Synthesis of Fault Attacks on Cryptographic Implementations

Gilles Barthe[1], François Dupressoir[1], Pierre-Alain Fouque[2], Benjamin Grégoire[4], and Jean-Christophe Zapalowicz[4]

[1] IMDEA Software Institute
gilles.barthe@imdea.org,francois.dupressoir@imdea.org
[2] Université de Rennes 1 and Institut universitaire de France
pierre-alain.fouque@ens.fr
[3] INRIA
benjamin.gregoire@inria.fr,jean-christophe.zapalowicz@inria.fr

Abstract. Fault attacks are active attacks in which an adversary with physical access to a cryptographic device, for instance a smartcard, tampers with the execution of an algorithm to retrieve secret material. Since the seminal Bellcore attack on RSA signatures, there has been extensive work to discover new fault attacks against cryptographic schemes, and to develop countermeasures against such attacks. Originally focused on high-level algorithmic descriptions, these works increasingly focus on concrete implementations. While lowering the abstraction level leads to new fault attacks, it also makes their discovery significantly more challenging. In order to face this trend, it is therefore desirable to develop principled, tool-supported approaches that allow a systematic analysis of the security of cryptographic implementations against fault attacks.

We propose, implement, and evaluate a new approach for finding fault attacks against cryptographic implementations. Our approach is based on identifying implementation-independent mathematical properties we call *fault conditions*. We choose them so that it is possible to recover secret data purely by computing on sufficiently many data points that satisfy a fault condition. Fault conditions capture the essence of a large number of attacks from the literature, including lattice-based attacks on RSA. Moreover, they provide a basis for discovering automatically new attacks: using fault conditions, we specify the problem of finding faulted implementations as a program synthesis problem. Using a specialized form of program synthesis, we discover multiple faulted implementations on RSA and ECDSA that realize the fault conditions, and hence lead to fault attacks. Several of the attacks found by our tool are new, and of independent interest.

1 Introduction

Embedded devices often play a central role in security architectures for large-scale software infrastructures. For instance, they are used pervasively for purposes such as authentication, identity management, and digital signatures. As a consequence, embedded devices are also a prime target for attackers. There are primarily two means to retrieve secret material from embedded devices. The first one is to carry non-invasive monitoring of the device and to obtain information from side-channels, such as timing and power consumption, electromagnetic radiations, or even noise. The second one is to perform active attacks, injecting faults that interfere with the normal execution of the devices, and to recover the secret information through the devices normal interface, or through side-channels. The effects of these faults vary: they may modify the control flow of the program by skipping a conditional test [2] or induce behaviours similar to buffer overflows [19]. In the context of cryptographic attacks, they often allow the adversary to directly recover secret keys. There are multiple ways to inject faults in devices; examples include power spikes, glitches on the clock signal, temperature variations, and electromagnetic radiations [2,25,4].

The existence of efficient fault attacks against cryptographic schemes was first demonstrated in [12] by Boneh, DeMillo and Lipton. They consider an algorithm, shown in Figure 3, for computing RSA signatures using the Chinese Remainder Theorem (CRT) and its standard recombination formula:

$$S = (S_q \cdot p^{-1} \bmod q) \cdot p + (S_p \cdot q^{-1} \bmod p) \cdot q \bmod N$$

where S_p and S_q are modular exponentiations of the reductions modulo p and q of the integer M that encodes the message m. The algorithm is popular in practice, because it achieves a significant speedup (approximately ×4) over the more elementary computation $S = M^d \bmod N$. Boneh et al. exhibit an efficient fault attack for retrieving the factorization of N, i.e. p and q, with a simple gcd computation. This attack requires knowledge of a valid signature S and a faulted signature $\widehat{S}$ for the same message M. A second attack, due to Lenstra, only requires knowledge of a single faulted signature $\widehat{S}$ for a known message M:

- in the first case, one can recover the factorization of N from the identity $\gcd(S - \widehat{S}, N) = q$;
- in the second case, one can recover the factorization of N from the identity $\gcd(M - \widehat{S}^e, N) = q$.

Both attacks, often known as the Bellcore attacks, are restricted to deterministic encodings. However, Boneh et al. describe another attack that applies to probabilistic encodings; unfortunately, this third attack is not as efficient as the others. In fact, it is only very recently that Fouque, Guillermin, Leresteux, Tibouchi and Zapalowicz [18] propose the first efficient fault attacks against RSA-CRT signatures with probabibilistic encodings; their attacks are applicable against PKCS#1 v2.0 (PSS) signatures. In addition to the RSA-CRT signature considered in [12], Fouque et al. consider another variant of RSA-CRT (Figure 4) based on Garner's recombination formula:

$$S = S_q + q \cdot (q^{-1} \cdot (S_p - S_q) \bmod p)$$

However, the main difference between [12] and [18] lies in their level of description of RSA-CRT: whereas Boneh et al. [12] consider a high-level algorithmic description in which modular exponentiation is treated abstractly, Fouque et al. consider reasonably detailed implementations, going down to algorithmic descriptions of modular multiplication. They consider four different implementations of modular exponentiation, among which the Square-and-Multiply algorithm and Montgomery's modular multiplication algorithm [31]; moreover, they consider for each of these implementations two variants based on the Coarsely Integrated Operand Scanning (CIOS) technique or the Separate Operand Scanning (SOS) techniques. The code for Montgomery modular multiplication based on CIOS is shown in Figure 1. Fouque et al. show that, by injecting faults in the implementations of Montgomery modular multiplication, one can obtain faulted signatures $\widehat{S}$ that are close multiples of p or q, and then use lattice-based techniques to recover the factorization of N with about 50 faulty signatures. This example provides strong evidence that analyzing implementations rather than algorithmic descriptions can lead to the discovery of interesting attacks. However, it also highlights a number of difficulties with this approach:

1. the number of faulted implementations grows exponentially in the length of the original program, in particular if multiple faults are considered;
2. some fault attacks require to tamper with some (but not all) loop iterations, or to add or remove some loop iterations; hence, the number of faulted implementations cannot even be bounded solely based on the length of the original program;
3. analyzing the effects of faults becomes very involved and error-prone, in particular for programs with loops;
4. there exist multiple implementations of basic arithmetic computations, requiring to repeat the analysis for each of them;
5. there might exist numerous countermeasures against a fault attack, requiring to repeat the analysis for each of the amended implementations.

This current trend towards analyzing security against fault attacks of implementations rather than high-level algorithmic descriptions is not specific to RSA signatures. In fact, it is also witnessed in elliptic curve cryptography. Biehl, Meyer and Müller [11] were among the first to consider fault attacks against elliptic curve cryptosystems; more specifically, they consider an elliptic curve variant of ElGamal encryption. Their attacks exploit some of the ideas from Boneh et al. and are cast in the setting of a high-level algorithmic description of scalar multiplication between a field element and a point in the curve. Later, Naccache, Nguyen, Tunstall and Whelan [35] exhibit fault attacks on implementations of DSA and its elliptic curve variant ECDSA, whose description is given in Figure 6. Their attack introduces a fault during the generation of the nonce k and is cast in an algorithmic setting. In contrast, more recent works [3,8,30] study fault attacks against implementations of ECDSA, based on detailed accounts of integer multiplication, scalar multiplication, and point doubling. For example, the attack on integer multiplication [3] by Barenghi et al. works by injecting faults during the integer multiplication of a known random value and the secret key. Then, by considering the textbook multiplication implementation, they show that it is possible to recover the secret key. Finally, the attack of [30] shows that it is possible to inject a fault during the conversion from projective to affine coordinates. These two attacks show that it is beneficial to consider all steps of an implementation-level description when looking for fault attacks. Our goal in this paper is to search for fault attacks by considering full implementation-level descriptions of cryptographic algorithms.

```
function CIOS($x, y$)
    $a \leftarrow 0$
    $y_0 \leftarrow y \bmod b$
    for $j = 0$ to $k - 1$ do
        $a_0 \leftarrow a \bmod b$
        $u_j \leftarrow (a_0 + x_j \cdot y_0) \cdot q' \bmod b$
        $a \leftarrow \left\lfloor \frac{a + x_j \cdot y + u_j \cdot q}{b} \right\rfloor$
    end for
    if $a \geq q$ then $a \leftarrow a - q$
    end if
    return $a$
end function
```

Fig. 1. The Montgomery multiplication algorithm. The x_i's and y_i's are the digits of x and y in base b; $q' = -q^{-1} \bmod b$ is precomputed. The returned value is $(xy \cdot b^{-k} \bmod q)$. Since $b = 2^r$, the division is a bit shift.

Thesis and contributions

The thesis of this work is that it is beneficial to develop and implement rigorous methodologies for discovering fault attacks on cryptographic implementations. To support our thesis, we propose and validate experimentally a principled, tool supported approach for discovering fault attacks on cryptographic implementations. Our approach relies on two broad contributions:

1. identifying *fault conditions*, a novel concept that captures the essence of fault attacks in a logical, implementation independent setting;
2. applying a form of program synthesis on concrete cryptographic implementations to automatically discover faulted implementations that realize fault conditions and lead to attacks.

A third, more practical contribution is an evaluation of our approach on implementations of RSA and ECDSA signatures. During the process, we discover several faulted implementations, some of which lead to new attacks of independent interest. We elaborate on these points below.

Fault conditions The first contribution (Section 3) is of methodological nature, and rests on the introduction of *fault conditions*. Informally, fault conditions are implementation-independent mathematical properties, specific to a cryptographic system, which capture sufficient conditions under which an attacker can launch a successful attack. Consider, for instance, the case of RSA signatures with public

```
function $\textsc{Exp}_{\textsc{Ladder}}(x, e, q, c)$
    $\bar{x} \leftarrow \mathrm{CIOS}(x, R^2 \bmod q)$
    $A \leftarrow R \bmod q$
    for $i = t$ down to $0$ do
        if $e_i = 0$ then
            $\bar{x} \leftarrow \mathrm{CIOS}(A, \bar{x})$
            $A \leftarrow \mathrm{CIOS}(A, A)$
        else if $e_i = 1$ then
            $A \leftarrow \mathrm{CIOS}(A, \bar{x})$
            $\bar{x} \leftarrow \mathrm{CIOS}(\bar{x}, \bar{x})$
        end if
    end for
    $A \leftarrow \mathrm{CIOS}(A, c)$
    return $A$
end function
```

Fig. 2. Montgomery's Ladder for computing modular exponentiations: $\textsc{Exp}_{\textsc{Ladder}}(x, e, q, c) = x^e \cdot c \bmod q$. $e_0, \ldots, e_t$ are the bits of the exponent e (from the least to the most significant), b is the base in which computations are carried out ($\gcd(b, q) = 1$) and $R = b^k$.

```
function $\textsc{Sign}_{\textsc{RSA–CRT}}(m)$
    $M \leftarrow \mu(m) \in \mathbb{Z}_N$                                ▷ message encoding
    $S'_p \leftarrow \textsc{Exp}_{\textsc{Ladder}}(M \bmod p, d_p, p, q^{-1} \bmod p)$
    $S'_q \leftarrow \textsc{Exp}_{\textsc{Ladder}}(M \bmod q, d_q, q, p^{-1} \bmod q)$
    $S \leftarrow S'_q \cdot p + S'_p \cdot q \bmod N$
    return $S$
end function
```

Fig. 3. RSA–CRT signature generation. p and q are large primes and $N = pq$ is the modulus. The public key is denoted by (N, e) and the associated private key by (p, q, d). The reductions d_p, d_q modulo $p - 1, q - 1$ of the private exponent, as well as the $p^{-1} \bmod q$ and $q^{-1} \bmod p$ CRT coefficients, are precomputed.

RSA modulus $N = pq$ of length n. Any adversary with knowledge of a value S that is multiple of p but not multiple of q can obtain p by performing a simple GCD computation, and then q by division. This is captured by the fault condition

$$S : S = 0 \ \mathsf{mod}\ p \land S \neq 0 \ \mathsf{mod}\ q$$

Figure 7 summarizes some relevant instances of fault conditions for RSA signatures; in Section 3, we also consider fault conditions for ECDSA signatures. For each of the fault conditions we consider, we exhibit an attack for retrieving the secret key. Broadly speaking, the attacks fall into two categories. The first one encompasses attacks that perform an elementary computation from a value satisfying the fault condition. The second one covers attacks that require many values that satisfy the fault condition, and involve more complex computations, typically based on lattice reductions. For the latter, we implement the attacks in a computer algebra system, and we experimentally validate their effectiveness for different choices of parameters.

Fault Models and Policies The literature offers a wide range of fault models, that affect both data flow (for example the null fault model in which integer variables can be set to a null value) and control flow (for example, the instruction skip fault model, where an instruction can be skipped). We consider various *fault policies*, that subsume a wide range of such fault model and provide fine-grained specifications of the faults that can be performed on implementations. They model faults using replacement clauses of the form (x, e) where x is a variable and e is an expression, or (c, c'), where c and c' are commands. These clauses respectively state that it is possible to replace x by e, and c by c' in the execution of the program.

```
function $\textsc{Sign}_{\textsc{RSA–Garner}}(m)$
    $M \leftarrow \mu(m) \in \mathbb{Z}_N$                                   ▷ message encoding
    $S_p \leftarrow \textsc{Exp}_{\textsc{Ladder}}(M \bmod p, d_p, p, 1)$
    $S_q \leftarrow \textsc{Exp}_{\textsc{Ladder}}(M \bmod q, d_q, q, 1)$
    $t \leftarrow S_p - S_q$
    if $t < 0$ then $t \leftarrow t + p$
    end if
    $S \leftarrow S_q + \big((t \cdot \pi) \bmod p\big) \cdot q$
    return $S$
end function
```

Fig. 4. RSA–Garner signature generation. The Garner coefficient $\pi = q^{-1} \bmod p$ is precomputed.

```
function $\textsc{ECScalMul}(k,P)$
    $R_0 \leftarrow \infty$
    for $i = t$ down to $0$ do
        $R_0 \leftarrow [2] \cdot R_0$
        if $k_i = 1$ then $R_0 \leftarrow R_0 + P$
        end if
    end for
    return $R_0$
end function
```

Fig. 5. Scalar Multiplication of an elliptic curve point by a field element. $[2]\cdot$ denotes point doubling operation, and $+$ denotes addition.

Automated synthesis of faulted implementations Identifying fault conditions that allow efficient attacks to exist is a manual process that requires cryptographic expertise, and some good understanding of the mathematical tools available for cryptanalysis. The significant pay-off of fault conditions is that the process of finding complying faulted implementations can be automated. Our second contribution (Section 4) is a fully automated method for discovering faulted implementations that verify the fault condition. Our method can be seen as an instance of *program synthesis*, an area that is currently undergoing rapid and significant progress (see Section 6). Broadly construed, the goal of program synthesis is to find, given a specification ϕ (for instance, ϕ might capture the input/output behavior of a program), a set of programs that satisfy ϕ. Because synthesis is computationally expensive, there exist many specialized forms of program synthesis that restrict the search space using non-functional requirements or by providing a partial description of the desired programs. We also specialize our synthesis algorithm to keep it computationally reasonable.

Specifically, our algorithm takes as input a fault condition ϕ, an implementation c, and searches for all faulted implementations of c that satisfy ϕ. The search is constrained by two additional inputs.

```
1: function $\textsc{Sign}_{\textsc{ECDSA}}(m)$
2:     $h \leftarrow H(m)$                                                  ▷ message encoding
3:     $k \overset{\$}{\leftarrow} [0, q-1]$
4:     $(u, v) \leftarrow [k] \cdot P$
5:     $r \leftarrow u \bmod q$; if $r = 0$ then goto step 3;
6:     $s \leftarrow k^{-1}(h + rx) \bmod q$; if $s = 0$ then goto step 3;
7:     return $(r, s)$
8: end function
```

Fig. 6. ECDSA signature based on an elliptic curve E over a prime field $\mathbb{F}_p$. P is a base point of order q and H is a cryptographic hash function of output length equal to the size of q. The private key is an element $x \in \mathbb{F}_q$ and the public key is denoted by (p, q, H, P, Q) with $Q = [x]P$.

Informal description	Fault condition	Attack technique	Validity
S is a multiple of p	$S = 0 \mathsf{~mod~} p \wedge S \neq 0 \mathsf{~mod~} q$	GCD computation	Prop. 1
S is an almost full linear combination of p and q	$\exists \alpha, \beta.\ S = \alpha\ p + \beta\ q \wedge \alpha, \beta < 2^{\frac{n}{2}-\varepsilon}$	Orthogonal lattices	Prop. 2
S is an almost full affine transform of p or q	$\exists \alpha, \beta.\ S = \alpha\ p + \beta \wedge \alpha < q, \beta < 2^{n/2-\varepsilon}$	Orthogonal lattices	Prop. 3

Fig. 7. Fault conditions for RSA signatures. The value of ε depends on the size n of the modulus and is a multiple of the words size.

The first additional input is a fault policy; the second, optional input is an upper bound on the number of faults we allow.

Our algorithm exploits many of the standard techniques used in other approaches to program synthesis, including weakest preconditions and invariant generation, and interfaces with SMT solvers for checking the validity of first-order formulae. In addition, our algorithm relies on an automated prover to simplify the intermediate conditions generated by weakest precondition computations; the prover is specialized to formulae that combine arithmetic inequalities and size constraints; such formulae include many fault conditions, including all those we explore in this paper (see Figure 7). On the other hand, our algorithm noticeably departs from recent works on program synthesis by its simplicity: indeed, physical limits on the number and nature of faults sufficiently constrain the search space for faulted implementations, allowing us to dispense from using more elaborate techniques that are required to manage very large search spaces (see Section 6). Experimental results, which we report below, demonstrate that our synthesis algorithm performs well on standard examples.

Application: old and new attacks on RSA and ECDSA signatures The third contribution of our work is a practical evaluation of our approach on RSA and ECDSA signatures. We carry the evaluation using the computer algebra system SAGE, and the EasyCrypt tool[4]. Concretely, we use the former for estimating the effectiveness of lattice-based attacks for different fault conditions, and the latter (or more precisely an implementation of our synthesis algorithm built on top of EasyCrypt) for synthesizing faulted implementations of RSA and ECDSA signatures. During the process, we rediscover many known attacks; moreover, we also discover many new attacks, several of which are efficient attacks of independent interest. We summarize our main findings below:

1. For RSA-CRT signatures based on Garner's recombination, we recover the basic and most efficient attack of [18] which injects a null fault in the last call to CIOS during the computation of modular exponentiation. We also discover a new efficient attack, based on forcing additional iterations in the last call to CIOS. This attack yields almost full affine transforms of p or q, a small number of which is sufficient to recover the factorization of the RSA modulus using orthognal lattices or Simultaneous Diophantian Approximations as in [20,26].

2. For RSA-CRT signatures based on the usual CRT recombination, we discover a new fault attack; to our best knowledge, this is the first efficient fault attack that works with randomized padding. The attack is based on forcing additional iterations in the last call to CIOS and yields almost full linear combinations of p and q. From a small number of such faulty signatures, the factorization of the RSA modulus can easily be recovered using orthogonal lattices.

3. For ECDSA signatures, we discover several new and efficient fault attacks for implementations based on the implementation of scalar multiplication given in Figure 5. A first attack is based on skipping the last iterations in the computation of scalar multiplication. A second attack is based on forcing the evaluation of a conditional inside the loop executed for the computation. The largest group of attacks (containing more than 100 faulted programs) is based on faulting the implementation of the point addition operation. Each faulted signature allows us to recover the least or most significant bits of the nonce; we then finish the attack using classic techniques, and obtain the secret key from a small number of faulty signatures. We also recover an existing attack [35] that lets the faulted algorithm produce valid signatures that may nevertheless be exploited in a similar fashion.

[4] `https://www.easycrypt.info`

2 Background on lattices

Lattice reduction is a powerful tool that is extensively used in the cryptanalysis of public-key cryptosystems. In this section, we provide a brief introduction to some key definitions and algorithms that are used in the paper.

A lattice $\mathbf{L}$ is a subgroup of $\mathbb{Z}^n$, i.e. a non-empty set of vectors closed under addition and inverse. Every lattice $\mathbf{L}$ has a basis, i.e. a finite set of linearly independent vectors that generate all elements in $\mathbf{L}$. Conversely, every set $(\boldsymbol{b_1}, \ldots, \boldsymbol{b_\ell})$ of linearly independent vectors over $\mathbb{Z}^n$ generate a lattice $\mathbf{L} = \langle \boldsymbol{b_1}, \ldots, \boldsymbol{b_\ell} \rangle$ consisting of all integer linear combinations of the $\boldsymbol{b_i}$'s.

A central problem with lattices is to compute nearly reduced bases, i.e. bases that consist of reasonably short and almost orthogonal vectors. There exist many efficient algorithms for performing lattice reductions, including the celebrated Lenstra-Lenstra-Lovasz (LLL) algorithm [29] and Block Korkin-Zolotarev (BKZ) variants [41]. Lattice reduction is an essential tool in cryptanalysis, and we use it extensively in our attacks. For our purposes, it is sufficient to know that the norm of the reduced vectors for a lattice $\mathbf{L}$ of dimension ℓ output by LLL can be approximated by $\alpha\sqrt{\ell} \cdot (\det \mathbf{L})^{1/\ell}$. Hadamard's inequality allows us to upper bound $\det \mathbf{L}$ by $\prod_{i=1}^{\ell} \|\boldsymbol{b_i}\|$. In theory, LLL outputs in polynomial-time a reduced basis and each vector of the base is related to the shortest ones by an approximation factor which is exponential in the dimension. BKZ algorithms allow different tradeoff between the quality of the approximation and the time complexity. In practice, LLL implementations are very fast and when the dimension is much less than 200 [44], it is expected that LLL will output the shortest vector since the approximation factor is $\alpha \approx 1.01$, as shown experimentally in [21]. In larger dimensions, the approximation factor increases (unless we increase the time complexity by a huge factor) and the success probability of our attack is reduced.

To any lattice $\mathbf{L}$ in $\mathbb{Z}^n$ is associated its *orthogonal lattice* $\mathbf{L}^\perp$, defined as the set of all vectors in $\mathbb{Z}^n$ that are orthogonal to all vectors of $\mathbf{L}$. The key properties of $\mathbf{L}^\perp$ are: $\det \mathbf{L}^\perp = \det \mathbf{L}$ and $\dim \mathbf{L}^\perp = n - \dim \mathbf{L}$. Consequently, orthogonal lattices are interesting when their dimension is very small. Indeed, when the dimension of $\mathbf{L}^\perp$ is higher than that of the original lattice, the bounds on the norm of the vectors in reduced bases imply that the norm of the shortest vector in $\mathbf{L}^\perp$ is smaller than the norm of the shortest vector in $\mathbf{L}$. It is possible to reduce the computation of the orthogonal lattice to lattice reduction in polynomial time [38]. Orthogonal lattices were introduced in cryptanalysis by Nguyen and Stern in [37], and have since found many applications [38].

Finding the Shortest non-zero Vector Problem of a lattice (SVP) is a well-known NP-hard problem, that is only approximated by LLL. When the dimension is small and $\alpha \approx 1$, it is likely that LLL solves this problem. In the case of orthogonal lattice, we need good properties for many reduced vectors and we assume that LLL will solve this problem. In the reduced bases output by LLL, vectors are ordered according to their norm and so the last vector of the basis is also the longest. Finally, finding the closest vector of a lattice to some point (CVP) is another NP-hard problem. There exist specific algorithms to solve this problem, but in practice, we use the embedded technique, which is a heuristic reduction from the CVP to the SVP problem.

3 Fault conditions

The primary goal of fault attacks is to induce outputs which satisfy an implementation-independent, mathematical property that guarantees that the secret key or some other confidential data can be efficiently recovered. Our approach critically relies on providing a precise formalization of these mathematical conditions, using *fault conditions*. Informally, a fault condition is a statement of the form

$$v_1, \ldots, v_n : \phi \rightsquigarrow s_1, \ldots, s_k$$

where ϕ is a logical formula that depend on $v_1, \ldots, v_n$, and such that an attacker with access to sufficiently many distinct tuples of values $(v_1, \ldots, v_n)$ satisfying ϕ is able to recover secrets $s_1, \ldots, s_k$

(typically parameters of the cryptosystem) with high probability. More formally, ϕ is a first-order formula over some first-order theory T, for instance modular arithmetic, and moreover all variables that appear free in ϕ but not on the left of the colon can only denote parameters of the cryptosystem.

In this section, we introduce several fault conditions for RSA and ECDSA, and show how, given sufficiently many satisfying values, one can efficiently retrieve either the factorization of the modulus (for the RSA case) or the secret key (for the ECDSA case). Many of these conditions appear implicitly in some variant form in the literature.

Convention All the conditions we consider are of the form $v_1, \ldots, v_n : \phi \rightsquigarrow p, q$ for RSA and $v_1, \ldots, v_n : \phi \rightsquigarrow x$ for ECDSA. Since the secret values $s_1, \ldots, s_k$ are determined by the case study, from now on we simply write $v_1, \ldots, v_n : \phi$.

3.1 Fault conditions for RSA signatures

Throughout this section, we assume that N is an RSA modulus of size n, product of two large primes p and q.

Finding multiples of p or q Our first fault condition considers faulted signatures that are a multiple of p or q. This fault condition enables attacks on RSA by simple gcd computations.

Proposition 1. *Given a single value S satisfying the condition:*

$$S : S \equiv 0 \bmod p \land S \not\equiv 0 \bmod q,$$

one can efficiently factor the RSA modulus N. Obviously the same result holds by switching p and q.

Proof. One can retrieve the factorization of N by performing a simple gcd computation between S and N.

This fault condition is implicit for instance in [18].

Finding "almost full" linear combinations of p and q Our second fault condition considers faulted signatures that are linear combinations of p and q with almost full coefficients. A variant of this fault condition is implicit in [14].

Proposition 2. *Assume that N is a balanced RSA modulus, i.e. p, q such that $p, q < 2^{n/2}$. Given a sufficient number of values that satisfy the fault condition:*

$$S : \exists x, y.\ S = x \cdot p + y \cdot q \land x, y < 2^{n/2-\varepsilon},$$

one can efficiently factor the RSA modulus N. The value ε depends on n and impacts the efficiency and success probability of the algorithm to recover the factorization.

Proof. Assume given vectors $\boldsymbol{S} = (S_1, \cdots, S_\ell)$, $\boldsymbol{x} = (x_1, \cdots, x_\ell)$ and $\boldsymbol{y} = (y_1, \cdots, y_\ell)$ such that $S_1 \ldots S_\ell$ are pairwise distinct and $S_i = x_i \cdot p + y_i \cdot q$ with $x_i, y_i < 2^{n/2-\epsilon}$ for $i = 1, \ldots, \ell$, so $\|\boldsymbol{x}\|, \|\boldsymbol{y}\| < \sqrt{\ell} 2^{n/2-\epsilon}$.

First, note that knowledge of $\boldsymbol{x}$ and $\boldsymbol{y}$ is sufficient to factor N since we can write a system of two linear equations using S_1 and S_2 in two variables p and q. However, it is not possible to gain knowledge of $\boldsymbol{x}$ and $\boldsymbol{y}$ directly and we will use orthogonal lattices to recover them, along similar lines as [14]. The idea is that the orthogonal lattice of $\boldsymbol{S}$ contains $\ell - 2$ vectors that are also orthogonal to $\boldsymbol{x}$ and $\boldsymbol{y}$. More precisely, $\boldsymbol{S}^{\perp} = \{\boldsymbol{x}, \boldsymbol{y}\}^{\perp} \oplus \boldsymbol{u}$, where $\oplus$ denotes here the direct sum of lattices and $\boldsymbol{u}$ is a one dimensional lattice generated by this vector since dimension of $\boldsymbol{S}^{\perp}$ is $\ell - 1$. It is easy to compute the orthogonal lattice of $\boldsymbol{S}$ in polynomial-time using LLL. In the output basis, we can identify the direction $\boldsymbol{u}$ by using some conditions on the reduced vector size thanks to the structure of the vectors in $\{\boldsymbol{x},\boldsymbol{y}\}^\perp$ that we detailed later. By removing this direction from a reduced basis of $(\boldsymbol{S})^\perp$, we get a basis of $\{\boldsymbol{x},\boldsymbol{y}\}^\perp$. Finally, by computing the orthogonal lattice of $\{\boldsymbol{x},\boldsymbol{y}\}^\perp$, we obtain a reduced basis $\{\boldsymbol{x'},\boldsymbol{y'}\}$ of the lattice $\{\boldsymbol{x},\boldsymbol{y}\}$.

Note that $\boldsymbol{x}$ and $\boldsymbol{y}$ are short vectors (we have an upper bound on them). Of course, there is no reason that they are the shortest vectors in the lattice, and $\boldsymbol{x'}$ and $\boldsymbol{y'}$ are likely different from them, so we are not done yet. However, we know that $\boldsymbol{x}$ and $\boldsymbol{y}$ are not too far from the reduced basis (we can estimate the size of the shortest vectors in $\{\boldsymbol{x},\boldsymbol{y}\}$). Therefore, we can enumerate all short vectors in the 2-dimensional lattice $\mathbf{L}' = \{\boldsymbol{x'},\boldsymbol{y'}\}$ to recover $\boldsymbol{x}$ and $\boldsymbol{y}$. The Gaussian heuristic allows us to estimate the number of vectors having a prescribed length and in our case, we only have to test a constant number of vectors. Indeed, $vol(\mathbf{L}') \approx \|\boldsymbol{x}\| \cdot \|\boldsymbol{y}\| \approx 2^{n-2\epsilon}$ and the number of lattice points of norm shorter than $d = \sqrt{\ell}2^{n/2-\epsilon}$ in $\mathbf{L}'$ is $d^2/vol(\mathbf{L}') \approx \ell$ since the lattice $\mathbf{L}'$ is a two-dimensional lattice.

The technical part consists in proving that we can identify $\boldsymbol{u}$ in the reduced basis of $\boldsymbol{S}^\perp$. This vector satisfies the condition

$$p \cdot \langle \boldsymbol{u},\boldsymbol{x}\rangle + q \cdot \langle \boldsymbol{u},\boldsymbol{y}\rangle = 0$$

since it is in $\boldsymbol{S}^\perp$, with $\langle \boldsymbol{u},\boldsymbol{x}\rangle$ and $\langle \boldsymbol{u},\boldsymbol{y}\rangle$ are non zero, otherwise $\boldsymbol{u}$ would also be orthogonal to $\boldsymbol{x}$ and $\boldsymbol{y}$. Consequently, the integers $\langle \boldsymbol{u},\boldsymbol{x}\rangle$ and $\langle \boldsymbol{u},\boldsymbol{y}\rangle$ satisfy $p \cdot a + q \cdot b = 0$ in the integer unknown a and b. Then, $(\langle \boldsymbol{u},\boldsymbol{x}\rangle, \langle \boldsymbol{u},\boldsymbol{y}\rangle)$ is in the orthogonal lattice of (p,q), which is a one-dimensional lattice. It is easy to see that the smallest non-zero vector is $\pm(q,-p)$ and consequently, any non-zero vector has a length larger than $2^{n/2}$, and in particular this implies that $|\langle \boldsymbol{u},\boldsymbol{x}\rangle|, |\langle \boldsymbol{u},\boldsymbol{y}\rangle| > 2^{n/2-1}$. This allows us to derive a lower bound on $\|\boldsymbol{u}\|$ using the Cauchy-Schwarz inequality: $\|\boldsymbol{u}\| \geq 2^\varepsilon/(2\sqrt{\ell})$. Finally, to recover the vector $\boldsymbol{u}$ in the reduced basis of $\boldsymbol{S}^\perp$ which is not in $\{\boldsymbol{x},\boldsymbol{y}\}^\perp$, we remove vectors which have a length larger than $2^\varepsilon/(2\sqrt{\ell})$. In order to have only one such vector, we force vectors of the reduced basis of $\{\boldsymbol{x},\boldsymbol{y}\}^\perp$ to have length shorter than this bound, and thus $\boldsymbol{u}$ will be easily and uniquely distinguished. It is easy to estimate the size of the vectors in the lattice $\{\boldsymbol{x},\boldsymbol{y}\}^\perp$ using its dimension $\ell - 2$ and volume, which can be upper bounded by Hadamard's inequality to $\|\boldsymbol{x}\| \cdot \|\boldsymbol{y}\| \approx 2^{n-2\varepsilon}$. Consequently, if $\sqrt{\ell}(2^{n-2\varepsilon})^{1/\ell-2} < 2^\varepsilon/(2\sqrt{\ell})$, then $\boldsymbol{u}$ will be the largest vector in the reduced basis of $\boldsymbol{S}^\perp$ and in practice it suffices to remove $\boldsymbol{b_{\ell-1}}$ from the reduced basis since the output vectors of LLL are in increasing order.

Relating this fault condition with [14] In [14], the authors force random faults on the modulus during CRT recombination and obtain a fault condition of the following form.

$$\widehat{S} : \exists \alpha, \beta.\ \widehat{S} = \alpha \cdot p(p^{-1} \bmod q) + \beta \cdot q(q^{-1} \bmod p) \wedge \alpha, \beta < 2^{n/2}.$$

If our condition is similar to theirs, the algorithmic problem ours captures is more general. Indeed, in the analysis, the crucial parameter is the ratio between the size of p, q and the size of α, β in the relation $S = \alpha \cdot p + \beta \cdot q$. The larger this ratio is, the easier the attack is since the target vector, called $\boldsymbol{u}$ above is larger. In our case, the size of this vector is close to $2^\varepsilon/\sqrt{\ell}$ while [14] consider a much larger one (their ratio is $\sqrt{N/\ell}$).

p, q	512 (bits)				1024 (bits)			
x_i, y_i	464	472	480	496	968	976	984	992
ℓ	22	26	33	74	37	44	53	67

Fig. 8. Minimal number of signatures ℓ to be faulted depending on the bitsize of x_i, y_i. Almost full linear combinations of p and q.

Finding "almost full" affine transforms of p or q Our third fault condition considers faulted signatures that are almost full affine transforms of p or q. This condition is implicit in [18].

Proposition 3. *Assume that N is a balanced RSA modulus, i.e. p, q such that $p, q < 2^{n/2}$. Given a sufficient number of values that satisfy the fault condition:*

$$S : \exists x, y.\ S = x \cdot p + y \wedge x < q, |y| < 2^{n/2-\varepsilon},$$

one can efficiently factor the RSA modulus N. The value ε depends on n and impacts the efficiency and success probability of the algorithm to recover the factorization.

Proof. Assume given vectors $\boldsymbol{S} = (S_1, \cdots, S_\ell)$, $\boldsymbol{x} = (x_1, \cdots, x_\ell)$ and $\boldsymbol{y} = (y_1, \cdots, y_\ell)$ such that $S_1 \ldots S_\ell$ are pairwise distinct and $S_i = x_i \cdot p + y_i$ with $x_i < q$, and $|y_i| < 2^{n/2-\varepsilon}$ for $i = 1, \cdots, \ell$.

The proof is similar to the previous case, but here we are looking for the $\boldsymbol{y}$ vector since it is sufficient to recover p by computing $\gcd(N, S_1 - y_1)$ and it is the only short vector as $\boldsymbol{x}$ is as large as the unknown p. Consequently, we expect to recover $\boldsymbol{y}$ as the shortest vector $\boldsymbol{x'}$ in $\{\boldsymbol{x'}, \boldsymbol{y'}\}$. The Gaussian heuristic allows us to conclude that $\boldsymbol{y}$ will be the shortest since the volume of $\mathbf{L}'$ is $\|\boldsymbol{x}\| \cdot \|\boldsymbol{y}\| < 2^{n-\varepsilon}$ using Hadamard and so $\|\boldsymbol{x'}\| \approx \sqrt{2}2^{(n-\varepsilon)/2}$. Consequently, $\|\boldsymbol{y}\| < \sqrt{\ell}2^{n/2-\varepsilon} < \|\boldsymbol{x'}\|$ for $\ell < 2^{\varepsilon+1}$.

The technical part is a little different as the condition for $\boldsymbol{u}$ is now:

$$p \cdot \langle \boldsymbol{u}, \boldsymbol{x} \rangle + \langle \boldsymbol{u}, \boldsymbol{y} \rangle = 0$$

with $\langle \boldsymbol{u}, \boldsymbol{x} \rangle$ and $\langle \boldsymbol{u}, \boldsymbol{y} \rangle$ are non zero, otherwise $\boldsymbol{u}$ would also be orthogonal to $\boldsymbol{x}$ and $\boldsymbol{y}$. Consequently, the integers $\langle \boldsymbol{u}, \boldsymbol{x} \rangle$ and $\langle \boldsymbol{u}, \boldsymbol{y} \rangle$ satisfy $p \cdot a + b = 0$ in the integer unknown a and b. Then, $(\langle \boldsymbol{u}, \boldsymbol{x} \rangle, \langle \boldsymbol{u}, \boldsymbol{y} \rangle)$ is in the orthogonal lattice of $(p, 1)$, which is a one-dimensional lattice. It is easy to see that the smallest non-zero vector is $\pm(1, -p)$ which implies $|\langle \boldsymbol{u}, \boldsymbol{y} \rangle| > 2^{n/2-1}$. As before, we can derive a lower bound on $\|\boldsymbol{u}\|$ using Cauchy-Schwarz, $\|\boldsymbol{u}\| \geq 2^{\varepsilon}/(2\sqrt{\ell})$. The end of the proof is the same except that since $\|\boldsymbol{x}\| < \sqrt{\ell}2^{n/2}$, $vol(\mathbf{L}') < 2^{n-\epsilon}$, and so we expect to recover $\boldsymbol{u}$ provided $\sqrt{\ell}(2^{n-\varepsilon})^{1/\ell-2} < 2^{\varepsilon}/(2\sqrt{\ell})$.

p, q	512 (bits)				1024 (bits)			
x_i, y_i	464	472	480	496	968	976	984	992
ℓ	23	28	35	77	39	46	56	71

Fig. 9. Minimal number of signatures ℓ to be faulted depending on the bitsize of y_i. Almost full affine transforms of p or q.

Implementation and evaluation of key recovery Here, we describe how the attacks outlined in the previous paragraphs can be performed. Moreover, we estimate the number of signatures required for recovering the factorization.

Implementation We use the SAGE computer algebra system [44] to implement the attacks. The attacks take as input a sufficient number of signatures $S_1, \cdots, S_\ell$ satisfying the fault condition given in Proposition 2 or Proposition 3. The implementation heuristically recovers the factorization of the RSA modulus N as follows.

- Compute an LLL-reduced basis $\{\boldsymbol{b_1}, \cdots, \boldsymbol{b_{\ell-1}}\}$ of the lattice $(S_1, \cdots, S_\ell)^\perp$. This is done by applying LLL to the lattice in $\mathbb{Z}^{1+\ell}$ generated by the rows of the following matrix:

$$\begin{pmatrix} \kappa S_1 & 1 & & 0 \\ \vdots & & \ddots & \\ \kappa S_\ell & 0 & & 1 \end{pmatrix}$$

  where κ is a suitably large constant, and removing the first component of each resulting vector [37].

- Compute an LLL-reduced basis $\{\boldsymbol{x'}, \boldsymbol{y'}\}$ of the orthogonal lattice $\{\boldsymbol{b_1}, \cdots, \boldsymbol{b_{\ell-2}}\}^{\perp}$. Again, this is done by applying LLL to the lattice in $\mathbb{Z}^{\ell-2+\ell}$ generated by the rows of

$$\begin{pmatrix} \kappa' b_{1,1} & \cdots & \kappa' b_{\ell-2,1} & 1 & & 0 \\ \vdots & & \vdots & & \ddots & \\ \kappa' b_{1,\ell} & \cdots & \kappa' b_{\ell-2,1} & 0 & & 1 \end{pmatrix}$$

and keeping the last ℓ components of each resulting vector.
- For all the linear combinations $\boldsymbol{z}$ of $\boldsymbol{x'}$ and $\boldsymbol{y'}$ that satisfy the size constraints, compute the test $\gcd(z_1 S_2 - z_2 S_1, N)$ or $\gcd(y_1 - S_1, N)$, depending on the fault condition considered, which allows to recover the prime factors p and q.

Evaluation We use our SAGE implementation to evaluate the number of signatures required for the attacks to succeed. The results are given in Figures 8 and 9. We see for instance that 35 values are required to retrieve the factorization of N when p and q are 1024-bit and the size of the x_is and y_is have size 960, i.e. 64 bits shorter than the full size.

3.2 Fault conditions for ECDSA signatures

For ECDSA signatures, we consider fault conditions of a different nature, that rely on partial knowledge of the nonce k used in the computation. We first consider a novel fault condition focusing only on faulting the scalar multiplication. Then, we discuss an already-exploited fault condition where k can be faulted during both the scalar multiplication and the computation of its inverse, as considered in [35]. In both cases, knowing some bits of the nonce k is sufficient to mount a classic lattice-based attack (see Appendix A). In the following, we assume that the message to be signed is known and its hash value is h and we denote $\mathtt{abs}$ the abscissa of an elliptic curve point, $\mathrm{lsb}_\ell\, k$ the ℓ least significant bits of k and $\gg$ for the right-shift operator.

Faulting r. Our fault condition considers faulted signatures such that r is computed using only some of the bits of k:

Proposition 4. *Given sufficiently many values satisfying one of the fault conditions:*

$$r, s : \exists k.\ r = \mathtt{abs}([k \gg \ell] \cdot P) \wedge s = k^{-1}(h + rx) \bmod q \qquad (1)$$

$$r, s : \exists k.\ r = \mathtt{abs}(\pm[2^\ell k \gg \ell] \cdot P) \wedge s = k^{-1}(h + rx) \bmod q \qquad (2)$$

one can efficiently retrieve the secret key x.

The proof of this proposition can be done in two parts, summed up by two facts. We do the proof for condition (1). It should be clear that the same proof applies for condition (2). In particular, Fact 2 tells us that it is sufficient to be able to recover ℓ bits of k to recover the secret key x, and the proof of Fact 1 clearly generalizes to condition (2), since its proof revolves around computations on curve points $\pm[2^\ell k \gg \ell] \cdot P$.

Fact 1 *Given a single pair (r, s) that satisfies the fault condition:*

$$r, s : \exists k.\ r = \mathtt{abs}([k \gg \ell] \cdot P) \wedge s = k^{-1}(h + rx) \bmod q,$$

one can efficiently retrieve the ℓ least significant bits of k.

Proof. Indeed we have the following equalities:

$$[\mathrm{lsb}_\ell\, k] \cdot P = [k] \cdot P - [2^\ell] \cdot [k \gg \ell] \cdot P,$$

$$[k] \cdot P = \left[\frac{h}{s}\right] \cdot P + \left[\frac{r}{s}\right] \cdot Q,$$

the first one results from the property on the faulted value r and the second one from the verification equation of ECDSA. The second equality implies that one can compute $[k] \cdot P$ using public parameters and the signatures. Our goal is to recover $\mathrm{lsb}_\ell\, k$. To this end, we show that we can compute the right hand side of the first equation since the first part is $[k] \cdot P$ and the second part is one of the points whose abscissa is r. Consequently, we can recover $\pm[2^\ell] \cdot [k \gg \ell] \cdot P$ by solving in y the quadratic equation $y^2 = r^3 + ax + b$ if the elliptic curve is the set of points (x, y) satisfying the equation $y^2 = x^3 + ax + b$. For one of these two candidates, the point has a small discrete log related to P and we can recover them using Pollard's lambda method for instance in time $O(2^{\ell/2})$. This algorithm will succeed only for the correct candidate and will likely fail for the other one.

Now, given sufficiently many faulty signatures, the secret x can be recovered using a technique based on lattices.

Fact 2 *Given a sufficient number of ECDSA signatures whose nonces k are partially known, one can efficiently retrieve the secret key x.*

The idea of the proof consists in using the congruence:

$$xr = sk - h \bmod q.$$

This relation reveals nothing about x when k is chosen truly at random in $\mathbb{Z}/q\mathbb{Z}$, but when some bits of k are known, some information about x is leaked since all other parameters are publicly known. Intuitively, knowing ℓ bits of k should reveal ℓ bits of information about x, so if q is n bits long, x should be recoverable from about n/ℓ faulty signatures. Moreover, the relation is affine in both x and k, and the problem can be attacked with lattices. Efficient algorithms for recovering x from a set of such signatures can be found in [23,36] and are briefly recalled in Appendix A, adapted to consider faulted signatures that are not necessarily valid.

Note to conclude that a similar result holds for the most significant bits. For example, condition 1 in this case could be written as follows.

$$r, s : r = \mathtt{abs}([k \bmod 2^{n-\ell}]P) \wedge s = k^{-1}(h + rx) \bmod q.$$

In this case, we retrieve the most significant bits of the nonces and it is not difficult to adapt the lattice to this case.

Using short randomness: faulting r and s. We also consider the following fault condition, implicitly used in the original attack on ECDSA by Nguyen et al. [35], where both the scalar multiplication and field inversion are faulted to simulated short values for k (that is, values whose most significant or least significant bits are zero).

Proposition 5 ([35]). *Given a sufficient number of pairs (r, s) that satisfy the fault condition:*

$$r, s : \exists k.\ r = \mathtt{abs}([\mathit{lsb}(k)] \cdot P) \wedge s = \mathit{lsb}(k)^{-1}(h + rx) \bmod q,$$

one can efficiently recover the secret key x.

Although we do not prove it, the validity of this fault condition is justified by its use in existing attacks.

Implementation and evaluation. We also implement our key recovery attacks on ECDSA in Sage to evaluate their performance. Some experimental values of (ℓ, d) are given in Figure 10.

q	160 (bits)			256 (bits)			384 (bits)	
ℓ	4	8	16	8	16	32	8	16
d	61 ($\simeq$70%)	23	11	38	17	9	61	26

Fig. 10. Minimal number of signatures d to be faulted depending on ℓ using curves brainpoolP160r1, brainpoolP256r1 and brainpoolP384r1. The percentage given in one case represents the success rate of the attack and could be increased by increasing the value of d.

3.3 Discussion

All the fault conditions considered above are intended to predicate over the output of faulted signatures. However, fault conditions may also relate outputs of faulted and valid signatures, or inputs and outputs of signatures. Examples of such fault conditions are given by the original Bellcore attack and by Lenstra's variant:

$$\begin{array}{r}S_1, S_2 : S_1 - S_2 \equiv 0 \bmod p \wedge S_1 - S_2 \not\equiv 0 \bmod q \\ M, S : S - M^e \equiv 0 \bmod p \wedge S - M^e \not\equiv 0 \bmod q\end{array}$$

Both conditions can be further refined. For instance, the fault condition for the Bellcore attack can be refined to express that one of the S_i, say for instance S_1, is a valid signature of a message m, and S_2 is a faulty signature of m. In fact, one can define a partial order on fault conditions[5] and prove that the above fault conditions are smaller than the fault condition given in Proposition 1.

4 Synthesis of Faulted Implementations

In this section, we present an automated tool that synthesizes faulted implementations that verify a fault condition. Our tool is built on top of EasyCrypt [7], a tool-assisted framework for verifying the security of cryptographic constructions.

$$\begin{array}{rlll}\mathcal{C} ::= & \mathsf{skip} & \\ | & \mathcal{C}; \mathcal{C} & \text{sequencing} \\ | & \mathcal{V} \leftarrow \mathcal{E} & \text{deterministic assignment} \\ | & \mathcal{V} \overset{\$}{\leftarrow} \mathcal{DE} & \text{random assignment} \\ | & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & \text{conditional} \\ | & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & \text{while loop} \\ | & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & \text{procedure call} \\ | & \mathbf{return}\ \mathcal{E} & \text{return expression}\end{array}$$

where $\mathcal{V}$ denotes the set of *variables* and $\mathcal{E}$ denotes the set of *expressions* and $\mathcal{DE}$ denotes the set of distribution expressions and $\mathcal{P}$ denotes the set of procedures.

Fig. 11. Syntax of programs

4.1 Programming and assertion language

We consider programs that are written in a core imperative language with deterministic and probabilistic assignments, conditionals, loops, procedure calls, and sequential composition; the syntax of

[5] $(v_1, \ldots, v_n : \phi) \leq (w_1, \ldots, w_m : \psi)$ if there exists an efficient and public n-ary function f that returns m-tuples of values and such that for every $v_1, \ldots, v_n$ such that $\phi(v_1, \ldots, v_n)$ holds and for every $w_1, \ldots, w_m$ such that $f(v_1, \ldots, v_n) = (w_1, \ldots, w_m)$, we also have $\psi(w_1, \ldots, w_m)$.

programs is given in Figure 11. The programming language essentially subsumes the language proposed in [10] and in particular is sufficiently expressive to capture cryptographic implementations.

Expressions used in programs, for instance on the right-hand side of assignments or as guards in conditional statements and loops are built inductively from user-defined constants, operators, and variables. In this paper, we specifically focus on expressions that are built from operations for modular arithmetic, and finite field and elliptic curve operations. We use a simple type system for expressions and programs, and we only consider well-typed programs.

Assertions are first-order formulae over the theories inherited from the expression language. Reasoning about assertions is delegated to the EasyCrypt proof engine, which can either use lemmas from libraries or invoke SMT solvers to prove the validity of an assertion.

4.2 Fault models and fault policies

Fault models Fault models are high-level specifications of the type of faults that can be injected on embedded devices; they generally target specific architectures, and are designed to reflect the effects and capacities of specific perturbation techniques.

For the purpose of this paper, it is sufficient to know that there exist two broad classes of fault models. The first class captures faults that modify the dataflow, for instance by setting a particular register to a default value (the null fault model) or to a constant but unknown value (the constant fault model), or by setting part of the register to a constant value (the zero high-order bits fault model and its variants). In practice, it is often important to consider models that combine several kinds of faults; for instance, one can consider a fault model which allows null faults on small registers, and constant faults on larger registers. Such faults are considered for example in [19], where the authors also justify their practical feasibility. The second class captures faults that modify the control flow, for instance by skipping an instruction (the instruction skip model), by forcing a conditional instruction to enter into a specific branch (the branch fault model), or by forcing the execution of a loop to be interrupted before the guard is set to false, or continued after it is set to false (the loop fault model). These models are classic and are considered in [39], for instance. Both models overlap, in the sense that one can sometimes achieve the same effect by a dataflow fault attack, or by a control flow fault attack.

Fault policies Instead of hardcoding the different fault models, our tool allows users to specify fine-grained fault policies that delineate very precisely the space of faulted implementations by describing which faults can be injected in the program. Fault policies are program specific, and are given by two sets of replacement clauses.

The first set consists of variable replacement clauses of the form (x, e) where x is a program variable and e is an expression; such a clause says that one can replace the variable x by the expression e in the course of program execution. These declarations can be used to model data faults; for instance, the null fault on x is captured by the clause $(x, 0)$, whereas the zero high order bits fault on x that sets r bits to zero is captured by the clause $(x, msb_r(x))$.

The second set consists of command replacement clauses of the form (c, c'), where c and c' are commands; such a clause says that one can replace the command c by the command c' in the course of program execution. These declarations can be used to model control flow faults; for instance, instruction skip faults on an assignment c is captured by the clause (c, skip), whereas branch faults are captured by the clause $(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, c_i)$ where $i = 1$ if the goal is to force execution to go into the true branch, and $i = 2$, otherwise. By convention, we require that all instruction replacements do not increase the set of modified variables, i.e. the set of modified variables of a command c' is a subset of the set of modified variables of the command c it replaces. This is the case for all control flow attacks described above, and is essential for the completeness of our tool.

Although it is useful in practice, fault policies do not currently include a mechanism to impose any locality constraint on the clauses, i.e. replacements may occur anywhere in the program. This can easily be circumvented by writing programs in pseudo-SSA form, for instance by adding subscripts for the different occurences of the same variable in the program.

Finally, fault policies may also include some upper bounds on the number of times a clause can be used to fault an implementation. This is useful to constrain the space of faulted implementations and to match physical constraints.

Discussion There is a direct relation between fault models and fault policies, in the sense that every fault model determines a unique fault policy for each program. However, many fault attacks require multiple faults and can only be captured by hybrid fault models, that combine several simpler ones. An example of hybrid fault model is one that considers null faults on variables that denote small registers (for instance, variables that store valuees smaller than 2^8), and constant faults on variables representing larger registers.

It would be interesting to develop a high-level language for describing hybrid fault models, and a compiler for generating automatically fault policies from high-level specifications. However, building the compiler requires a significant amount of infrastructure, including the ability to automatically infer program invariants: for the example discussed above, the compiler would need to infer that the value held by a variable x is always smaller than 2^8 in order to generate the clause $(x, 0)$. We leave the design of this high-level language and the implementation of the compiler for future work, and require for now that fault policies (albeit in some edulcorated form) are given as input to the tool.

4.3 Algorithm

Our tool takes as input a (non-faulted) implementation written in the programming language of EasyCrypt, a fault condition, and a fault model, and optionally a precondition ψ. It outputs a set of faulted implementations that satisfy the fault condition and are valid faults of the original implementation with respect to the fault model considered. The core of the tool is an algorithm that interleaves the computation of weakest preconditions, logical simplifications, and generation of faults. For simplicity, we describe a non-deterministic and inefficient version of the algorithm, whereas the implementation uses a more efficient implementation, and some caching and early pruning techniques for the smart exploration of the search space. We initially explain how the algorithm works on straightline programs, i.e. programs without loops, conditionals, and procedure calls. Then, we explain how to extend the algorithm to procedure calls and loops. First, we define the notion of faulted instruction.

Faulted commands The fault policy determines for each command c of the program a set of faulted instances, consisting of commands c' that can be obtained from c according to the fault policy. All commands are faulted instances of themselves, and moreover the command c' is a faulted instance of c if there exists an instruction replacement clause (c, c'). Moreover, there are some specific rules for each construct of the language.

- $x \leftarrow e[e_1, \ldots, e_n / y_1, \ldots, y_n]$ is a faulted instance of $x \leftarrow e$, provided for $i = 1 \ldots n$, the replacements of y_i by e_i are allowed by the fault policy.
- the commands $\mathsf{while}\ b\ \mathsf{do}\ c'$, and $\mathsf{if}\ b\ \mathsf{then}\ c'; \mathsf{while}\ b\ \mathsf{do}\ c$, and $\mathsf{while}\ b'\ \mathsf{do}\ c; \mathsf{if}\ b\ \mathsf{then}\ c'$ are all faulted instances of $\mathsf{while}\ b\ \mathsf{do}\ c$, where c' is a faulted instance of c, and b' is a guard that forces exactly one less iteration of the loop body.

The last clause captures faults on the first and last iteration of a loop, and can be extended to model faults on the first and last k iterations of a loop, for $k \geq 1$.

Straighline programs The algorithm is given as input a fault policy, and manipulates triples of the form $(c, \phi, \widehat{c})$. Initially, the algorithm is given the triple (c, ϕ, skip) consisting of the program being analyzed against fault attacks, the fault condition, and the empty statement. At each iteration, the algorithm consumes the last command of c and outputs a new triple $(c', \phi', \widehat{c'})$ as follows;

1. c is decomposed into a sequence $c'; i$, where i is the last command of the program (necessarily an assignment or a random sampling). If c is empty, then the algorithm checks if ϕ is a consequence of the precondition, and returns $\widehat{c}$ if this is the case and nothing otherwise;
2. the algorithm checks whether i affects ϕ, i.e. if any of the variables modified by i occur in ϕ. If not, the algorithm breaks to the next iteration with $(c', \phi, \widehat{c'})$, where $\widehat{c'} = i; \widehat{c}$;
3. if some variable modified by i occurs in ϕ, then the algorithm chooses non-deterministically a faulted instance i' of i;
4. the algorithm computes the weakest precondition of i' on ϕ. For instance, the rules for computing weakest preconditions of deterministic and random assignments are:

$$\begin{aligned}&\mathcal{WP}(x \leftarrow e, \phi) = \phi\{x \leftarrow e\}\\&\mathcal{WP}(x \overset{\$}{\leftarrow} d, \phi) = \forall v \in \mathsf{dom}(d), \phi\{x \leftarrow v\},\end{aligned}$$

   where $\mathsf{dom}(d)$ is the set of values that have a non-zero probability in d. Note that the weakest precondition computation takes an assertion and returns an assertion. This is achieved by viewing probabilistic assignments as non-deterministic assignments over the domain of the distribution from which the assignment is sampled;
5. the algorithm applies logical simplifications to the assertion ϕ output by the weakest precondition computation. The output is a new assertion ϕ' that has fewer free variables than ϕ;
6. the algorithm proceeds to the next iteration with $(c, \phi', \widehat{c'})$, where $\widehat{c'} = i'; \widehat{c}$.

Breaking to the next iteration in step 2 and performing logical simplifications in Step 5 may in fact significantly prune the search space, without ruling out any potential attacks: computing the weakest precondition on a command i whose left-hand-side does not appear in the fault condition never changes that fault condition, whichever fault may be selected. Indeed, our algorithm is sound and relatively complete for straightline code, in the sense that, given an oracle that can decide logical implications, the algorithm would return all faulted versions c' of c such that the Hoare triple $\{\psi\}c'\{\phi\}$ is valid. In practice, logical implications are verified using SMT solvers, and hence the implementation might actually fail to find a valid fault attack.

Procedure calls Our tool deals with programs that make non-recursive procedure calls by entering into the code of the procedure when reaching a call. This is intuitively equivalent to inlining all procedure calls and applying the non-procedural analysis to the inlined code. Although it is certainly possible to develop more sophisticated approaches, including ones that deal with recursive procedure calls, based on state-of-the-art techniques, our elementary approach has the advantage of simplicity and is sufficient for most implementations of cryptographic algorithms.

Loops Dealing with loops is the main source of complexity for our tool, as computing weakest preconditions requires knowing some useful loop invariants, i.e. assertions that hold throughout all iterations of the loop body. We provide two elementary mechanisms for dealing with loops: an invariant generator, and an algorithm for turning (user-provided) invariants for non-faulted loops into invariants for their faulted instances. There is admittedly significant scope for improving these mechanisms, in particular by building upon recent developments in invariant generation; we leave this avenue for future work.

Pruning We use two main pruning techniques for improving the efficiency of the search algorithm. First, since SMT solvers are a clear performance bottleneck, we cache all SMT queries and their result. Second, we maintain a table of all intermediate statements $(c, \phi, \widehat{c})$, and abort execution whenever the algorithm computes a triple which coincides in the first and second component with an element of the table.

5 Applications

Using our tool, we are able to discover many attacks on implementations of RSA-CRT and ECDSA signatures. Several of these attacks are new, and of independent interest. In this section, we review in some detail the most relevant attacks we find.

5.1 RSA-CRT signatures

We consider a CRT-based implementation of RSA that uses the Montgomery ladder (Figure 2) for modular exponentiation and the CIOS algorithm (Figure 1) for modular multiplication. We consider implementations using both Garner's recombination algorithm (Figure 4) and the standard CRT recombination with optimizations (Figure 3). Most of the attacks we find involve faults injected during the last call to CIOS in the ladder (line 13, Figure 2), which takes the result of the exponentiation back into its classical representation. We assume that the parameter x in CIOS is stored in a shift-register, used to extract its individual digits in base b.

Finding multiples of p or q. Using the fault condition from Proposition 1, and allowing null faults on small variables (that contain integers mod b) we recover the most basic and efficient attack of [18], which sets q' to 0 during the final call to $\mathsf{CIOS}(\bar{S}_q, 1)$. In addition, the tool also finds several variants of the fault, indicating which (combinations of) variables can be set to 0 to fulfill the fault condition. For example, setting both u_j and x_j to 0 throughout the computation still yields a null result.

This attack and its variants only work when the final call to CIOS occurs with 1 as second argument. This is not always the case when CRT recombination is used, since the call to CIOS can be used to optimize a multiplication away as illustrated in Figure 3. In this case, by adding control flow faults to the fault policy, our tool also finds that faulting q' to 0 and doubling the number of loop iterations during this final call forces its result to zero. Indeed, in this case, after the normal number of iterations, the shift-register initially containing x now contains zero and any further loop iteration simply shifts a to the right, eventually forcing it to zero as well. A much simpler, albeit much less elegant, control flow fault involves simply faulting the initial loop condition so no computation is performed.

Finding "almost full" linear transforms of p or q. It may not always be possible to skip the loop entirely, or to ensure that the loop is run at least twice as many times as expected. However, it may be easier to inject faults on loop counters that consistently add a small (possibly unknown) number of iterations. Our tool automatically finds that such faults, when q' is set to zero during the additional loop iterations are in fact sufficient to guarantee the fault condition from Proposition 3 using both Garner and CRT recombination. For each additional iteration, the size of the exponentiation's result is reduced by the size of a base b digit, quickly leading to a result that can be exploited by the classic lattice-based attacks described in Section 3.1.

Alternatively, instead of faulting the control flow and a variable, our tool also finds that simply setting q' and x_j to zero during the last iterations of the loop leads to a similarly faulted signature, that fulfills the desired fault condition.

Finding "almost full" linear combinations of p and q. When given the fault condition from Proposition 2, our tool finds that running the previous size-reducing attacks on both half-exponentiations yields a suitable faulted signature when using the classic CRT recombination rather than Garner's. The relative efficiency of the lattice-based attack from Section 3.1 compared to the one from Section 3.1 may justify the additional faults.

5.2 ECDSA signatures

We also run our tool on the ECDSA signature algorithm. We consider an implementation where scalar multiplication is computed using MSB-first Double-and-Add (Figure 5). The main challenge here is that the fault conditions we consider are very precise, in the sense that they give a full functional description of the result depending on some (faulted) inputs. We therefore need not only to be able to find the faults, but also to be able to prove the functional correctness of the non-faulted algorithms.

Faults on the randomness We first consider the fault condition from Proposition 5, that we generalized from [35]. The tool finds that performing a zero-higher-order bit fault on k after it is sampled is sufficient to guarantee the fault condition (as we then have $k = k \gg \ell$). However, we do not automatically find more complex attacks (that use proposition 5) on the algorithms computing scalar multiplications and field element inversions. We believe that our tool would in fact find such attacks given precise enough implementations for these operations, and precise enough loop invariants for their non-faulted versions.

Faults on scalar multiplication. Fault condition (1) from Proposition 4 allows our algorithm to quickly focus the fault search on the computation of the scalar multiplication in ECDSA. The tool discovers that exiting the loop early when computing $[k] \cdot P$, and letting all other computations occur normally, yields signatures (r, s) that fulfill fault condition 4(1).

The second fault condition (2) from Proposition 4 leads to a slightly more flexible overall attack, since it does not require the number of faulted iterations to be known. Given this fault condition and an abstract algorithmic description of the ECDSA algorithm, our tool finds that forcing the branch condition at line 5 (Figure 5) to false for a number of iterations towards the end of the loop yields an exploitable result. Generalizing, faulting line 6 or its implementation such that it computes $R_0 \leftarrow \pm R_0$ instead of $R_0 \leftarrow R_0 + P$ would yield the same result.

Faults on point addition This observation leads us to consider more concrete refinements of the point addition algorithm. In particular, we consider a register-level algorithm for Jacobian-Jacobian point addition, as presented by Murdica [34, Algorithm 36]. This algorithm, shown in Figure 12, Appendix B, is only correct when applied to distinct curve points that are not at infinity or inverse of each other.

Given the implementation where the partial point addition algorithm is wrapped in tests ensuring it is applied correctly (that is, $Q, R \neq \infty$ and $Q \neq \pm R$), our tool quickly finds that faulting the conditional checks is sufficient to force the fault condition: by faulting the test that checks whether the second argument is infinite, we can easily force the wrapped addition algorithm to return its first argument, forcing the fault condition.

However, since the base point P is of order q, and R_0 is always a scalar multiple of P, such checks can be optimized away when the addition algorithm is used for scalar multiplication. With an additional condition that P's abscissa is not zero (which is true for Bernstein's Curve25519 and all NIST-supplied base points, for instance) our tool finds null faults, and some faults in combined models involving null faults and instruction skips, that lead to the faulted computation of $R_0 + P$ returning $-R_0$. Performing this fault during the last iterations of the Double-and-Add loop then yields a faulted ECDSA signature that fulfills fault condition 4(2) and can be used in the lattice-based attack. Our tool yields a list of more than 100 ways to fault point addition in a useful way. Some of them are detailed in Appendix B.

6 Related work

Formal methods for cryptography Our work is more closely related to a recent series of articles that apply formal methods to fault attacks. However, our emphasis is on finding fault attacks against implementations, whereas other works focus on proving absence of fault attacks against algorithmic descriptions or implementations. Two independent efforts by Christofi, Chetali, Goubin and Vigilant [15] and by Rauzy and Guilley [40] prove the absence of fault attacks against RSA-CRT with

Vigilant countermeasure. In a similar spirit, Moro et al. [32] propose an approach based on redundancy to protect implementations against instruction skip attacks. More recently, Barthe, Dupressoir, Fouque, Grégoire, Tibouchi and Zapalowicz [6] formally verify the security RSA-PSS against non-random faults using EasyCrypt [7].

Another recent series of papers use type systems and SMT solvers for verifying whether cryptographic implementations are correctly masked [33,9,17]; in particular, Eldib and Wang [16] have developed a method for synthesis of masking countermeasures.

Synthesis Program synthesis is an active area of research that is undergoing rapid and significant progress, thanks to novel and practically achievable approaches, and to advances in SMT solvers. In contrast to the early works that pursue deductive program synthesis, where the program is extracted from the proof of a theorem, typically a $\forall\exists$ statement, most of the current work focuses on inductive program synthesis, and uses SMT solvers. Many works on inductive synthesis, notably early ones, have focused on loop-free programs [42,22,24]. Other recent works allow synthezing programs with loops; for instance, Srivastava, Gulwani and Foster [43] introduce proof-theoretic synthesis, a variant of synthesis that combines inference of loop invariants and synthesis of loop-free programs. However, this approach is limited to programs whose loop invariants fall into a limited class of assertions. Syntax-guided program synthesis [1] is a recently proposed framework that subsumes many of the previous approaches to synthesis. One ambition of this project is to develop a framework for testing and comparing different implementations, and in particular to provide a common input format inspired from SMT-LIB for synthesis tools. In the future, it would be interesting to suggest automated discovery of fault attacks as a challenge for syntax-guided synthesis competitions.

Our approach shares many similarities with program repair, an instance of program synthesis that aims at automatically eliminating deficiencies in code. Informally, a program repair algorithm takes as input a program p and a property ϕ that must be satisfied by the output of p, and computes by small successive modifications of p a program p' that satisfies ϕ. There exist many approaches to program repair; some of them are based on genetic algorithms [28], others are based on code contracts [45]. We refer the reader to a recent overview [27] for more information. The connection with program repair is very direct; indeed, one can even view faulted implementations as a form of program repair for the attacker. However, the techniques used in program repair are not immediately applicable to finding fault attacks on cryptographic implementations.

7 Concluding remarks

We have presented a new approach to discover automatically fault attacks on cryptographic implementations. The technical core of our approach is a new and practical form of program synthesis. Pleasingly, the tool that implements our approach is able to discover new and interesting attacks. An exciting perspective for further work is to apply our tool to an extensive class of implementations. There are also interesting directions for improving and extending our tool. The first one is to integrate state-of-the-art invariant generation and synthesis techniques in the tool. Another one is to implement a synthesis algorithm based on relational verification in order to deal with relational fault conditions, i.e. fault conditions that relate faulted and valid signatures. Although cast in a different context, the work reported in [5] provides an excellent starting point. Yet another one would be to use synthesis for discovering countermeasures against fault attacks as done in [17] for side-channel attacks.

References

1. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.
2. R. J. Anderson and M. G. Kuhn. Low cost attacks on tamper resistant devices. In B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, editors, *Security Protocols Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 1997.

3. A. Barenghi, G. Bertoni, A. Palomba, and R. Susella. A novel fault attack against ECDSA. In *HOST*, pages 161–166. IEEE Computer Society, 2011.
4. A. Barenghi, G. M. Bertoni, L. Breveglieri, and G. Pelosi. A fault induction technique based on voltage underfeeding with application to attacks against AES and RSA. *Journal of Systems and Software*, 86(7):1864–1878, 2013.
5. G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to SIMD loop synthesis. In *PPOPP*, pages 123–134. ACM, 2013.
6. G. Barthe, F. Dupressoir, P.-A. Fouque, B. Grégoire, M. Tibouchi, and J.-C. Zapalowicz. Making RSA-PSS provably secure against non-random faults. Cryptology ePrint Archive, Report 2014/252, 2014. `http://eprint.iacr.org/`.
7. G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Aug. 2011.
8. A. Bauer, E. Jaulmes, E. Prouff, and J. Wild. Horizontal collision correlation attack on elliptic curves. In *Selected Areas in Cryptology*, Lecture Notes in Computer Science, pages –. Springer, 2013.
9. A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne. Sleuth: automated verification of software power analysis countermeasures. In *Cryptographic Hardware and Embedded Systems-CHES 2013*, pages 293–310. Springer, 2013.
10. M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, May / June 2006.
11. I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 131–146. Springer, Aug. 2000.
12. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 37–51. Springer, May 1997.
13. D. Boneh and R. Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In N. Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 129–142. Springer, Aug. 1996.
14. E. Brier, D. Naccache, P. Q. Nguyen, and M. Tibouchi. Modulus fault attacks against RSA-CRT signatures. In B. Preneel and T. Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 192–206. Springer, Sept. / Oct. 2011.
15. M. Christofi, B. Chetali, L. Goubin, and D. Vigilant. Formal verification of a CRT-RSA implementation against fault attacks. *J. Cryptographic Engineering*, 3(3):157–167, 2013.
16. H. Eldib and C. Wang. Synthesis of masking countermeasures against side channel attacks. In *Computer Aided Verification (CAV'14)*. Springer, 2014. To appear.
17. H. Eldib, C. Wang, and P. Schaumont. SMT-based verification of software countermeasures against side-channel attacks. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 62–77. Springer, 2014.
18. P.-A. Fouque, N. Guillermin, D. Leresteux, M. Tibouchi, and J.-C. Zapalowicz. Attacking RSA-CRT signatures with faults on montgomery multiplication. In E. Prouff and P. Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 447–462. Springer, Sept. 2012.
19. P.-A. Fouque, D. Leresteux, and F. Valette. Using faults for buffer overflow effects. In S. Ossowski and P. Lecca, editors, *SAC*, pages 1638–1639. ACM, 2012.
20. P.-A. Fouque, G. Martinet, and G. Poupard. Attacking unbalanced RSA-CRT using SPA. In C. D. Walter, Çetin Kaya. Koç, and C. Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 254–268. Springer, Sept. 2003.
21. N. Gama and P. Q. Nguyen. Predicting lattice reduction. In N. P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 31–51. Springer, Apr. 2008.
22. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.
23. N. Howgrave-Graham and N. P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001.
24. S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.
25. M. Joye and M. Tunstall, editors. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012.

26. J. C. Lagarias. The computational complexity of simultaneous diophantine approximation problems. In *23rd FOCS*, pages 32–39. IEEE Computer Society Press, Nov. 1982.
27. C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Jornal*, 21:421–443, 2013.
28. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38:54–72, 2012.
29. A. Lenstra, H. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.
30. D. Maimut, C. Murdica, D. Naccache, and M. Tibouchi. Fault attacks on projective-to-affine coordinates conversion. In E. Prouff, editor, *COSADE*, volume 7864 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2013.
31. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.
32. N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, pages 1–12, 2014.
33. A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler assisted masking. In E. Prouff and P. Schaumont, editors, *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2012.
34. C. Murdica. *Physical Security of Elliptic Curve Cryptography*. PhD thesis, Télécom ParisTech, 2014.
35. D. Naccache, P. Q. Nguyen, M. Tunstall, and C. Whelan. Experimenting with faults, lattices and the DSA. In S. Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 16–28. Springer, Jan. 2005.
36. P. Q. Nguyen and I. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Des. Codes Cryptography*, 30(2):201–217, 2003.
37. P. Q. Nguyen and J. Stern. Merkle-Hellman revisited: A cryptoanalysis of the Qu-Vanstone cryptosystem based on group factorizations. In B. S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 198–212. Springer, Aug. 1997.
38. P. Q. Nguyen and J. Stern. Lattice reduction in cryptology: An update. In W. Bosma, editor, *ANTS*, volume 1838 of *Lecture Notes in Computer Science*, pages 85–112. Springer, 2000.
39. D. Page and F. Vercauteren. Fault and side-channel attacks on pairing based cryptography. Cryptology ePrint Archive, Report 2004/283, 2004. `http://eprint.iacr.org/2004/283`.
40. P. Rauzy and S. Guilley. A formal proof of countermeasures against fault injection attacks on CRT-RSA. *Journal of Cryptographic Engineering*, pages 1–13, 2013.
41. C.-P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994.
42. A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.
43. S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, 2010.
44. W. Stein et al. *Sage Mathematics Software (Version 4.8)*. The Sage Development Team, 2012. `http://www.sagemath.org`.
45. Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 61–72. ACM, 2010.

A Proof of Fact 2

We know the ℓ least significant bits of k (see Corollary 1), so we can write $k = 2^\ell(k \gg \ell) + \operatorname{lsb}_\ell k = 2^\ell b + \operatorname{lsb}_\ell k$ for some integer $b \geq 0$. We then get from $xr = sk - h \bmod q$:

$$xr \cdot 2^{-\ell}s^{-1} = b - h \cdot 2^{-\ell}s^{-1} + \operatorname{lsb}_\ell k \cdot 2^{-\ell} \bmod q.$$

Now let t and u two values which can be computed from known or retrieved information, such as:

$$t = r \cdot 2^{-\ell}s^{-1} \bmod q \quad \text{and} \quad u = -h \cdot 2^{-\ell}s^{-1} + \operatorname{lsb}_\ell k \cdot 2^{-\ell} \bmod q$$

The inequality $b < q/2^\ell$ can be expressed in terms of t and u as:

$$0 \leq xt - u \bmod q < q/2^\ell.$$

Therefore, if we denote by $|\cdot|_q$ the distance to $\mathbb{Z}/q\mathbb{Z}$, i.e. $|z|_q = \min_{a\in\mathbb{Z}} |z - aq|$, we have:

$$\begin{aligned} |xt - u - q/2^{\ell+1}|_q &\leq q/2^{\ell+1}, \\ |xt - v/2^{\ell+1}|_q &\leq q/2^{\ell+1}, \end{aligned}$$

where v is the integer $2^{\ell+1}u + q$. Given a number of faulty signatures (r_i, s_i) of various messages, say d of them, the same method yields pairs of integers (t_i, v_i) such that

$$|xt_i - v_i/2^{\ell+1}|_q \leq q/2^{\ell+1}. \tag{3}$$

The goal is to recover x from this data. The problem is very similar to the hidden number problem considered by Boneh and Venkatesan in [13], and is approached by transforming it into a lattice closest vector problem.

More precisely, consider the $(d+1)$-dimensional lattice L spanned by the rows of the following matrix:

$$\begin{pmatrix} 2^{\ell+1}q & 0 & \cdots & 0 & 0 \\ 0 & 2^{\ell+1}q & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 2^{\ell+1}q & 0 \\ 2^{\ell+1}t_1 & \cdots & \cdots & 2^{\ell+1}t_d & 1 \end{pmatrix}$$

Inequality (3) implies the existence of an integer c_i such that:

$$|2^{\ell+1}xt_i - v_i - 2^{\ell+1}c_iq| \leq q. \tag{4}$$

Now note that the row vector, called *hidden vector*,

$$\mathbf{c} = (2^{\ell+1}xt_1 + 2^{\ell+1}c_1q, \cdots, 2^{\ell+1}xt_d + 2^{\ell+1}c_dq, x)$$

belongs to L and $\mathbf{c}$ is very close to the row vector $\mathbf{v} = (v_1, \cdots, v_d, 0)$. Indeed, by (4), the distance from $\mathbf{c}$ to $\mathbf{v}$ is bounded as:

$$\|\mathbf{v} - \mathbf{c}\| \leq q\sqrt{d+1}.$$

We thus have a CVP to solve. In practice, we use an embedding technique to reduce CVP to SVP. This technique consists in computing the $(d+2)$-dimensional lattice L' spanned by the rows of the matrix

$$\begin{pmatrix} L & 0 \\ \mathbf{v} & 1 \end{pmatrix}$$

The row vector $(\mathbf{v} - \mathbf{c}, 1)$ is short, belongs to L' and we hope this is the shortest vector of L'. This assumption implies a condition on the required number of signatures depending on the parameter ℓ and the modulus. An estimate which makes it possible to recover the private key is:

$$d \gtrsim \frac{n}{\ell - \log_2 \sqrt{\pi e/2}}.$$

The above estimate is heuristic, but it is possible to give parameters for which attacks of this kind can be proved rigorously [36].

B Some attacks on elliptic curve point addition

We consider the following algorithm for point addition in Jacobian coordinates, optimized to use a small number of registers. The algorithm is correct whenever neither R nor P is the point at infinity, and $R \neq \pm P$. It assumes R is initially stored in registers (T_1, T_2, T_3), and P in (T_4, T_5, T_6). Note that

```
function EcAdd($R$,$P$)
    $T_7 = T_3 \cdot T_6$;
    $T_8 = T_3^2\{\dagger, \circ\}$;
    $T_3 = T_3 \cdot T_8$;
    $T_3 = T_5 \cdot T_3$;
    $T_9 = T_4 \cdot T_8$;
    $T_8 = T_6^2$;
    $T_1 = T_1 \cdot T_8$;
    $T_8 = T_8 \cdot T_6$;
    $T_2 = T_2 \cdot T_8$;
    $T_9 = T_9\{\star\} - T_1$;
    $T_8 = T_3 - T_2\{\dagger, \circ\}$;
    $T_3 = T_7 \cdot T_9$;
    $T_7 = T_9^2$;
    $T_9 = T_7\{\circ\} \cdot T_9$;
    $T_7 = T_1 \cdot T_7\{\dagger\}$;
    $T_2 = T_2 \cdot T_9$;
    $T_1 = T_8^2$;
    $T_1 = T_1\{\star\} - T_9$;
    $T_1 = T_1 - T_7; \{\star\}$
    $T_1 = T_1 - T_7; \{\star\}$
    $T_7 = T_7 - T_1$;
    $T_8 = T_8 \cdot T_7$;
    $T_2 = T_8 - T_2\{\circ\}$;
    return $(T_1, T_2, T_3)$;
end function
```

Fig. 12. Elliptic curve point addition.

the values of T_4, T_5 and T_6 are untouched by the algorithm. All the faults we detail are such that these registers are left untouched by the faulted algorithm as well.

Our tool finds a large number of fault attacks such that the final state of registers (T_1, T_2, T_3) encodes $-R$. We describe three of them in Figure 12, variables marked with a symbol are set to 0 during the corresponding attack. Full lines marked with a symbol are skipped by the corresponding attack. We run our tool in various fault model and find:

- 1 attack involving 3 null faults (marked using †);
- 116 attacks (some are variants of each other) involving 4 null faults (an example is given using ∘);
- 9 attacks involving 2 null faults and 2 instruction skips (an example is shown using ⋆).