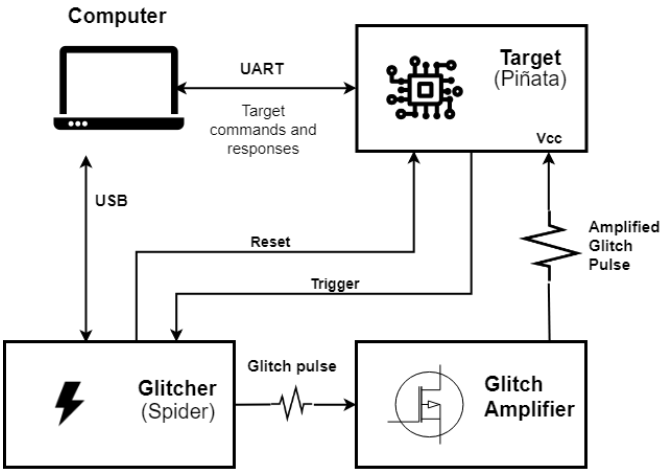

Figure 1: diagram of the experimental setup

Coupled to the experimental setup is a PC oscilloscope, used to observe signals from the target and the glitcher. This is used to determine the length of time a section of code takes, and to observe the effects of the glitches on the supply voltage of the target. A photograph of the complete setup is shown in Figure 2

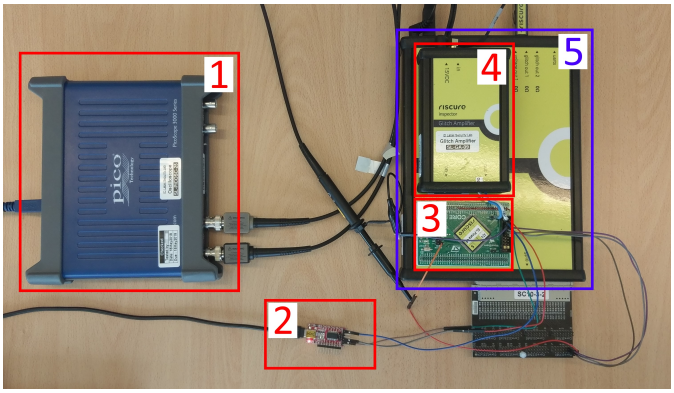

Figure 2: photo of experimental setup

The items depicted in Figure 2 are the following: PC oscilloscope (1), UART interface to PC (2), Piñata target device (3), Glitch amplifier (4), Spider glitcher (5).

3.2 Target Code

The code running on the target device is simple, it copies data between a source and destination buffer in memory. These buffers represent the locations that the public modulus of an RSA key might be copied between before verifying a signature. Studying the effect of performing FI attacks on these buffers allows us to predict the type of fault that may be induced into a real RSA key's public modulus.

To study the effects of the induced faults in a controlled way, the target needs to react in a predictable manner each time. In addition, the data being copied and the unused registers are prepared so that various types of faults may be differentiated. The target is prepared in the following way:

- A 64 byte source memory buffer is initialized with the value `0x55`.
- A destination buffer of the same size is initialized with the value `0x44`.
- Unused registers `r4` through `r12` are initialized with the hexadecimal values `0x Cn Fn Bn Dn`, where n is the register number in hexadecimal (`4` through `C`).

Usually a memory buffer is initialized with zero, making it impossible to distinguish between zeroed data and data that was already present in the destination buffer. Initializing the destination buffer with a different value allows us to distinguish between these faults. Initializing the registers with a known pattern allows us to differentiate between data being copied from the incorrect register and other faults. The 64 byte buffer is smaller than common RSA keys, but it is still sufficient for studying the effects of V-FI on the copying of data. A smaller buffer is preferable because it keeps the amount of data recorded per test, and amount of time spent outputting this data over UART, manageable. This is important when doing a great many tests.

To eliminate any side-effects from the C-library's memcpy implementation, or any optimizations the compiler may apply, the memory copy routines are coded in ARM assembly. Three variants are implemented: *byte*-wise (8 bits at a time), *word*-wise (32 bits at a time) and *multiple-word*-wise (128 bits at a time). These are implemented using ARM's `LDRB` and `STRB`, `LDR` and `STR`, and `LDM` and `STM` instructions, respectively. While not exact analogs, these implementations are a simplified form of the ARM compiler's *memcpy()* implementations. For example, the ARM *memcpy()* uses the multi-word copy for buffers up to 64 bytes. [22] The code for these loops may be found in appendix A.

Once the copy operation has completed, the target will send a start byte `0xAA`, the contents of the destination buffer (64 bytes), and then an end byte `0xBB` out on the UART. This data will be captured by the computer for later analysis. Note that these start and end bytes are not present in the registers or in the source or destination buffer.

Lastly, but perhaps most importantly, a GPIO output pin is driven high just before the copy operation begins, and then low again after it finishes (but before the data is sent out over the UART). This is the trigger signal for the Spider to know when to apply the glitch, and to know when the copy loop finishes. If the trigger signal does not fall low within a certain time limit, the Spider will reset the target by pulling its reset line low.

Normal operation of this setup should look like the following:

1. The device boots up and waits for a command.
2. The PC sends a command initiating the copy using one of the three implementations.
3. The device prepares the source and destination buffer and registers.
4. The device pulls the trigger output high.
5. The device copies the data.
6. The device pulls the trigger output low.
7. The device sends the start byte, contents of the destination buffer, and end byte.
8. The PC records the result.

When a glitch is introduced in step 5, any number of different errors may occur. Due to the careful preparation we are able to distinguish between the following possibilities:

- If the loop exits early, then the end of destination buffer contains the original data `0x55...0x44`.
- If the loop skips an iteration, a single unit of data in the destination buffer is not overwritten `0x55...0x44...0x55`.
- If data is zeroed, the buffer contains `0x00`.
- If data is copied from an incorrect register, the buffer contains one or more bytes like `0xCn, 0xFn, 0xBn, 0xDn`.
- If a single bit has been flipped, the destination buffer contains bytes with a hamming distance of one. For example `0x55` → `0x15`.
- If other data is present, it could mean that data was fetched from an incorrect memory address, or that some other error has occurred.

Other faults may occur, such as short responses, no response, random corruption, etc. This could be because the glitch had an effect on instructions that were not intended to be corrupted, meaning an effect occurred outside of the copy loop being targeted. Sending the start and end byte not as a part of the buffer but using separate instructions allows us to identify when instructions or data outside of the copy loop have been corrupted.

In this manner, faults can be induced in an automated way, their results recorded, and further processed and studied offline.

3.3 Characterization

To arrive at a fault model, a characterization must be made of how the target responds to voltage glitch pulses. This characterization describes the way a target responds to different shapes of glitch pulses, i.e. the length of time and voltage of such pulses. Using the setup and code described in Sections 3.1 and 3.2, the target is subjected to many automated, repeated voltage glitches. By recording and classifying the responses, the target is characterized. The goal is to characterize the effects of voltage glitches on a specific, pre-defined operation. In an attack scenario, the glitch should only induce a fault in the copy operation, and not when the target is processing other instructions, which would put the target in an undefined state. Therefore, it is undesirable to apply glitches to the device when it is not inside the targeted copy loop. To achieve this, the amount of time spent inside each copy loop was measured by observing the state of the trigger line using the oscilloscope. Recall that the trigger line is driven high when the copy starts, and low again when the copy is finished. Figure 3, Figure 4 and Figure 5 show the traces captured by the oscilloscope, which show the time spent for each loop variant. (Note that in these figures the comma is used as a decimal separator.)

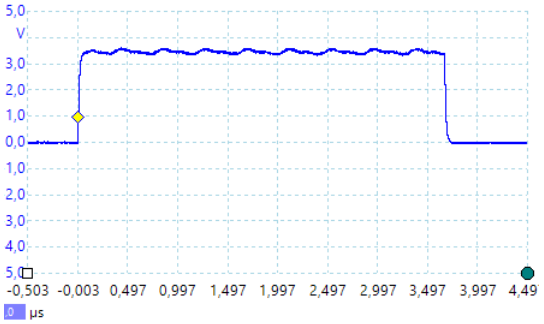

Figure 3: Timing trace of byte-wise copy loop

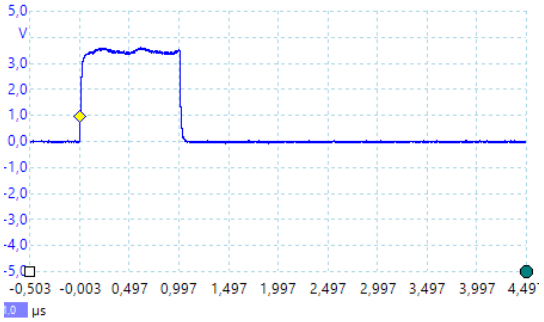

Figure 4: Timing trace of word-wise copy loop

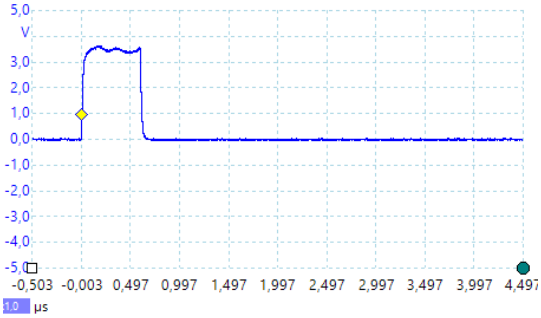

Figure 5: Timing trace of multi-word-wise copy loop

Only a single measurement is shown in these figures, but several measurements were taken to ensure the stability of the loop timings. From the traces we derive the approximate time spent in the copy loop. This is shown in Table 1.

Table 1: Copy loop timing measurement

Type	Time (ns)
byte	3650
word	1000
multi-word	570

The timing parameters used for glitching are selected to apply the glitch in approximately the middle third of each loop, attempting to avoid glitches that overrun the copy loop, potentially causing side-effects outside of the loop. If such an effect does occur, it can be identified when interpreting the data and marked as unsuitable. Therefore the timing parameters allow for small variations in loop timing. Table 2 shows the timing intervals used.

Table 2: Interval of initial glitch timing parameters

	length (ns)	delay (ns)
byte	20-1000	1250-2400
word	10-340	350-680
multi-word	10-190	150-350

In the initial experiment, 691,007 tests were performed over a period of 24 hours. In each test the glitch voltage, length, and delay from the trigger event are chosen randomly from predefined intervals. The parameters from Table 2 provide the interval for the timing parameters used. The voltage was varied between 0.1 and 3.6 volts in all tests. The responses from the target were recorded and classified based on the following criteria:

- **Expected** response: Glitch had no effect.
- **Success**: Successful glitch, the response is of correct length and includes start- and end markers, but contains different data.
- **Mute**: The target did not output any data.
- **Other**: The target gave response that was of incorrect length, missing markers, or had to be reset.

Figure 6, Figure 7 and Figure 8 show the characterization for the three copy loop variants. In these graphs, expected responses are shown in **green**, mute responses are shown in **yellow**, and glitched responses in **red**. Responses classified as "other" are not shown. Note that the scale of the Y-axis for each graph is different, because the timing parameters for each loop are different. Also, the glitch delay is not shown on the graphs.

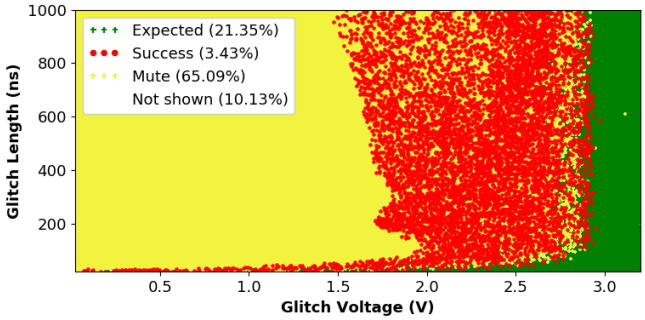

Figure 6: Characterization of byte-wise copy, 229,815 tests

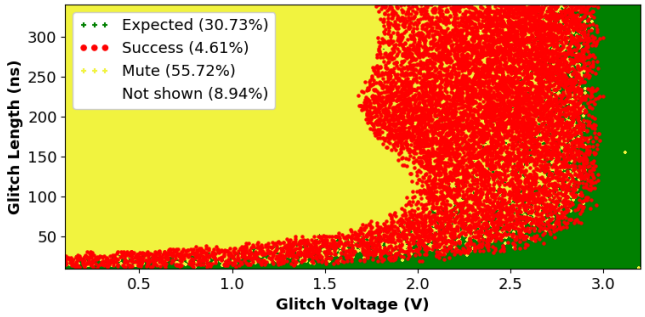

Figure 7: Characterization of word-wise copy, 230,123 tests

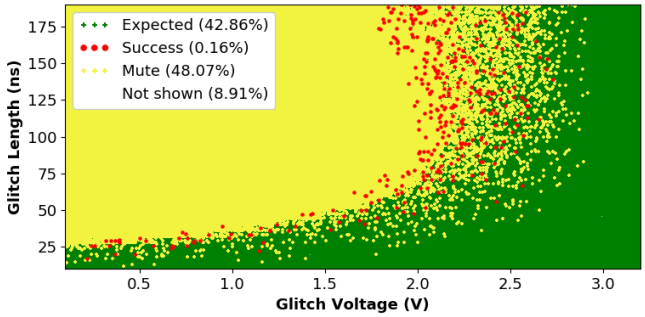

Figure 8: Characterization of multi-word-wise copy, 231,069 tests

A further improvement in success rate can be made by fixing a parameter. In this case the glitch voltage was

fixed at 2.5V. The delay and length were still varied. In this experiment 1,003,062 tests were performed over 24 hours. Figure 9, 10 and 11 show the characterization of glitch delay versus glitch length with the voltage fixed at 2.5V.

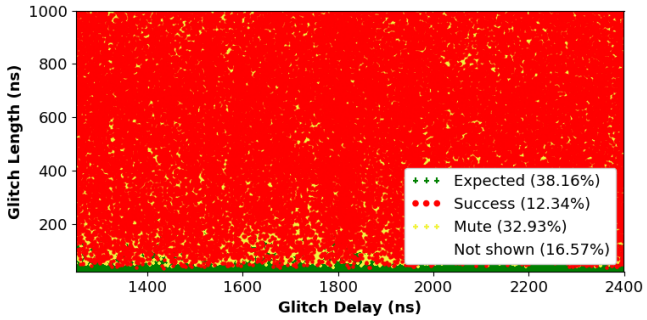

Figure 9: Characterization of byte-wise copy, 2.5V fixed, 335,269 tests

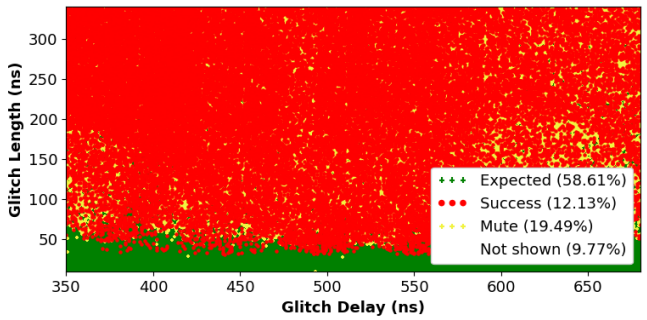

Figure 10: Characterization of word-wise copy. 2.5V fixed, 334,029 tests

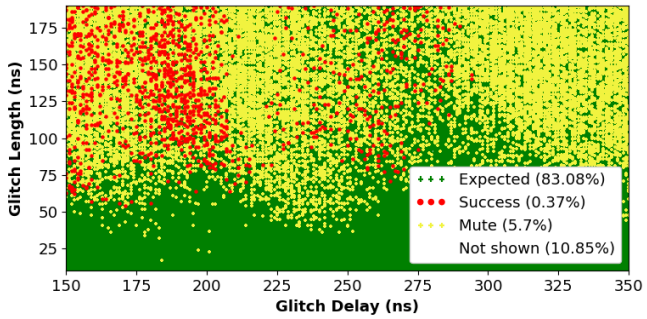

Figure 11: Characterization of multi-word-wise copy, 2.5V fixed, 333,765 tests

A clear improvement in success rate is shown. Further improvement is still possible, but this provides a workable starting point for the next experiment. It is worth mentioning that the success rate for the multi-word copy is much lower than the other variants. This could be explained by the fact that much of the "bookkeeping" of the copy such as incrementing the index registers is done in microcode, and that the implementation is much faster.

3.4 Fault Model

The voltage, length, and delay parameters learned from the characterization in Section 3.3 are further refined to improve the success rate. Specifically, the glitch length is now also fixed for all three copy loops to 112 ns for the byte and multi-word loop, and 148 ns for the word-loop. For the multi-word loop, the glitch voltage is again varied between 2 and 2.5V. Using these parameters 3,191,236 tests were performed over 66 hours. Of these tests, 205,366 desired (red) faults were observed (or 6.43%).

Using the categorization described in Section 3.2, the desired faults were classified and tallied in Table 3 below.

Table 3: Types of faults observed

Type	Amount	% of total
Early break	130,621	63.6%
Single skip	15,985	7.8%
Zeroed	4,532	2.2%
Other registers	2,957	1.5%
Flipped bits	2,103	1%
Other/mixed	49,168	23.9%
Total	205,366	100%

4 RSA with weakened moduli

RSA is an asymmetric cryptosystem which can be used to encrypt and decrypt messages using a public-private key pair. [11] The sender encrypts a message using the recipients public key, which consists of a public exponent e and a public modulus n. The recipient decrypts the message using their private key, consisting of a private exponent d and the same public modulus n. RSA may also be used for authentication (signing a message). In this case, the usage of the keys is inverted, the private key will be used to encrypt a message (usually containing a cryptographic hash of a larger message) that can be decrypted using the public key. In both cases the values e and n are public, while the private exponent d is to be kept a secret by the party using it.

For large values of n, deriving the private exponent d in a computationally feasible manner requires the prime factorization of n. Factoring a large integer composed of the product of two similarly-sized primes is currently not possible in a feasible amount of time. The security of RSA relies on these assumptions. [11] There are different methods for obtaining the values for n, e and d. In the general case they begin with choosing two prime integers p and q, the product of these integers is the public modulus n, the other values are chosen and/or derived.

However, if n can be perturbed using some means to induce faults, then the resulting integer n' is very likely no longer a product of two large prime integers. Factor-

ing this integer n' can therefore be expected to be much easier. [18] Once the factors of n' are known, a private key d' may be derived from it, which can then be used to sign messages that will verify against the key e, n'.

4.1 Factoring weakened moduli

When the public modulus is perturbed one of the following situations may occur where the modulus becomes the product of:

- Two other similarly-sized prime integers.
- Two other prime integers, one of which is small.
- More than two prime integers.
- Prime integers and prime powers.
- Or n' may become prime.

In all but the first case, factoring n' is expected to be much more feasible than n. In the case that n' is prime, factoring is trivial, requiring only a (probabilistic) primality test. In the case that n' is composed of two or more prime integers, the success of factoring is dependent on the size of their prime factors and the factoring method used. In the general case of RSA the most efficient current method to factor n into its prime factors is by using General Numeric Field Sieve (GNFS). However, this algorithm is not dependent on the size of the prime factors of n, making it inefficient when n is not composed of similarly-sized factors. [23] Since we can reasonably expect several smaller factors, more efficient algorithms exist, such as Pollard's ρ method for factors up to 60 bits, and Lenstra's Elliptic-Curve Method (ECM) for factors up to 128 bits. [24] [25]. Razavi et al. have successfully used ECM in a similar application. [18] Based on their success we chose the same algorithm. Factoring a composite integer n' with ECM can be done as follows:

1. Find a possible factor of n' using ECM.
2. Divide n' by this factor.
3. Perform a probabilistic primality test (such as Miller-Rabin [26]) on the result.
4. If the result is a composite number, repeat step 1 with step 2's output. If it is prime, n' is factored.

Note that in very rare cases the primality test may incorrectly identify a composite as a prime, this is called a pseudoprime. For our application this does not matter, because such numbers will still work correctly within RSA.

[1]Known Fermat primes are of the form $2^{2^n} + 1$ where $0 \leq n < 5$)

[2]gcd being the greatest common divisor, using Euclid's algorithm

4.2 Key Generation

The key generation algorithm for RSA starts with choosing two prime integers p and q. These integers should be about equal in size, chosen at random and large enough to yield a number that is not easily factored. Next, the values for the public and private exponents e and d should be chosen and/or computed. The method described by Rivest et al. chooses an integer d such that it is relatively prime to Euler's totient ϕ of n, and e such that $e \cdot d \equiv 1 \mod \phi(n)$. In modern day usage, e is often chosen beforehand and d is then computed to fulfill the same restriction. When e is chosen beforehand often a Fermat prime[1] is used to provide an optimization for exponentiation calculations. Certain choices of e can also weaken RSA and are best avoided. [12] In our attack, the value of e is not modified, in which case d' needs to be derived from n' and e.

Thus, the algorithm for obtaining the values n, e, d is as follows:

1. Compute the public modulus as the product of 2 prime integers:
$$n = p \cdot q \tag{1}$$
2. Compute Euler's totient of n:
$$\phi(n) = (p-1) \cdot (q-1) \tag{2}$$
3. Choose the public exponent e such that e is relatively prime with $\phi(n)$, meaning that e satisfies the following equation:[2]
$$gcd(e, \phi(n)) = 1 \tag{3}$$
4. Compute the private exponent d to be the multiplicative modular inverse of $e \mod \phi(n)$
$$d \equiv e^{-1} \mod \phi(n) \tag{4}$$

4.3 Key generation with perturbed keys

While RSA usually works with two large, similarly sized prime integers, it will work with any number of prime factors for n. (Provided the message is relatively prime to the modulus.) As described in section 4, in most cases the prime factorization of n' will result in more than two prime factors. In this case, the algorithm for computation of d' from the prime factors of n' described in section 4.2 requires a modification in step 2. For this we consider three cases:

1. If n' is prime:
$$\phi(n') = n' - 1 \tag{5}$$
2. If n' is composed of more than two prime factors:
$$\phi(n') = (p_1 - 1) \cdot (p_2 - 1) \ldots (p_r - 1) \tag{6}$$
For all factors $p_1 \ldots p_r$ in n'.

3. If n' includes prime power factors:

$$\phi(n') = p_1^{k_1-1} \cdot (p_1 - 1) \dots p_r^{k_r-1} \cdot (p_r - 1) \quad (7)$$

For all such prime power factors $p_1 \dots p_r$ in n' where $k > 1$.

If the factorization of n' contains both prime power factors ($k > 1$) and prime factors ($k = 1$), then the relevant formulas 6 and 7 are applied to each factor and their product is taken. Meaning, we take the totient of each factor and multiply them together to obtain $\phi(n')$. The rest of the steps in the algorithm to calculate d' are the same. One special case should also be mentioned where n' is not relatively prime to e. Because in our case e is fixed, another modulus should then be selected.

4.4 Note on prime power factors

If the factorization of n' results in prime power factors, there is an effect on the usability of the resulting key. RSA is only guaranteed to work for every message m that is relatively prime to n. [11] If the perturbed modulus n' is composed of one or more prime power factors (p^k), the success of a correct decryption is dependent on the divisors of the message. A relationship exists between the divisors of m and n':

Let p^k be a prime power factor of n', where k is the multiplicity of p in n' and $k > 1$:

- If $gcd(m, n') = 1$, m will decrypt properly.
- If $gcd(m, n') = p^k$, m will decrypt properly.
- If $gcd(m, n') = p^x$, where $1 <= x < k$, decryption will fail.
- If $gcd(m, n') = p_1^{k_1} \cdot p_2^{k_2} \dots \cdot p_r^{k_r}$, etc, meaning the product of any of the full factors, m will decrypt properly.

4.5 Implementation

The approach outlined in section 4.1 was implemented in a tool. The tool takes as input a public modulus n, a fault model to simulate, and a list of byte offsets to apply the fault to. The fault is applied to each chosen offset in the modulus, creating a unique modulus n' for each simulated fault. It is then attempted to factor the resulting moduli using ECM. As it is not known beforehand what amount of time a full factorization might take, a time limit was implemented. When the timeout is reached, any incomplete factoring attempts will be halted and its results discarded. If a successful factorization has been obtained, the tool calculates the corresponding private key as described in section 4.2, which may later be used for signing messages.

The tool was implemented using SAGE, a Python-based free and open-source mathematics system. [27] The underlying implementation of ECM uses gmp-ecm.[3] At the time of writing no multi-threaded implementations of ECM are known to the author, therefore it was decided to spawn one instance of ECM per available core to support multiprocessing. The total amount of time a given run takes is therefore dependent on the number of cores made available to the tool, and the number of faults to apply. The formula for calculating the worst-case total run time is as follows:

$$D = \frac{G \times T}{C} \quad (8)$$

Where total duration D is equal to the number of glitched moduli G times timeout T over number of available cores C. For example; attempting to factor all single-byte glitches for a 4096-bit modulus on a 16-core machine with a timeout of 60 seconds takes at most 1920 seconds, or 32 minutes.

$$\frac{512 \times 60}{16} = 1920$$

Given C available cores, changing the timeout allows for varying the total run time to suit a particular use-case.

The source code of this tool has been made available on GitHub[4].

5 Applying glitches to RSA keys

This section discusses selecting a suitable fault model and applying it to randomly generated RSA keys.

5.1 Selecting a Fault Model

To select the most suitable fault model for modifying RSA public moduli, the selected fault should be common, repeatable and predictable.

Early Break. As shown in section 3.4 the most common fault model with 63.6% is the early break scenario, where a copy loop exits early and leaves the last n bytes of the buffer untouched. In our experiments `0x44` was used to distinguish between exiting a loop early and having bytes set to `0x00`. However, when initializing a memory buffer it is common to fill it with zeroes (using *calloc()* or *memset()* for example), so we assume this to be the case in our experiments. When an integer has its last byte set to `0x00`, it will add 2^8 as a factor. When the last 2 bytes are set to `0x0000`, it will add 2^{16} and so on. As discussed in section 4.4, these are not ideal because not every message will decrypt correctly when a factorization includes prime power factors.

Skip loop iteration. The next most common fault with 7.8% is skipping a loop iteration. This fault has a predictable effect where a single unit of data in the destination buffer is not copied over. The fault where a single

[3] `http://ecm.gforge.inria.fr/`

[4] `https://github.com/ivovanderelzen/GlitchRSA/`

byte or word is zeroed can be grouped in this category because it has the same effect on the data if the destination buffer was zero-initialized.

Other faults. The other fault models are less suitable because they are less common and/or not as predictable. Register contents can be useful only if the contents of the registers is known and predictable. In our experiments the unused registers were set to known values, but this is not the case on a real-world target. Memory content faults are only useful when memory contents are known and stable. Single bit flips might be suitable but they are almost 8 times less common than iteration skips with 1% of the total. Mixed faults where multiple fault types occur at the same time are also not predictable.

Repeatability of single byte skip. In a separate experiment, it was attempted to target a single loop iteration of the byte-wise copy loop. In this experiment, 1,160,778 tests were performed. Of these, in 1946 tests the chosen loop iteration was skipped, or approximately 1.7 in a 1000. We calculate the number of test needed to have a 95% chance of hitting the targeted byte thusly: $\frac{\ln(1-0.95)}{\ln(1-0.0017)} = 1760.7$. Rounding this up it means that 1761 tests need to be taken to reach a 95% probability of hitting the desired loop iteration. With the observed glitch rate of 12 per second, this will take 147 seconds, or approximately 2.5 minutes. Using a more conservative glitch rate of one per 10 seconds, the time needed is approximately 292 minutes, or under 5 hours. This means that such a targeted single loop iteration skip is feasible.

Selected fault The fault model "skip loop iteration" is the most-common fault model which is not guaranteed to result in prime power factors, and it can be precisely targeted. Therefore it has been selected as a fault model to use for factorization testing.

5.2 Factorization testing.

Based on the fault model selected in section 5.1, randomly generated RSA keys had a simulated fault applied and a factorization was attempted using the tool described in section 4.5. Each time a test was run, a keysize was randomly chosen from 512, 1024, 2048 and 4096 bits and a key of that size was generated using OpenSSL. For each generated key, a fault model was randomly chosen from zeroing out a single byte, a single word (4 bytes) and a multi-word (4 words). This fault was then applied to each unit of data inside the modulus, yielding multiple moduli per key. Each of those resulting moduli were then attempted to be factored within a time limit of 60 seconds. The time limit was based on the success of similar work using the same timeout by Razavi et al. [18] The number of perturbed moduli per key that are attempted to factor is dependent on the key size and fault model. For example, a 512 bit key with single-byte fault model will produce 512/8 = 64 different moduli. A 4096-bit key with the same test will produce 4096/8 = 512 moduli. Using multi-word, the same key size will produce 4096/128 = 32 moduli. In this way, from 1234 unique randomly generated RSA keys, 146,512 unique perturbations of those keys were tested. When a modulus is factored within 60 seconds, it is considered a successful factorization. The number of successful and unsuccessful factoring attempts were recorded. The breakdown of key sizes by fault models tested are shown in Table 4.

Table 4: Number of keys tested per fault model

Fault \ Size	512	1024	2048	4096	Total
Byte	127	134	132	115	508
Word	124	111	109	95	439
Multi-Word	88	74	66	59	287
Total	339	319	307	269	1,234

Figure 12, Figure 13 and Figure 14 show a box plot of the average number of successes per key by key size, meaning how many perturbed moduli were factored within 60 seconds for any given key.

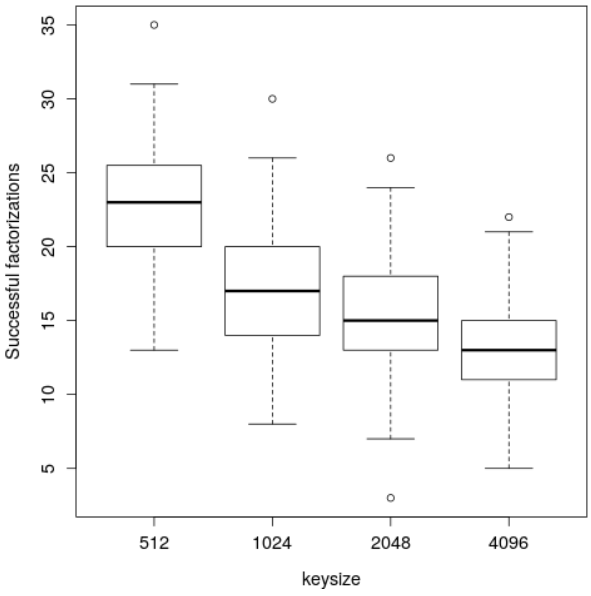

Figure 12: Factoring success rate for single-byte fault

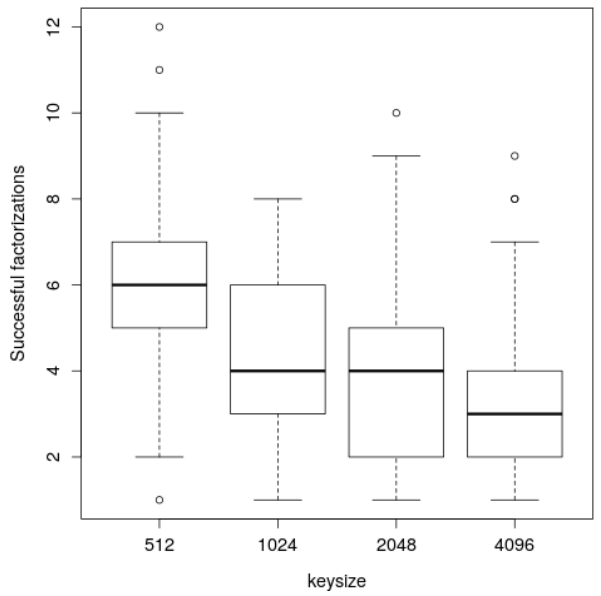

Figure 13: Factoring success rate for single-word fault

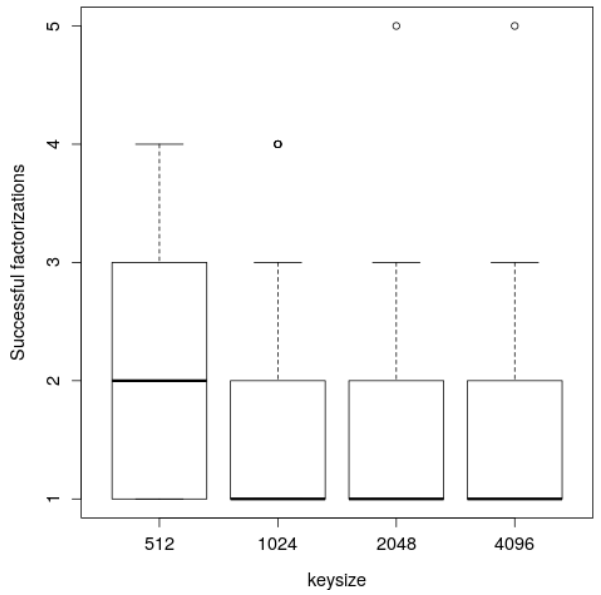

Figure 14: Factoring success rate for multi-word-fault

It is worth noting that while only 11,150 of 146,512 different moduli, or 7.6% were successfully factored within 60 seconds, **all** of the attempted keys had *at least* one successful factorization. This means that even with a timeout of only 60 seconds, there is a high probability for finding at least one successful factorization. Depending on the fault model used, it is very likely that several modulus perturbations will result in successful factorizations.

6 Discussion

This research shows that with a limited investment in time and resources, RSA keys can be perturbed on an embedded device using V-FI and the resulting moduli factored. However, several limitations apply, which will be discussed in this section.

To perform the glitching experiments we used proprietary hardware and software, described in Section 3.1. However, this attack should also be possible with cheaper, open source hardware, such as a ChipWhisperer[5]. Using such hardware will lower the initial financial investment needed for doing V-FI research. Also, the target device used in this research is a commercially available training target prepared to facilitate FI attacks. (Power supply filtering removed, easy access to signals, etc.) Other target devices could also be used, although they will have to be prepared in a similar way to the target device used in this research.

In Section 3.3 it is demonstrated that data can be modified while it is being copied from one location in memory to another by inducing a fault by means of V-FI. This research is based on the assumption that the public modulus of an RSA key is at some point copied between memory locations before use. If the modulus is not copied before use, this attack will not be successful. However, at some point the public key data will have to be loaded from non-volatile memory. In such a case this operation could be attacked. Other cases may exist such as a PEM-encoded public key first being base 64 decoded and then ASN.1 decoded before the values are used. In such a case this process could be attacked.

The ARM assembly code used for copying data between the memory buffers are simplified versions of assembly code that might be emitted by a C compiler when compiling a *memcpy()* function. Due to time constraints, it was not possible to verify the results against an implementation of RSA signature verification on the target device. The actual implementation of RSA and the handling of public key material may differ between cryptographic libraries. In some cases a custom implementation of *memcpy()* might be used, or there might not be a single function copying the data. Given a target's sensitivity to V-FI, it should still be possible to induce faults into the data being copied, but a fault model should be obtained for each implementation.

In Section 3.4 we show what types of data faults can be achieved, and describe them in a fault model. When obtaining this fault model, the target was entirely under our control. This allows for easily setting triggers around targeted code. It also gives insight into all of the code running on the target and how it may affect the desired glitch. With real-world targets, one doesn't always have this luxury. Other methods of determining timing parameters exist, either by observing the target's outputs directly, or by techniques such as Side-Channel Analysis (SCA).

In the fault model it is assumed that the destination buffer is initialized to zero before use. This is a common practice but it may not be true in all cases. When a buffer is not initialized with zero, the contents may be different.

[5] `http://newae.com/tools/chipwhisperer/`

In such a case the same attack can be performed, but the contents of the destination buffer before copying need to be known.

Evaluating the usefulness of the fault models in Section 5.1, we find that the most beneficial modification an attacker can make to an RSA public modulus is caused by skipping a single iteration of a copy loop. This fault model provided flexibility in type and location of the fault. Given a more restricted fault model, can similar success be achieved? As shown in Figure 14, in the most restricted fault model case tested, using multi-word zeroing using 4096 bit keys the mean success rate was one successful factorization out of 32 attempts per key. However, given a mean factoring success rate of 14 out of 512 for the single byte fault (as shown in Figure 12) the success rate is approximately 1 in 36.5, which is similar. More factorization testing and statistical evaluation of the results are needed to conclusively answer this question.

Section 4.1 describes that when a modified modulus is factored, it is not likely to be a product of two large prime integers. In some cases, the factorization will contain prime power factors. When this is the case, the resulting key will not work for all messages, this is detailed in Section 4.4. In such a case a different factorization should be selected, or if that is not possible, the message may be modified to fit the key, for example by appending data.

6.1 Mitigations

Even though it is generally assumed that an RSA public key does not have to be kept secret, it should still be protected against FI attacks. Generic countermeasures against FI apply, which includes hardware countermeasures and software countermeasures. Hardware countermeasures can include physical tamper-proofing, additional power supply filtering, local chip capacitance, voltage drop-out detection, storing sensitive data in special areas of memory, and others. Hardware countermeasures are discussed by Bar-El et al. [28] Software countermeasures against FI also exist, perhaps most importantly double- and triple-checking sensitive data before use. An overview of software countermeasures against FI is provided by Witteman. [29]

7 Conclusion

We have examined the effects of Voltage Fault Injection on the security of RSA public key verification, and evaluated the practicality of this attack.

When data is being copied between memory locations on a target embedded device, faults can be induced into this operation using Voltage Fault Injection. This is demonstrated in Section 3. In Section 4 we show that if the data being copied holds the public modulus of an RSA key, the modulus will be modified and thereby its security weakened.

For a fault to be of benefit to the attacker, it should be both predictable and repeatable. One such fault is causing a copy loop to skip a single iteration, resulting in a single unit of data having the original buffer contents. In Section 5.1 we show that this fault can be reproduced in a targeted way, within minutes.

Using a fault model of zeroing a single unit of data, some modified RSA public moduli can be easily factored. In Section 5.2 we show that we successfully obtain factorizations of all keys attempted, using a timeout of only 60 seconds.

Slightly adapting the RSA key generation algorithm, private keys can be constructed that correspond to the modified public modulus. If a target holds the same modulus in memory as a result of V-FI, messages signed using this key will verify correctly.

Given that using Voltage Fault Injection a desirable fault can be introduced into an RSA public modulus at a precise location and within a reasonable amount of time, and given that at least one perturbed modulus of each key can be factored within 60 seconds, we find that modifying the RSA public modulus using voltage fault injection is indeed a practical means of weakening RSA signature verification.

8 Future Work

Use open-source hardware. To lower the financial investment required to perform this attack, it is suggested to attempt reproducing the V-FI part of this research using open-source glitching hardware and software.

Study effect of multi-prime RSA on signature schemes. Various RSA signature schemes exist. We suggest looking into the effect on various signing schemes PKCS#1 v1.5, RSA-PSS, RSA-OAEP, etc. In any case, standard RSA-CRT signature calculation will not work, because it uses coefficients based on two prime factors.

Implement signature verification on target. A next step in this research is to implement RSA signature verification on the target, and to attempt to verify a counterfeit signature using the attack described in this paper.

Attack a real-world secure boot implementation. Applying this attack against one or more secure boot implementations will provide valuable insight on the impact of this attack.

Factoring testing. More varied fault models and longer timeouts could be tested to evaluate the feasibility of factoring under different conditions.

9 Acknowledgements

The author would like to thank the following people for their contribution to this research:

- Ronan Loftus, for supervising and providing guidance.

- Baris Ege, for providing valuable mathematical insights.
- Albert Spruyt, for assisting with the experimental setup.

I would also like to extend my thanks to Riscure for hosting this project, and to all the wonderful people at Riscure who made my short tenure there a very pleasant one.

References

[1] ARM Limited. ARM TrustZone Software Architecture - Secure boot, 2009. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/CACGCHFE.html [Accessed July 2018].

[2] NXP B.V. Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HABv4, 2018. https://www.nxp.com/docs/en/application-note/AN4581.pdf [Accessed July 2018].

[3] Unified EFI, Inc. Unified Extensible Firmware Interface Specification Version 2.3.1, Errata C, 2012. https://www.uefi.org/sites/default/files/resources/UEFI_2_3_1_C.pdf [Accessed July 2018].

[4] Jean-Pierre Seifert. On authenticated computing and RSA-based authentication. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 122–127. ACM, 2005.

[5] James A. Muir. Seifert's RSA Fault Attack: Simplified Analysis and Generalizations. Cryptology ePrint Archive, Report 2005/458, 2005. https://eprint.iacr.org/2005/458.

[6] Albert Spruyt. Building fault models for microcontrollers. SNE Research Project, 2012. http://rp.delaat.net/2011-2012/p61/report.pdf.

[7] James Gratchoff. Proving the wild jungle jump. SNE Research Project, 2015. http://rp.delaat.net/2014-2015/p48/report.pdf.

[8] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault model analysis of laser-induced faults in SRAM memory cells. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 89–98. IEEE, 2013.

[9] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE, 2013.

[10] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of ARMv7-M architectures. In *Hardware oriented security and trust (host), 2015 ieee international symposium on*, pages 62–67. IEEE, 2015.

[11] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[12] Dan Boneh et al. Twenty years of attacks on the RSA cryptosystem. *Notices-American Mathematical Society*, 46:203–213, 1999.

[13] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking computations. 1996.

[14] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and J-P Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 260–275. Springer, 2002.

[15] Eric Brier, Benoît Chevallier-Mames, Mathieu Ciet, and Christophe Clavier. Why one should also secure RSA public key elements. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 324–338. Springer, 2006.

[16] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low voltage fault attacks on the RSA cryptosystem. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, pages 23–31. IEEE, 2009.

[17] Alessandro Barenghi, Guido Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. Low Voltage Fault Attacks to AES and RSA on General Purpose Processors. *IACR Cryptology ePrint Archive*, 2010:130, 2010.

[18] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium*, pages 1–18, 2016.

[19] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, 2017. USENIX Association.

[20] Riscure. Piñata Datasheet, 2017. https://www.riscure.com/uploads/2017/07/pi%C3%B1ata_board_brochure.pdf [Accessed July 2018].

[21] Riscure. Spider Datasheet, 2017. https://www.riscure.com/uploads/2017/07/spider_datasheet_1.pdf [Accessed July 2018].

[22] Arm Limited. How do the ARM compilers handle memcpy()?, 2011. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3934.html [Accessed July 2018].

[23] Arjen K Lenstra, Hendrik W Lenstra, Mark S Manasse, and John M Pollard. The number field sieve. In *The development of the number field sieve*, pages 11–42. Springer, 1993.

[24] John M Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[25] Hendrik W Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.

[26] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

[27] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.1)*, 2017. http://www.sagemath.org.

[28] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[29] Marc Witteman. Secure application programming in the presence of side channel attacks, 2017. https://www.riscure.com/uploads/2017/08/Riscure_Whitepaper_Side_Channel_Patterns.pdf [Accessed July 2018].

Appendices

A Copy Loop Assembly Listings

In all variants r2 points to the source buffer, and r3 points to the destination buffer.

A.1 Byte-Wise LDRB Version

```
mov         r0, #0
loop8:
    ldrb    r1, [r2]
    strb    r1, [r3]
    add     r2, r2, #1
    add     r3, r3, #1
    add     r0, r0, #1
    cmp     r0, #64
    bne     loop8
```

A.2 Word-Wise LDR Version

```
mov         r0, #0
loop32:
    ldr     r1, [r2]
    str     r1, [r3]
    add     r2, r2, #4
    add     r3, r3, #4
    add     r0, r0, #4
    cmp     r0, #64
    bne     loop32
```

A.3 Multiple-Word LDM Version

```
mov         r0, #0
loopm:
    ldm    r2!, {r9, r10, r11, r12}
    stm    r3!, {r9, r10, r11, r12}
    add    r0, r0, #16
    cmp    r0, #64
    bne    loopm
```

Note that the index registers in the `LDM/STM` variant are not explicitly increased, because the assembly instructions already provide this functionality.