TAMPERE UNIVERSITY OF TECHNOLOGY

Memory Tampering Attack on Binary GCD Based Inversion Algorithms

Citation
Aldaya, A. C., Brumley, B. B., Sarmiento, A. J. C., & Sánchez-Solano, S. (2018). Memory Tampering Attack on Binary GCD Based Inversion Algorithms. *International Journal of Parallel Programming*, 1-20. https://doi.org/10.1007/s10766-018-0610-x

Year
2018

Version
Peer reviewed version (post-print)

Link to publication
TUTCRIS Portal (http://www.tut.fi/tutcris)

Published in
International Journal of Parallel Programming

DOI
10.1007/s10766-018-0610-x

License
Other

Take down policy
If you believe that this document breaches copyright, please contact cris.tau@tuni.fi, and we will remove access to the work immediately and investigate your claim.

Download date:25.09.2020

This is a post-peer-review, pre-copyedit version of an article published in International Journal of Parallel Programming. The final authenticated version is available online at: https://doi.org/10.1007/s10766-018-0610-x

Memory tampering attack on binary GCD based inversion algorithms

Alejandro Cabrera Aldaya ·
Billy Bob Brumley ·
Alejandro J. Cabrera Sarmiento ·
Santiago Sánchez-Solano

Abstract In the field of cryptography engineering, implementation-based attacks are a major concern due to their proven feasibility. Fault injection is one attack vector, nowadays a major research line. In this paper, we present how a memory tampering-based fault attack can be used to severely limit the output space of binary GCD based modular inversion algorithm implementations. We frame the proposed attack in the context of ECDSA showing how this approach allows recovering the private key from only one signature, independent of the key size. We analyze two memory tampering proposals, illustrating how this technique can be adapted to different implementations. Besides its application to ECDSA, it can be extended to other cryptographic schemes and countermeasures where binary GCD based modular inversion algorithms are employed. In addition, we describe how memory tampering-based fault attacks can be used to mount a previously proposed fault attack on scenarios that were initially discarded, showing the importance of including memory tampering attacks in the frameworks for analyzing fault attacks and their countermeasures.

Keywords fault attacks · binary GCD · bitstream manipulation · FPGA memory tampering · ECDSA

1 Introduction

Implementation attacks are a threat to cryptographic devices, aiming at recovering secret data exploiting implementation inherent characteristics. Several implementation attack

Alejandro Cabrera Aldaya (✉)
Universidad Tecnológica de la Habana José Antonio Echeverría (CUJAE), La Habana, Cuba
email: aldaya@gmail.com

Billy Bob Brumley
Tampere University of Technology, Tampere, Finland
email: billy.brumley@tut.fi

Alejandro J. Cabrera Sarmiento
Universidad Tecnológica de la Habana José Antonio Echeverría (CUJAE), La Habana, Cuba
email: alex@automatica.cujae.edu.cu

Santiago Sánchez-Solano
Instituto de Microelectrónica de Sevilla, IMSE-CNM, (CSIC/Universidad de Sevilla), Seville, Spain
email: santiago@imse-cnm.csic.es

vectors have been proposed in the literature [12, 23, 24, 30]. Among them, fault injection has received generous attention in the scientific community due to the real-world security threats this class represents [6].

FPGA are very attractive for implementing cryptographic algorithms due to several factors such as: its inherent parallel execution, use of custom datapath, and on-field reconfigurability features [13, 15, 18, 33]. These devices are well-suited for different scenarios of applications like high-performance computing hardware accelerators and embedded systems. For this reason FPGAs are a major target for implementation attacks, in particular, some research lines focus on dedicated features of these devices from the attacker point of view [2,34,38]. For example, in [34] the authors investigate side-channel leakages of different parts of the embedded FPGA block memories, where in [2] these memory elements are analyzed regarding their susceptibility to fault injection attacks.

Fault injection attacks focus on perturbing the execution of a cryptographic algorithm for extracting secret data by analyzing the algorithm's output in the presence of a fault [6]. Several methods for injecting faults during an algorithm execution exist, including but not limited to: clock signal glitching, electromagnetic pulses, Focused Ion Beams (FIB), and laser shots [6, 36]. Recently in [2], the authors show how to recover an AES secret key by means of tampering FPGA embedded block memory. The authors in [36] argue that fault injections via memory tampering is feasible on some FPGA platforms.

However, in general terms, modular inversion algorithms, in particular those based on the binary greatest common divisor (binary GCD) algorithm, receive less attention in the scientific community regarding implementation attacks, even though they are heavily employed in several cryptographic schemes and even used for implementing countermeasures [10, 21, 28, 39]. Inspired by previous work [3] on side-channel analysis of the Binary Extended Euclidean Algorithm (BEEA), this work shows how a memory tampering technique can be used to severely limit the output space of binary GCD based modular inversion algorithms in cryptographic implementations, and subsequently shows how to exploit this limited output space to recover private keys in cryptosystems utilizing these algorithms.

The main contributions of this work are the following:

1. We show how to use a memory tampering-based fault attack to considerably limit the output space of binary GCD based modular inversion algorithms.
2. We use the proposed memory tampering-based fault attack to recover private keys from an ECDSA implementation, were a binary GCD based inversion algorithm is employed (Section 4). To the best of our knowledge, this is the first work analyzing public key cryptographic algorithms regarding FPGA memory tampering attacks.
3. The presented attack only requires a single signature to succeed independent of the key size while the analysis in [3] and [17] require several more ones and are key-size dependent. Furthermore, we show how this attack can be adapted to different implementations (Section 4).
4. In addition, we show how, by means of memory tampering, previously proposed fault attacks can be implemented in scenarios that were initially discarded (Section 5.1).

This paper is organized as follows. Section 2 gives general background on memory tampering as an implementation attack vector, presenting an application scenario where the proposed attack can be applied against a remote FPGA implementation. Section 3 presents the BEEA to compute modular inversions, introducing some useful notation. Section 4 describes our technique to limit the modular inversion output space by means of a tampering attack, with two different approaches that both fall under the same

threat model. Section 5 describes the ECDSA algorithm and analyzes some fault attacks against it w.r.t. the proposed memory tampering technique, subsequently leading to ECDSA private key recovery. Section 6 discusses some countermeasures for protecting an implementation against the proposed attack. Conclusions are presented in Section 7.

2 Tampering FPGA embedded memories

In contrast to *classic* fault attacks such as glitching [11], FPGA memory tampering is relatively a new attack vector. The authors of [2] showed how the embedded block memory content of some FPGA architectures can be easily tampered using publicly documented tools. Regarding this topic, during last few years, several works examine how to generally apply this technique, increasing the number of FPGA platforms from different vendors threatened by this attack [36,37,38], even including real-world applications [37]. Furthermore, these techniques can be used to mount powerful fault attacks against cryptographic algorithms [36].

In general terms, these bitstream memory tampering procedures require an unencrypted and unauthenticated bitstream [2,38]. Therefore, if any of these protections were present, the adversary should find a way to bypass them first. In this regard, successful side-channel attacks targeting these mechanisms have been proposed in literature [25,26]. In addition, in [36] some results were presented about tampering encrypted bitstreams. However, it is important to highlight that this technique generates random tampering content instead of a deterministic one, as required in the attack proposed in this manuscript (see Sect. 4 for details).

This work assumes an adversary capable of tampering with a cryptography constant: the value of p in a modular inversion implementation. This is possible, for example, in an FPGA implementation where the value is stored in an embedded block memory, making its tampering very straightforward as presented in [2] and described in Appendix A. However, if this value is stored in other FPGA hardware primitives (like look-up tables, LUTs), the work in [36,38] validates that this modification could be possible to perform fault attacks [36].

FPGA embedded memory tampering based fault attacks have several advantages over other fault injection approaches such as glitching. Each fault injection technique has an associated unsuccessful injection probability. In the case of clock glitching this probability is, in general, non-negligible because it depends on several variables not controlled by the attacker. For example, a clock glitch during the execution of an instruction in a microprocessor could change the instruction opcode in an unpredictable way [5,27]. However, the injection of a fault by means of memory tampering on FPGAs is very reliable as the memory replacement procedure is deterministic and free of errors [2,37,38]. For the sake of completeness, the memory tampering procedure employed in this work is detailed at Appendix A. We have tested said procedure on Spartan-3 and Spartan-6 FPGAs, however the documentation indicates that it applies to other families as well, like Virtex and modern 7 Series [40,41].

2.1 Target application examples

Usually, implementation attacks on FPGA platforms are considered in embedded system settings where the adversary has physical access to the targeted device, hence access to the non-volatile memory where bitstream is stored [2]. However, reports on remote attacks

on FPGA also exist [19]. Dynamic partial reconfiguration (DPR) is a powerful feature with demonstrated advantages for enhancing implementation performance. Besides, some authors consider it a double-edged sword for cryptographic applications, as it allows some attacks like faults injection and hardware trojan horses [7, 31].

These features and attack vectors allow the application of memory tampering-based fault attacks not only where the adversary has physical access to the targeted device but to remote applications as well. For example, consider the scenario with an FPGA hardware cryptographic accelerator on a high-performance computing platform with DPR capabilities for upgrade purposes [15]. The adversary has no physical access to this system, but due to a software defect exploit, it gains remote access to the DPR interface and can dynamically reconfigure the FPGA with a maliciously crafted bitstream.

Regarding a memory tampering attack, if the adversary has a copy of the original bitstream the application of the presented attack is straightforward as this scenario is very similar to the physical access scenario detailed in [2] (see Apprendix A for details). However, relaxing this constraint and considering the adversary does not have access to the original bitstream, even in this setting, the attack can succeed. For example, the adversary can generate partial bitstreams where the reconfigured parts are only embedded block memory contents. Then it can perform a *blind* remote fault injection attack, guessing by trial-and-error tests where the targeted BEEA memory content is located (similar to BiFI attacks introduced in [36]). Finally, the adversary can apply the attack described later in Sect. 4.1. This methodology does not guarantee success, but is a plausible attack scenario. The authors of [36] recently proposed this form of *blind* fault attack, showing how to inject a successful fault into unknown bitstreams in a fully automated setting.

These examples show that memory tampering fault attacks through bitstream manipulation are not only a concern for embedded systems, but extends to remote FPGA applications as well.

3 Binary GCD based modular inversion algorithm

In 1967, Stein proposed a binary algorithm for computing the greatest common divisor of two integer numbers [35]. Its main advantage over classical Euclidean algorithms is that division operations in the latter are replaced by simple right-shift operations. This proposal offers very good performance in hardware and software platforms especially when the algorithm's inputs are large [14, 22].

Stein's algorithm extends to compute modular multiplicative inverses; this algorithm is often known as Binary Extended Euclidean Algorithm (BEEA) and Algorithm 1 shows a pseudocode.

Cryptographic algorithms such as RSA and ECDSA [28] often need modular inversions. In addition, some leakage countermeasures [10, 21, 39] even employ modular inversions. We perform our analysis on the binary GCD based algorithm shown in Algorithm 1 (BEEA) in the context of inverting an integer k modulo p. Without losing generality, we considered one algorithm input is secret (k) while the other (p) is known, as it is a typical scenario in cryptographic applications such as RSA and ECDSA [3, 4].

It is worth noting that our results are not strictly limited to the BEEA depicted in Algorithm 1 because the approach easily extends to other variants, like Kaliski's modular inversion algorithm [20] and recent constant-time implementations [9, 32].

Algorithm 1 Binary Extended Euclidean Algorithm (BEEA)

Inputs: Integers k and p such that $\gcd(k, p) = 1$
Output: $k^{-1} \mod p$

```
1: u = k
2: v = p
3: A = 1
4: C = 0
5: while u ≠ 0 do
6:    while even(u) do                 ⎫
7:       u = u / 2                     ⎪
8:       if even(A) then               ⎬ u-loop
9:          A = A / 2                  ⎪
10:      else                          ⎪
11:         A = (A + p)/2              ⎭
12:   while even(v) do                 ⎫
13:      v = v / 2                     ⎪
14:      if even(C) then               ⎬ v-loop
15:         C = C / 2                  ⎪
16:      else                          ⎪
17:         C = (C + p)/2              ⎭
18:   if u ≥ v then                    ⎫
19:      u = u - v                     ⎪
20:      A = A - C                     ⎬ sub-step
21:   else                             ⎪
22:      v = v - u                     ⎪
23:      C = C - A                     ⎭
24: return C mod p
```

In Algorithm 1 some steps are grouped and labeled following the notation in [3]. The *u-loop* (resp. *v-loop*) corresponds to the loop that divides variable u (resp. v) by two, whereas the *sub-step* labels the execution of subtraction operations.

3.1 BEEA execution flow related information

In the field of side-channel attacks, binary GCD based algorithms are an attractive target due to highly input-dependent execution flow [1, 3, 17]. The presented attack does not rely on the existence of any side-channel leakage. However, we borrow some notation and analysis from these works that prove useful for our proposed attack.

An execution of the BEEA (regarding variables u and v) results in a series of division by two operations of the variables u or v followed by one of the subtractions at Step 19 or Step 22. Labeling each division by two with a capital **L** and each subtraction with a capital **S**, Fig. 1 illustrates an execution flow example regarding these variables.

LLLSLLSLSLLLLLSL...LS

Fig. 1 BEEA execution flow example regarding variables u and v.

From previous side-channel analysis literature for the BEEA, its execution flow can be characterized by two variables (Z_i and X_i). Our work requires only the definition of Z_i. Each iteration i has its corresponding Z_i, representing the number of division by two executed at ith-iteration (i.e. number of **L** between two **S** in Fig. 1). For example, in Fig. 1, the first three Z_i are: 3, 2, and 1.

4 Memory tampering attack against BEEA

This section presents a fault attack (based on FPGA embedded memory tampering) such that it is possible to fully recover the BEEA output with a single execution. This section adopts the fault attack model of [36, Sect. 2.2]:

> We suppose that the adversary has read- and write-access to the external memory which stores the bitstream of the target device. Consequently, he can arbitrarily modify the FPGA embedded block memory sections in the bitstream, and observe the FPGA behavior accordingly.

Concretely applying this to our proposed attack leads to the following *adversarial model*: an adversary can choose the modulus employed for computing a modular inverse with the BEEA. Following this model, this section furthermore assumes that an attacker *chooses* one of the BEEA inputs through tampering the value of p.

Note that the application scenarios described in Sect. 2.1 fulfill this adversarial model. However, the bitstream manipulation attack proposed in [36] against encrypted bitstream does not apply as it generates random tampering values while the assumed adversarial model requires *choosing* it.

A procedure for reading and tampering Xilinx FPGAs embedded block memories is described in Appendix A. As this procedure can be used to read the content of all embedded block memories, it can disclose by itself some secret values stored in them (e.g. keys). However, we assume only public information is stored in these elements, as the former scenario does not add a valuable scientific contribution.

In addition, it is worth noting that this scenario does not limit the proposed attack to FPGA platforms and bitstream manipulation attacks, as there could be other means to achieve (or mimic) the desired memory tampering [24].

4.1 Attack description: selecting good tampering values

Memory tampering-based fault attacks, like the ones described in Sect. 2, belong to the permanent fault attack category [11] because their effects persist during the whole algorithm execution. This means the fault might affect not only the targeted algorithm part but others as well.

This section covers two memory tampering-based fault attack cases taking into account if the fault only affects the modular inversion operation, or if it affects other arithmetic operations like, for example, the final modular reduction in Algorithm 1. **Case 1** analyzes the former scenario, where the tampering is *local* to the BEEA, whereas **Case 2** analyzes a *non-local* to BEEA scenario. This distinction is interesting since some implementations, especially on hardware platforms, duplicate the storage of the modulus p to achieve better performance [18]. However, we feel it is more important to view this case distinction as an example of how the presented attack can be adapted to different scenarios and implementation constraints.

Case 1 (local to BEEA fault): One plausible method of recovering the secret k by tampering the value of p consists of replacing p by a low bit-length number q. This implies that the BEEA output (f) will be bounded by (1) [22].

$$-2q < f < 2q \quad (1)$$

Therefore the search space for the BEEA output (i.e. $4q$) is small as q is small. The success rate for this case is only determined by the probability the BEEA returns zero. In this regard, it can be proved that the BEEA in Algorithm 1 *only* returns zero when k is divisible by q, where q is the greatest odd divisor of k. This is perhaps a natural observation by the definition of the BEEA, however, note that the BEEA (Algorithm 1) does not return zero if k and q share an even divisor because it assumes an odd modulus p. Therefore if k and q share a power of two divisor, during the first iteration it will be removed from u and v continuing the algorithm like if k and q does not share this common divisor in the first place. For **Case 1** this distinction is irrelevant. However, **Case 2** relies critically on it.

Hence, (2) defines the probability to get a non-zero output from the BEEA for an odd prime number q. Therefore, if p is replaced by a 16-bit prime number (q), the probability of obtaining a non-zero output is higher than 99 % having fewer than 2^{18} output candidates.

$$\Pr[q \nmid k] = 1 - \frac{1}{q} \tag{2}$$

Case 1 provides initial motivation for our research, with our observation that tampering p allows for an extremely restricted BEEA output space. But we note the ratio of the number of candidates to tampered p is close to one, intuitively explaining this result. We now move on to a more subtle case, where we push this ratio towards zero, presenting a technique to retain this extremely restricted BEEA output space yet with a tampered p having a similar magnitude as the untampered p.

Case 2 (non-local to BEEA fault): This case covers a less restrictive attack model as it considers that the memory tampering fault also affects other operations. This leads to a fault attack scenario where not only the inner BEEA operations are affected by the tampering but other modular operations as well. We refer to this type of memory tampering-based fault as *non-local* to the BEEA.

In this case, assume p is tampered with a small number as in **Case 1**, and this tampering also affects reduction operations. Then: k, the BEEA output, and other secret-related variables could be also reduced modulo a small value, reducing the information that an adversary could get. This situation fits very well the ECDSA signature generation procedure (Algorithm 4 analyzed in Sect. 5.2). Note that during the computation of S a modular reduction is performed modulo a small number. Thus, the information leak of the private key would be small in comparison to what can be achieved.

From this analysis, we extract two requirements: (i) the tampering value must ensure (with some probability) that the BEEA gives a non-zero output as in **Case 1**; and (ii) the tampering value should facilitate the recovery of a large amount of secret bits.

To fulfill these two requirements, we propose tampering p with $\tilde{p}$ as in (3), where q is a low bit-length prime number as in **Case 1**.

$$\tilde{p} = q2^{\log_2 p - \log_2 q} \tag{3}$$

When tampering the BEEA modulus using (3), the algorithm's output (f) is not uniformly distributed over $[1, \tilde{p})$; indeed, it has an extreme bias. Table 1 shows an example inputs-output pair for this scenario, showing that f has the form $f = a2^j - 2^i + b$ for a large value of bit index j, small value of bit index i, and small a and b.

The rationale behind (3) follows that, as $\tilde{p}$ will have a long tail of bits equal to zero, during the first iteration of the BEEA, v (initialized to $\tilde{p}$) reduces to q due to the execution of the *v-loop*. Creating, in the very first iteration, a significant bit-length

Table 1 BEEA inputs and output example when tampering p using (3).

k	`0xcf7b84b982ad83face0e0b69864a09d5461c46a2d45a4e4f87cc6d48751e8268`
$\tilde{p} = 65521 \cdot 2^{240}$	`0xfff1000000000000000000000000000000000000000000000000000000000000`
BEEA output (f)	`0x40f3b17f7fffffffffffffffffffffffffffffffffffffffffffffffffffbff3`

difference between variables u and v. Therefore, during several iterations the condition at the *sub-step* ($u \geq v$) will evaluate as true.

Figure 2 illustrates this concept. The figure shows the bit-length progression of variables u and v just before the execution of the *sub-step* during an execution of the BEEA with a random 256-bit k and a fixed $\tilde{p}$ that follows (3) using a 16-bit q. As seen, the bit-length of v (initialized to $\tilde{p}$) starts at 256 and it abruptly drops to 16 (i.e. bit-length of q) at the very first iteration. Then it remains the same for several iterations where $u \geq v$ will hold. Due to the importance of the iteration index where $v > u$ in the forthcoming analysis, we denote this iteration by t.

Fig. 2 Bit-lengths of u and v during the execution of the BEEA for a 256-bit k and $\tilde{p} = 65521 \cdot 2^{240}$

The attack objective is to recover the BEEA output f and subsequently use it to disclose a secret of a cryptographic scheme that employs the BEEA (e.g. ECDSA in Sect. 5). As seen in Algorithm 1, the algorithm's output is equal to the last value of variable C. Also, during the algorithm execution C only depends on A and itself.

As previously discussed, in the first iteration the *u-loop* and the *v-loop* execute. However, during the execution of the *v-loop*, C holds its initial value (zero). This variable keeps its value until iteration t. Hence, during the *sub-step* at this iteration, C gets updated. On the other hand, A is processed at the *u-loop* and at the corresponding *sub-step* branch during iterations $1 \leq i \leq t$. Thus, C being equal to zero during these iterations, Algorithm 2 recovers the value of A at t-iteration just before the *sub-step* execution.

However, the adversary does not know the Z_i, as it requires the exploitation of side-channel leakage (e.g. [3,17]). However, to apply Algorithm 2 the adversary does not need to know each individual Z_i but it *only* requires the sum of all Z_i until iteration t (N). We observe it is possible for an adversary to get a very good approximation for this value.

Algorithm 2 Reconstruction algorithm for variable A until iteration t

Input: $N = \sum_{i=1}^{t} Z_i$, modulus $\tilde{p}$
Output: value of A at iteration t (just before the *sub-step* execution)

```
1: A = 1
2: for j = 1 to N = Σ_{i=1}^{t} Z_i do
3:   if even(A) then
4:     A = A / 2
5:   else
6:     A = (A + p̃)/2
7: return A
```

Following Fig. 2, u (initialized to a full-length k) must lose about $\log_2 k - \log_2 q$ bits to result in a value less than v (initialized to $\tilde{p}$ but reduced to q at first iteration) to make this condition true. Therefore the adversary knows that (4) will hold.

$$N = \sum_{i=1}^{t} Z_i \geq \lfloor \log_2 k \rfloor - \lfloor \log_2 q \rfloor , \tag{4}$$

with the r.h.s. approximated by $\lfloor \log_2 p \rfloor - \lfloor \log_2 q \rfloor$ since $\lfloor \log_2 p \rfloor$ is an estimate for the bit-length of k with estimation error ε and probability defined by (5). Therefore (4) transforms to (6),

$$\Pr\left[\lfloor \log_2 p \rfloor - \lfloor \log_2 k \rfloor < \varepsilon\right] = \sum_{i=0}^{\varepsilon-1} \frac{1}{2^{i+1}} \tag{5}$$

$$\lfloor \log_2 p \rfloor - (\lfloor \log_2 q \rfloor + \varepsilon) \leq N = \sum_{i=1}^{t} Z_i \leq \lfloor \log_2 p \rfloor - 1 \tag{6}$$

where ε represents the bit-length estimation error, and other casual bit-length reductions possibly occurring during the *sub-step* executions. We note the impact of the latter on (6) is negligible.

From (6) and (5) the real value of N is bounded by (7).

$$\begin{aligned} Nmin < N < Nmax \\ Nmin = \lfloor \log_2 p \rfloor - \lfloor \log_2 q \rfloor - \varepsilon \\ Nmax = \min(\lfloor \log_2 p \rfloor - \lfloor \log_2 q \rfloor + \varepsilon\ , \lfloor \log_2 p \rfloor - 1) \end{aligned} \tag{7}$$

At this point the adversary knows two bounds for N with probability that the correct one lies within this range (7) defined by (5). Therefore by selecting ε, the adversary will have $2\varepsilon + 1$ candidates for N that lead to the same amount of candidates for A using Algorithm 2. This method is very efficient because the success probability increases exponentially on ε while the number of candidates only grows linear on this magnitude.

Our simulation results indicate the value of N at iteration t follows a normal distribution. Therefore the set of candidates A generated by Algorithm 2 for $[Nmin, Nmax]$ has more probable elements grouped at the center of the set.

After this analysis, the adversary has the following information about BEEA variables at the start of iteration $t + 1$:

- $2\varepsilon + 1$ candidates for A
- C is equal to $-A$ (so it can be computed with candidates for A)

- u is odd and satisfies $u < q$
- $v = q - u$ (so it can be computed with candidates for u)

Therefore the attacker has bounds for all variables involved in the BEEA. Thus the attacker needs only to find a pair of (A, u) that yields the correct value of f, leaving the total number of possible candidates defined by (8).

$$\#candidates = (2\varepsilon + 1)\frac{q-2}{2} \tag{8}$$

For this last phase, assume the adversary has an oracle ($\mathcal{O}$-`BEEA`) that says if a given f is the BEEA output for some secret k and $\tilde{p}$. Then, the adversary can execute a *resumed* version of the BEEA (i.e. a BEEA that start at $t+1$ with user specified u, v, A, and C) and check if the returned value f is the correct one using the oracle $\mathcal{O}$-`BEEA`. Algorithm 3 outlines this procedure.

Algorithm 3 Final bruteforcing for recovering BEEA's output.

Input: ε, $\tilde{p}$, $\mathcal{O}$-`BEEA`(f) oracle for some unknown secret k
Output: Correct output ($f \leftarrow$ BEEA$(k, \tilde{p})$)

```
1:  A = Generate all A candidates for [Nmin, Nmax] using Alg. 2 and ε
2:  Sorted_A ← Sort A by most probable element first (considering that N distributes normal)
3:  for all A ∈ Sorted_A do
4:      C = −A                        // Step 23 at the sub-step
5:      u = q − 2                     // Start from the greater value for u at iteration t
6:      v = q
7:      while u > 0 do
8:          v = v − u                 // Step 22 at the sub-step
9:          f ← BEEA(u, v, A, C, p̃)   // Resume BEEA execution from iteration t + 1
10:         if O-BEEA(f) then
11:             return f
12:         else
13:             u = u − 2             // Next candidate for u
```

After presenting the formal description of the attack, we now discuss simulation results. We executed Algorithm 3 for 1000 randomly chosen 256-bit k with $\tilde{p} = 65521 \cdot 2^{240}$ and $\varepsilon = 30$. For each of these runs we tracked if the attack succeeded or not and the number of candidates tested if successful. The success rate was higher than 99%, as expected since the success probability is very close to unity for $\varepsilon = 30$ and a 16-bit q. Only one run failed due to a zero BEEA output and the rest succeeded. Regarding the number of candidates, during the simulations we executed Algorithm 3 with and without the sorting in Step 2, using the same inputs. Figure 3 contains the histograms for both experiments, with expected results since N follows a normal distribution; the sorting step considerably reduces the number of tested candidates.

Summarizing this attack, the success rate is given by the joint probability between (i) the BEEA outputting a non-zero value (Eq. (2)); and (ii) the correct selection of ε (Eq. (5)). However, the bit-length of q mainly governs the attack complexity (Eq. (8)) since ε is small. Therefore, the selection of these values is a compromise between success probability and attack complexity. The simulation results show that $\varepsilon = 30$ and a 16-bit prime q leads to a success rate higher than 99%. One important aspect is the success rate and computational effort are independent of the bit-length of k, so it performs similar for different key sizes.

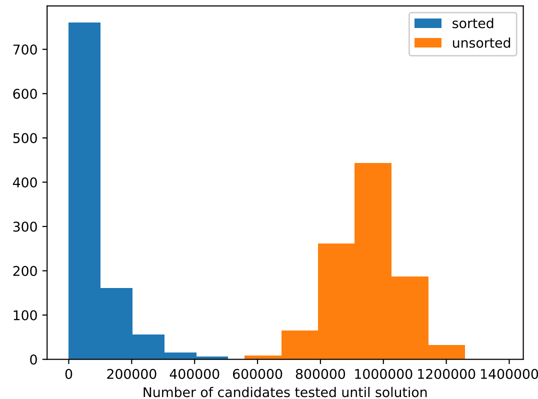

Fig. 3 Histogram of the number of candidates needed to get a solution.

5 Application example: ECDSA vs a memory tampering capable adversary

In this section the security of ECDSA signature generation procedure is analyzed considering a memory tampering capable adversary. First, this algorithm is presented for supporting its analysis in forthcoming sections then, at Sect. 5.1, it is shown how previously published attacks can be easily implemented after a memory tampering attack. Finally at Sect. 5.2 the proposed attack is applied to ECDSA.

The ECDSA algorithm is the elliptic curve variant of the digital signature algorithm standardized by NIST [28]. Algorithm 4 shows the pseudocode of the ECDSA signature generation procedure. This algorithm generates a digital signature for a public message (m) employing the secret private key (d), where h corresponds to the application of a hash function to the message m and is also considered public.

Algorithm 4 ECDSA signature generation

Inputs: private key (d), elliptic curve generator (G), hash of m (h), order of G (p)
Output: a signature for message m: (r, S)

1: Select k at random such that $0 < k < p$
2: $(x, y) = k \cdot G$
3: $r = x \mod p$
4: $S = k^{-1}(h + rd) \mod p$
5: **if** $r = 0$ **or** $S = 0$ **then**
6: **goto** 1
7: **else**
8: **return** (r, S)

Each generated signature involves selecting a random secret nonce k satisfying $0 < k < p$, performing scalar multiplication of this nonce with the elliptic curve generator point (G), and reducing the resulting value (x) modulo p [28].

At Step 4, the linear part of the signature generation computes the modular inverse of k and uses it to calculate the public value S. This step is the most relevant for this work because the presented attack analyzes the impact of tampering the value of p to recover the BEEA output during the modular inversion of the secret nonce k.

In the context of ECDSA, the BEEA inputs are two integer numbers k and p, such that $0 < k < p$ where p is a known prime number of hundreds of bits (at least 256 bits for modern security standards). In addition to the straightforward recovery of the private key from the knowledge of k using Step 4, every bit of this nonce must be kept secret to avoid recovery of the private key by solving an instance of the Hidden Number Problem [29].

Regarding this work, it is worth noting that ECDSA employs two finite fields. The first one is the finite field over which the elliptic curve is defined where its arithmetic operations are only used during the scalar multiplication at Step 2. The computation of S uses a prime finite field defined by p. This distinction is important because the presented attack is based on inserting faults by tampering the value of p, meaning that this modification should only affect arithmetic operations defined over this field, making this attack applicable to all ECDSA-allowed elliptic curves and independent of the employed scalar multiplication algorithm.

5.1 Fault attacks on ECDSA: memory tampering in the realm of previous works

This section shows how two previously proposed fault attacks against ECC are implementable by a tampering capable adversary. Although not an exhaustive list, we consider that it effectively demonstrates that memory tampering based fault attacks should be considered in the threat model of some implementations, especially on FPGA based ones.

In 2000, Biehl et al. presented several fault attacks against some elliptic curve cryptosystems [8]. That work introduced a branch of fault attacks against ECC, known as weak curve attacks, which aims at injecting a fault to force the computation of the scalar multiplication on a weak curve where the ECDLP can be efficiently solved. Regarding ECDSA in [8], the authors presented a fault attack based on inserting a fault on the generator point G. This attack fault model considers that an adversary can change one bit of G, being it easily implemented for a memory tampering-capable adversary as G is fixed and likely to be hardcoded in the implementation.

Another very interesting previous result related to this manuscript appears in [16]. The authors presented a combined fault and side-channel analysis attack against elliptic curve cryptosystems. Regarding ECDSA the authors noted that their attack does not apply because this primitive uses a fixed point (G). However, in the context of a memory tampering attack it is possible for an adversary to modify the generator point (G) to apply the attack proposed in [16], adding ECDSA to the list of schemes threatened by the combined attack proposed in [16].

The attack presented in [16] has the advantage over [8] that the former is effective even in the presence of some validity check countermeasures [8]. On the other hand, the latter offers a pure algebraic key recovery without needing any side-channel information. Despite these differences, the most important issue is that memory tampering techniques available against FPGA platforms can be used to implement some previous fault models: creating new attack scenarios that were initially discarded, as ECDSA in [16].

5.2 Impact of proposed memory tampering attack on the BEEA for ECDSA

In this section we analyze the impact of the proposed memory tampering-based fault attack against BEEA implementations in the context of ECDSA. In this scenario there are two main factors to consider. First, variables that affect the success probability and

second, the definition of a BEEA oracle ($\mathcal{O}$-`BEEA`). The most significant factor in the success probability of BEEA output recovery Algorithm 3 is q, as it controls the probability that the BEEA does not return zero. However, in the context of ECDSA this is irrelevant since in the case that the BEEA returns zero during the computation of S, the algorithm itself will loop to generate another k until obtaining a non-zero S and r (see Algorithm 4). Therefore the attacker only needs to tamper the value of p, wait for a signature pair (r, S), and then perform the attack with the success probability determined only by ε.

Regarding the definition of an oracle $\mathcal{O}$-`BEEA`, solving for the private key d in the equation to compute S in Algorithm 4 yields (9) where the term f represents the faulty BEEA output returned by this algorithm when its inputs are k and the tampered value $\tilde{p}$. Therefore, (9) enumerates private key candidates from f candidates, checking which of them yields a solution to the ECDLP: $P = d \cdot G$, where P is the public key.

$$d = (Sf^{-1} - h)r^{-1} \mod \tilde{p} \tag{9}$$

From (9), if $\tilde{p}$ is coprime with f and r, the private key can be directly recovered. On the other hand, if $\tilde{p}$ is not coprime with f or with r, straightforward recovery of d using (9) is not possible. However, (9) is solvable even when the required inversions do not exist.

Rearranging (9) by combining inversions yields (10) where $n = \lfloor \log_2 p \rfloor - \lfloor \log_2 q \rfloor$. Since q and 2^n are coprime, a divide-and-conquer approach based on the CRT solves (10).

$$d = (S - hf)(rf)^{-1} \mod q2^n \tag{10}$$

Applying the CRT to (10) transforms it to (11), where the problem of solving a linear congruence with a nonexistent inverse reduces to solving two linear congruences (12) and (13) with a modulus that admits practical recovery of d.

$$d \equiv a \cdot q(q^{-1} \bmod 2^n) + b \cdot 2^n(2^{-n} \bmod q) \mod q2^n \tag{11}$$

$$a \equiv (S - hf)(rf)^{-1} \mod 2^n \tag{12}$$

$$b \equiv (S - hf)(rf)^{-1} \mod q \tag{13}$$

In the case that $\gcd(rf, q2^n) = 2^l$, (14) leads to recovery of the least significant $n - l$ bits of a value of a that satisfies (12). Therefore, an exhaustive search for the missing l most significant bits of a recovers d using (11).

$$a \equiv \left(\frac{S - hf}{2^l}\right)\left(\frac{rf}{2^l}\right)^{-1} \mod 2^{n-l} \tag{14}$$

In the case that $\gcd(rf, q2^n) = q$, one obtains a similar equation to (14) using (13). With small q (e.g. a 16-bit number), an exhaustive search locates the value of $b \bmod q$ satisfying (13). Finally, if $\gcd(rf, q2^n) = q2^l$, the previous cases combine to yield a solution. Considering the product rf random, $\gcd(rf, q2^n)$ should be small with good probability, therefore these exhaustive searches are feasible.

Using this method for solving (10), every candidate for f becomes a candidate for d, therefore no solutions are lost in the process. The last issue is the fact that the correct private key could be greater than $\tilde{p}$. ECDSA private keys are draw at random in $\mathbb{Z}_p^*$, therefore as $\tilde{p}$ is built such that it has the same bit-length as p, the correct private key

can only take two values: d as returned by (10) or $d+\tilde{p}$. In this regard it is worth noting that the adversary can decrease the probability that the latter is the solution by choosing q with a long chain of ones in its most significant bits (for example, $q = 65521 = \texttt{0xfff1}$), as the private key should have a chain of ones of at least the same size as that of q to be greater than $\tilde{p}$.

Summarizing the impact of the presented attack on ECDSA, the success rate is only determined by ε, hence according to (5) it can be selected to be very close to unity with a minimal impact on the computational cost as analyzed in Sect. 4.1. The computational cost of the entire attack on ECDSA is expressed in terms of number of scalar multiplications to get the correct private key. This cost splits into two parts: (i) the cost of recovering the BEEA output and (ii) the cost of solving private key recovery equation (10). Following the respective analysis, (15) gives the worst case number of scalar multiplications needed, where the nested sums represent the generation of each candidate f using Algorithm 3.

$$\#\ \textit{max scalar multiplications} = 2 \sum_{Amin}^{Amax} \sum_{u=1}^{q-2} \gcd(rf, q2^n) \tag{15}$$

We simulated the complete attack for ECDSA 1000 times for two values of q and $\varepsilon = 30$. In both cases the success rate was 100 %, the difference between q relies on the number of scalar multiplications needed to find the solution. The 1000 runs for $q = 3$ took about 10 minutes to complete while for $q = 65521$ it took about 10 hours[1]. This difference is due to the $q = 3$ inner loop being removed from Algorithm 3 (i.e. second sum in (15)) while for a 16-bit q it requires more effort. Additionally as f does not distribute uniformly, our data suggest there exists a slight bias on this value to be divisible by q, therefore for a 16-bit q this leads to more candidates to test while solving (10). We note the attack complexity bound (8) is not optimal. However, as simulation running times show, the presented attack is affordable in practice, especially for $q = 3$. These results suggest the proposed memory tampering-based fault attack has a significant impact on the security of ECDSA: one signature pair is sufficient to recover the private key in practical time independently of the key size.

During these simulations it was assumed that the adversary can tamper the value of the modulus p using, for example, the procedure detailed in Appendix A. Therefore, we exclude this part from the bulk simulations described above and tested the tampering procedure independently (see Sect. 2). This is possible because said tampering procedure is deterministic and free of errors, therefore, it does not have any influence on the attack success rate nor its running time. Furthermore, as described in this section the presented attack is completely algebraic and independent of the memory tampering procedure, therefore it can be applied to other scenarios where it is possible to tamper the value of the modulus used by an implementation of the BEEA.

6 Countermeasures

The presented attack critically relies on cryptographic algorithm parameter tampering. Therefore, ensuring the authenticity of these parameters should be sufficient for protecting them. In this regard, public key-based authentication is preferred for avoiding

[1] Runs were serially executed on an Intel i7-3770 3.4 GHz using a Python implementation of the attack.

side-channel attacks like those presented in [25, 26]. However one-time verification (i.e. at startup) might be insufficient as this attack could be triggered *on-the-fly* (see Sect. 2.1), requiring the use of runtime countermeasures, like, for example, verifying ECDSA-generated signatures before outputting them.

On the other hand, verify-before-output approach degrades performance. At the same time, using Fermat's Little Theorem (FLT) for computing modular inverses should offer better protection against the proposed attack in both settings. However, applying **Case 1** model to the FLT (i.e. tampered modulus is only used by the FLT), also leads to a vulnerable scenario. Let us considering p is tampered with an small value, then the FLT output will be small. This increases the probability of getting two signatures generated with the same FLT output, opening the door to a nonce-reuse attack against ECDSA.

In summary, FLT selection is preferred upon previous, specially when the modulus is *shared* among all modular arithmetic operations.

7 Conclusions

Fault, memory tampering, and bitstream manipulation implementation attacks are very powerful techniques for compromising the security of a cryptographic algorithm. In this work, we presented a memory tampering-based fault attack on an implementation of binary GCD based modular inversion algorithms. The targeted implementation is the Binary Extended Euclidean Algorithm for computing modular inverses, but it extends to others based on it. We analyzed two tampering values, showing how the attack can be adapted to different scenarios, in both cases employing only one BEEA execution with very high probability and almost negligible computational effort.

We applied the proposed attack to ECDSA, achieving a success rate of 100 % supported by formal analysis and experimental validation. This attack is not elliptic curve dependent, thus it applies to any ECDSA-allowed elliptic curve.

Our results demonstrate the memory tampering technique is a very powerful attack vector. Furthermore, we also showed how our proposal allows applying the attack presented in [16] in the ECDSA setting, a cryptographic scheme initially discarded in that work.

Future work includes applying our technique to construct malicious software binaries and/or certificate-based cryptosystem parameters, with an analysis on the resulting real-world software behavior in this context.

Acknowledgments

This work was partially funded by Academy of Finland (Grant No. 303814) and Spanish Government (with support from FEDER) (Project No. TEC2017-83557-R).

References

1. Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 185–203. Springer Berlin Heidelberg, 2007.
2. Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. AES T-Box tampering attack. *Journal of Cryptographic Engineering*, 6(1):31–48, 2016.

3. Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA Vulnerabilities of the Binary Extended Euclidean Algorithm. *Journal of Cryptographic Engineering*, 7(4):273–285, 2017.
4. Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *International Journal of Circuit Theory and Applications*, 45(2):199–213, 2017.
5. Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 105–114. IEEE, 2011.
6. 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.
7. Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, Xuan Thuy Ngo, and Laurent Sauvage. Hardware trojan horses in cryptographic ip cores. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 15–29. IEEE, 2013.
8. Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In *Annual International Cryptology Conference (CRYPTO)*, pages 131–146. Springer, 2000.
9. Joppe W Bos. Constant time modular inversion. *Journal of Cryptographic Engineering*, 4(4):275–281, 2014.
10. Arnaud Boscher, Helena Handschuh, and Elena Trichina. Blinded fault resistant exponentiation revisited. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC),*, pages 3–9. IEEE, 2009.
11. Mathieu Ciet and Marc Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Designs, codes and cryptography*, 36(1):33–43, 2005.
12. Jean-Luc Danger, Sylvain Guilley, Philippe Hoogvorst, Cédric Murdica, and David Naccache. A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *Journal of Cryptographic Engineering*, 3(4):241–265, 2013.
13. Guerric Meurice De Dormale, Philippe Bulens, and J-J Quisquater. An improved montgomery modular inversion targeted for efficient implementation on fpga. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 441–444. IEEE, 2004.
14. Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version. *Journal of Cryptographic Engineering*, 4(1):33–45, 2014.
15. F. A. Escobar, X. Chang, and C. Valderrama. Suitability analysis of fpgas for heterogeneous platforms in hpc. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):600–612, Feb 2016.
16. Junfeng Fan, Benedikt Gierlichs, and Frederik Vercauteren. To infinity and beyond: Combined attack on ECC using points of low order. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 143–159. Springer, 2011.
17. Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 83–98, Vancouver, BC, 2017. USENIX Association.
18. Tim Güneysu. Utilizing hard cores of modern FPGA devices for high-performance cryptography. *Journal of Cryptographic Engineering*, 1(1):37–55, 2011.
19. Anju P Johnson, Sayandeep Saha, Rajat Subhra Chakraborty, Debdeep Mukhopadhyay, and Sezer Gören. Fault attack on aes via hardware trojan insertion by dynamic partial reconfiguration of fpga over ethernet. In *Proceedings of the 9th Workshop on Embedded Systems Security*, page 1. ACM, 2014.
20. Jr. Kaliski B.S. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.
21. Ágnes Kiss, Juliane Krämer, Pablo Rauzy, and Jean-Pierre Seifert. Algorithmic countermeasures against fault attacks and power analysis for RSA-CRT. In *International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, pages 111–129. Springer, 2016.
22. Donald E Knuth. *Seminumerical algorithms, Volume 2 of The Art of Computer Programming*. Addison-Wesley, 1981.
23. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, pages 388–397. Springer Berlin, 1999.
24. Juliane Krämer. *Why Cryptography Should Not Rely on Physical Attack Complexity*. Springer Publishing Company, Incorporated, 1st edition, 2015.
25. Amir Moradi, Markus Kasper, and Christof Paar. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures. In *Topics in Cryptology–CT-RSA 2012*, pages 1–18, San Francisco, USA, 2012. Springer.

26. Amir Moradi and Tobias Schneider. Improved side-channel analysis attacks on xilinx bitstream encryption of 5, 6, and 7 series. In François-Xavier Standaert and Elisabeth Oswald, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 71–87, Cham, 2016. Springer International Publishing.
27. Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.
28. National Institute of Standards and Technlogy (NIST). Digital Signature Standard (DSS). FIPS 186-4, 2013.
29. Phong Q Nguyen and Igor E Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Designs, Codes and Cryptography*, 30(2):201–217, 2003.
30. Thomas Popp. An introduction to implementation attacks and countermeasures. In *Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, pages 108–115. IEEE Press, 2009.
31. Debapriya Basu Roy, Shivam Bhasin, Sylvain Guilley, Jean-Luc Danger, Debdeep Mukhopadhyay, Xuan Thuy Ngo, and Zakaria Najm. Reconfigurable lut: a double edged sword for security-critical applications. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 248–268. Springer, 2015.
32. Erkay Savaş and Çetin Kaya Koç. Montgomery inversion. *Journal of Cryptographic Engineering*, 8(3), 2018.
33. Patrick R Schaumont. *A practical introduction to hardware/software codesign.* Springer Science & Business Media, 2012.
34. Shaunak Shah, Rajesh Velegalati, Jens-peter J-P Kaps, and David Hwang. Investigation of DPA Resistance of Block RAMs in Cryptographic Implementations on FPGAs. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 274–279. IEEE, 2010.
35. Josef Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.
36. Pawel Swierczynski, Georg T Becker, Amir Moradi, and Christof Paar. Bitstream Fault Injections (BiFI)–Automated Fault Attacks against SRAM-based FPGAs. *IEEE Transactions on Computers*, 67(3), 2017.
37. Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. Interdiction in practice—Hardware Trojan against a high-security USB flash drive. *Journal of Cryptographic Engineering*, 7(3):199–211, 2017.
38. Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, and Christof Paar. FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(8):1236–1249, 2015.
39. Elena Trichina and Antonio Bellezza. Implementation of elliptic curve cryptography with built-in counter measures against side channel attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 98–113. Springer, 2002.
40. Xilinx Inc. Data2MEM User Guide, 2010.
41. Xilinx Inc. Vivado Design Suite User Guide: Embedded Processor Hardware Design, 2017.

A Xilinx FPGAs embedded block memory tampering procedure

This section describes a procedure for extracting and modifying Xilinx FPGA embedded block memory content from bitstream files. This procedure was partially described in [2] and is based on Xilinx `data2mem` tool public documentation [40]. The described flow applies for Spartan-6 and below FPGA families, for 7 Series family, the process is very similar following [41]. During this section it is assumed that the adversary obtained the bitstream for a given FPGA and it does not contain any protection.

SRAM-based FPGAs at every power-on cycle are configured with a bitstream file. It is often called a "file" as it is produced at development stage using software tools, however, the content of this file is, usually, stored in non-volatile memory external to the FPGA for ensuring the power-on cycle. This file contains the synthesized hardware encoded in a non-public format. While almost every change in the bitstream requires hardware design re-synthesis, changing embedded flow memory content does not.

This modification is possible using the Xilinx tool `data2mem` which was specifically developed for this use case [40]. In addition to this usage, this tool has a *debug* option (i.e. `-d` flag) that allows extracting the content of any embedded block memory (BRAM) from a bitstream file. Therefore, `data2mem` could be seen as a BRAM content encoder/decoder from the non-public bitstream format. Thus, employing this tool it is possible to have read/write access to the content of any BRAM used in the design.

The general procedure for tampering embedded block memory content follows:

1. Read the content of all BRAMs in a given bitstream.
2. Identify which BRAM stores the value of interest and its address.
3. Tamper said BRAM with the malicious value.

These steps are deterministic, free of errors and their running times are negligible. The next two sections describe them in detail.

A.1 Reading BRAMs content

For reading BRAM content from a bitstream file, `data2mem` requires the bitstream file itself and a BMM (i.e. BRAM Memory Map) file that indicates which BRAM to be read[2]. An adversary at first does not known at which BRAM the targeted value is stored nor its address. However, using public documentation of the FPGA part number it is possible to create a BMM file that maps every BRAM in that FPGA. Then using the `data2mem` tool with the `-d` flag an output file is obtained with the content of every BRAM in the bitstream [40].

Therefore, using this simple and deterministic procedure the adversary can read all BRAM content and identify which one contains the value of interest. In addition, the adversary also extracts at which address of said BRAM the value is stored.

Regarding the application to the BEEA explained in Sect. 4, this targeted value is the input modulus that is usually known for an adversary (see ECDSA example in Sect. 5). Therefore, its identification is immediate.

A.2 Tampering BRAMs content

Having identified which BRAM and address will be tampered, the adversary can use the tool `data2mem` for modifying a given BRAM address. In this use case, this tool needs (i) the targeted bitstream (ii) a BMM file indicating which BRAM to modify (iii) a file with the new content [40]. After executing this tool with these input files, a new bitstream file is obtained that is compliant with Xilinx format. Therefore, it can be used to configure an FPGA and observe its behavior with the tampered value.

[2] Detailed format description of this file can be found in [40].