Trapping ECC with Invalid Curve Bug Attacks

Renaud Dubois

Thales Communications and Security
4, Avenue des Louvresses
92230 Gennevilliers – France
firstname.lastname@thalesgroup.com

Abstract. In this paper we describe how to use a secret bug as a trapdoor to design trapped ellliptic curve $E(\mathbb{F}_p)$. This trapdoor can be used to mount an invalid curve attack on $E(\mathbb{F}_p)$. $E(\mathbb{F}_p)$ is designed to respect all ECC security criteria (prime order, high twist order, etc.) but for a secret exponent the point is projected on another unsecure curve. We show how to use this trap with a particular type of time/memory tradeoff to break the ECKCDSA verification process for any public key of the trapped curve. The process is highly undetectable : the chosen defender effort is quadratic in the saboter computational effort. This work provides a concrete hardly detectable and easily deniable example of cryptographic sabotage. While this proof of concept is very narrow, it highlights the necessity of the Full Verifiable Randomness of ECC.

keywords: Bug Attacks, Fault Attacks, ECC, Invalid Curve Attack, ECKCDSA, Kleptography, NSA, Paranoia, Verifiable Randomness, Sabotage-resilient Cryptography.

1 Introduction

1.1 Context

According to the leak of NSA memos of E. Snowden (as reported in the New York Times in 2013 [9]) the NSA managed to become the sole editor of the Dual EC-DRBG Standard [2]. Using this status, the agency inserted a backdoor as part of the large scale bullrun decryption program [10]. This increased the concern about the capacity for an organization to specify trapped cryptosystems. Workshops [8] and Projects like [3] ask the question : "*is there some trap we ignore in the widely used standardized elliptic curve ?*". Those standard curves still resist to classical cryptanalysis (transfers, Pollard-rho, etc.), some like P224 are vulnerable to fault attacks [23] because the twist rho order is below 2^{59}. However the complexity remains high and requires a physical access to the cryptographic device. There is still a possibility that the community is missing a kind of attack relying on either a new theoretic advance or different assumptions. In [14] the authors analyse this possibility of a manipulation of the standards assuming the theoretic existence of an unknown class of attacks. We provide such a class making different assumptions rather than mathematical breakthrough:

- the saboteur may introduce a secret bug in a cryptographic hardware or library,
- the saboteur is able to specify the public elliptic curve $E(\mathbb{F}_p)$ used.

Under those assumptions, we describe how a saboteur can design a specific elliptic curve such that an attacker can forge a valid ECKCDSA signature for any victim using the trapped device/library and curve. The model is strong, but let's adopt the most paranoid point of view. Snowden revealed that intentional hardware modifications of security products sent to targeted organizations was used by NSA. Largest hardware corporations are American companies. As for the software, the debian [5] and heartbleed bug [21] illustrated the possibility to introduce an error in an open source library.

1.2 Prior art

Kleptography In [29], the authors introduced the first description of a cryptographic sabotage, and referred this concept as **kleptography**. They presented a backdoor on RSA key generation such that the saboteur may recover any private key generated by the trapped device. A series of works by Young gave descriptions of backdoored cryptosystems for Diffie-Hellman Key exchange, DSA, ElGamal and Schnorr (see [30], [31], [32]). In [13], a similar work is done over symmetric encryption.

(Un)security model. In [25], the authors introduced a taxonomy to characterize the cryptographic sabotages with different properties :

- secrecy including undetectability, conspiracy and deniability,
- utility including easiness, severity, durability and moniterability,
- scope including scale, precision, and control.

We will discuss the characterization of the constructed trapped system with respect to these properties later on. They also introduce four roles : the saboteur, the victim, the defender and the attacker. In the article the design of cryptosystems is supposed to be done by the defender. Unfortunately history shows that the saboteur himself can be part of the design.

Fault attacks and bug attacks. A fault attack [18] is an invasive attack that uses a physical perturbation (heat, laser, etc.) to perturb the execution of a cryptographic device to extract its secret material. This kind of attack requires having physical access to the cryptographic hardware. In [16] the authors introduce the idea of trapping material with a secret bug to transform fault attacks into possibly remote attack. They then exhibit how some versions of RSA may be broken if a secret bug in a multiplier for a single couple of values known by the attacker exists. Later they give some attacks over simple Pohlig-Hellman cryptosystems. Currently ECC is more and more deployed and the attack is not straightforward.

The bug attack could be exploited to lever the fault attack over Elliptic curve cryptosystems described in [15] into a remote attack, for example over Static Diffie Hellman protocol. But an obvious way to circumvent the attack is to check that the result of computation is still over the curve. Moreover static Diffie Hellman is known to be insecure for many reasons (perfect forward secrecy, etc.) and should not be implemented.

Trapped elliptic curve. In [28], Teske describes a method to insert a backdoor in the design of a public curve. First a very specific weak curve is constructed over the binary field F_2^{161}. Then a public curve related to the weak one by a secret hidden isogeny is published. With the knowledge of the secret isogeny, the ECDLP can be broken, while it cannot on the public curve. Only a few binary fields are suitable for the construction (namely F_2^{161}, F_2^{154}, F_2^{182}, F_2^{189} and F_2^{196}) .They then conclude that any curve over those fields constructed without verifiable randomness may be trapped by the given algorithm. Surprisingly there is not much echo on this devastating result for the credibility of binary curves. Long before Joux's breakthrough on binary field DLP, some decided to avoid those curves due to the suspicions this article inspired.

Our Contribution. In this paper, we describe how to insert a tradpoor in a public curve using a secret arithmetical bug instead of a hidden isogeny. The construction works over any finite field. Then we describe a bug attack over the ECKCDSA verification scheme. The way to mount a bug attack on ECKCDSA is not straightforward. Because of the use of a nonce, we need to break the protocol with a single message. We stress that even if the victim performs sanity checks, that the attack is undetectable by a point testing.

2 Sabotage security

In the following we will consider the following roles :

- the saboteur $\mathcal{S}$: the designer of the trapped device,
- the victim $\mathcal{V}$: the user of the trapped device,
- the attacker $\mathcal{A}$: the entity trying to take advantage of the trap to break the security of the device,
- the defender $\mathcal{D}$: the entity trying to find a trap in a device.

In classical cryptography, the security level τ is defined by the effort required by an attacker to break the system. In symmetric cryptography, the size of the key is equal to the security level, i.e the best attack for an attacker should require as much effort as an exhaustive seach over the key. We define the **sabotage security** σ as the computational effort required by $\mathcal{D}$ to detect a trap.

The sabotage security is directly related to the concept of security level mentioned above but by exchanging the classical role of $\mathcal{A}$ against $\mathcal{D}$ by $\mathcal{D}$ against $\mathcal{S}$. As the security level has impact on the easiness for the user (the higher the security level, the higher the computational cost to compute a protocol), sabotage security level has impact on the easiness for the attacker : the higher sabotage security, the lower detectability but probably the higher computational cost.

The scenario described in this article is as follow:

- $\mathcal{S}$ designs a trapdoored elliptic curve using Algorithm 1,
- $\mathcal{A}$ computes a malicious message in $\sigma 2^{\frac{\sigma}{2}}$ steps using algorithm 3,
- $\mathcal{V}$ computes ECKCDSA verification of M as true using Algorithm 2,
- $\mathcal{D}$ tries to detect the presence of a bug/trapdoor by blackbox inspection in $O(2^{\sigma})$ steps.

Notations In this paper we will write the law over elliptic curve additively. We use the short Weierstrass form $y^2 = x^3 + ax + b$. The following constant will be used:

- p is the characteristic of the prime field $\mathbb{F}_p$,
- $E(\mathbb{F}_p)$ is an elliptic curve over a prime field $\mathbb{F}_p$,
- q is the order of $E(\mathbb{F}_p)$,
- the symbol " $\tilde{}$ " is for a value that has been faulted by a bug at some previous point of the computation,
- τ is the security level. It is usually set as $80, 128, 192$ or 256 according to the application (short or long term secret, civil or military),
- σ is the sabotage security level.

3 Invalid Curve Bug Attacks

3.1 Principle

Invalid Curve Attack was introduced in [15]. It relies on the fact that given the Weierstrass equation $y^2 = x^3 + ax + b$ of an elliptic curve over a prime field $E(\mathbb{F}_p)$ with base point G, the doubling and addition formulas do not depend on the coefficient b. The following table illustrates this property by giving the formulas for affine coordinates (but it is the case for all representation system).

Doubling	Addition
if $y = 0$ then $2P = P_\infty$, else	
$\lambda = \frac{3x^2+a}{2y}$	$\lambda = \frac{y_1-y_2}{x_1-x_2}$
$x_2 = \lambda^2 - 2x$	$x_3 = \lambda^2 - x_1 - x_2$
$y_2 = -\lambda^3 + 3\lambda x - y$	$y_3 = -\lambda^3 + 2\lambda x_1 + \lambda x_2 - y_1$

Table 1. Doubling and Addition law over $E(\mathbb{F}_p)$

Thus, if a point is not checked to be on the curve, $\mathcal{V}$ could be lead to compute over a curve $\tilde{E}(\mathbb{F}_p)$ of equation $y^2 = x^3 + ax + \mathbf{c}$. $\mathcal{S}$ can select $\tilde{E}(\mathbb{F}_p)$ with some weak security properties (for example a very smooth order). A simple way to circumvent this attack is to check if a given point is on a curve. If we only count multiplications, it has a negligible cost of 5 multiplications, compared to an average cost of $C \times 2\tau^\star$ for a scalar multiplication of a point, necessary in most of ECC protocols operation. Recently, this attack was extended to several models of curves (including Edwards and Hessian) [24]. Another kind of attack relying on the projection over a weaker curve is described in [23]. Here a fault is injected at some point of the computation, such that the faulted value results in jumping from a secure curve to its less secure twist.

Invalid Curve Bug Attacks [ICBA] In [19], the authors exhibit how to mount an invalid curve attack on the OpenSSL up to version 0.9.8g. Precisely, the implementation of the modular multiplication over the specific field has a bug which occurs with probability $\approx 2^{-28}$. When the bug triggers, the faulted result of a modular multiplication $r = xy \bmod p$ is $\tilde{r} = xy \pm 2^{256} \bmod p$. This gives the possibility to obtain a result over one of the two faulty curves E_{+256} and E_{-256}. However the ECDLP remains difficult on this two curves (so the ICBA is qualified in the article as "a first attempt"), so the authors show how to break static Diffie Hellman protocol with an adaptative attack requiring 633 queries.

The attack has the following limitations :

- the amount of work required by $\mathcal{A}$ is the same as the one required by $\mathcal{D}$ to detect the bug by black box inspection. The error was found and corrected due to the inspection of faulted execution traces.
- the bug can be prevented by testing if the point belongs to the curve **after** the scalar multiplication, limiting its impact to ECDH only.

[⋆] The constant C depends on the coordinates system used ,C could be $18M$ for projective, the point here is not to discuss which representation is the most efficient.

Note that this bug was present in OpenSSL and fixed, but was also found to be present in Nettle (used for GNUTLS) only very recently (see table 2).

3.2 From genuine error to malicious design

Our thesis is that if $\mathcal{S}$ knows any Zero-Day bug in a cryptographic library prior to the design of a curve, he can use it to break a protocol over this curve. As long as the bug is not publicly revealed, only $\mathcal{S}$ can break the protocol. We show that it is also possible for him to trap a curve for ECKCDSA to hide the occurrence of the bug such that even if the bug is discovered, the curve will not be identified as willingly weak.

First design : ICBA on OpenSSL P256 bug For example, knowing the OpenSSL P256 bug, one could propose the following prime order $P'256$ curve :

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, a = -3 \bmod p, b = 19086 \bmod p$$

$\mid P'256 \mid= 115792089210356248762697446949407573529928204550950567091257955015483879887733$

Contrary to E_{+256} and E_{-256} which seem to behave like random curves, we chose $P'256$ such that ECDLP is easy in E'_{-256} :

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, a = -3 \bmod p, b = 19086 - 2^{256} \bmod p$$

$\mid E'_{-256} \mid= 115792089210356248762697446949407573530293404966142140267282850015878136948013$

$$= 3.11.23.311.840841.5763663089.124818543833.66178631919259.429985282118939.\underbrace{28497959920504661}_{\approx 2^{55}}$$

The simple MAGMA code to generate this curve is given in Annex A.1. Note that instead of just incrementing b, we could use the somewhat rigid process of NIST curves (i.e use a Hash function) until a weak curve is found.

Second design : Circumventing Point testing The idea of our Invalid Curve Bug Attacks is to exploit a secret bug (except for $\mathcal{S}$) such that a valid point at the input of a function will "jump" to a **chosen** invalid weak curve at some point of the computation. We **design** the curve such that the result of a specific computation over this curve is P_∞, which belongs to every curve. Thus the point testing countermeasure is ineficient as both input and output of the scalar multiplication are on $E(\mathbb{F}_p)$. We now describe the principle of the attack with respect to affine representation for simplicity sake, but it can be extended for any other representation. To do this we assume that a faulted multiplier has a secret bug. This bug must have a probability of random apparition lower than $2^{-\sigma}$ to respect the sabotage security level. We assume that the bug is such that the computation of the square of one specific value x gives a wrong result $\tilde{x}^2$. This bug is assumed to happen with an undectable probability (e.g 2^{-128}). With the knowledge of this bug, $\mathcal{S}$ can design a safe curve and an unsafe curve linked to each other by this secret intermediate computation. More precisely the $\mathcal{S}$ looks for a value of a such that the result of the doubling of a point (x, y) is a point of order 2 (i.e $\tilde{y_2} = 0$). This constraint leads to the following system :

$$\begin{cases} \tilde{y_2} = -\tilde{\lambda}^3 + 3\tilde{\lambda}.x - y = 0 \text{ , with} \\ \tilde{\lambda} = \frac{3\tilde{x}^2+a}{2y}, \\ \tilde{x}_2 = \tilde{\lambda}^2 - 2x. \end{cases}$$

Thus given $x, \tilde{x}^2$ and a random y , $\mathcal{S}$ can compute one of the root of the degree 3 univariate polynomial over $\mathbb{F}_p$ of variable a

$$P_{x,\tilde{x}^2,y}(a) = -\left(\frac{3\tilde{x}^2+a}{2y}\right)^3 + 3x\frac{3\tilde{x^2}+a}{2y} - y$$

to obtain the desired value of a. The parameters of the "good" curve are p, a, b, x, y, q and the parameters of the bad curve are $p, a, c, \tilde{x_2}, \tilde{y_2}, 2q'$ with

$$\begin{cases} a &= Root(P_{x,\tilde{x}^2,y}) \\ b &= y^2 - x^3 - ax \\ \tilde{x}_2 &= \tilde{\lambda}^2 - 2x \\ c &= y^2 - \tilde{x}_2^3 - a\tilde{x}_2 \end{cases}$$

Once found, $\mathcal{S}$ obtains two curves, a safe and unsafe one such that :

- $E(\mathbb{F}_p)$ has prime field $\mathbb{F}_p$ and with some luck (i.e iteration over y and SEA (Shoof-Zlkies-Atkin algorithm [26]) executions) prime order q,
- $\tilde{E}(\mathbb{F}_p)$ has order $2q'$,
- The doubling of the $E(\mathbb{F}_p)$ point of coordinates (x, y) with the trapped material projects the result over the point of $\tilde{E}(\mathbb{F}_p)$ $(\tilde{x}_2, 0)$ of order 2.

Note: Using a degree of freedom over the family of bugged square, one could set coefficient a to a specific value, iterate over x and compute a roots of polynomial in y : $P_{x,\tilde{x}^2,a}(y)$. This provides more rigidity (however still less than NIST process).

3.3 Hiding the bug with chosen sabotage security

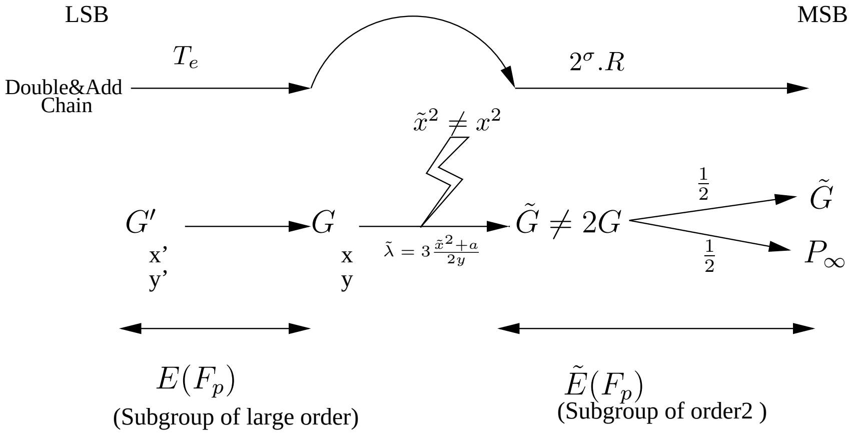

Fig. 1. Trapping $E(\mathbb{F}_p)$ with the bug to jump over $\tilde{E}(\mathbb{F}_p)$.

In order for the bug to be undetectable, $\mathcal{S}$ needs to introduce some backdoor to avoid its disclosure. To do so, he choses a secret value of trapped scalar T_e over $\mathbb{F}_q$ and compute $T_e^{-1} \mod q$ and the scalar multiplication $G' = T_e^{-1}.G$. The length of T_e is set according to $\sigma : \sigma = log_2(T_e)$. The probability to detect randomly the incidence of the bug by random computation is $O(2^{-\sigma})$. Then $\mathcal{S}$ publishes the public parameters p, a, b, q, G'. This process is described in Algorithm 1. Now for a "double and add" multiplication implementation of the computation of a scalar multiplication $\lambda.G'$, if $\lambda = T_e + 2^s.r$, then the computation of the first σ bits of the exponent leads to the intermediate result $G = T_eG'$ as shown in Figure 1. Then the next doubling is the faulted $\tilde{G}$ over the bad curve. Once over the "bad curve" $\tilde{E}(\mathbb{F}_p)$, the rest of the scalar multiplication is $r.\tilde{G}$. One half (the even values) of the $2^{\tau-\sigma}$ possible values for r will give P_∞ as bugged result which representation is common for all curves.

Algorithm 1 Trapped elliptic curve generation

```
Require: $\sigma$, $p$ prime, Bugged square $x \rightarrow \tilde{x}^2 \neq x^2$
q = 0
while $q$ is not prime** do
    Pick random $y$ in $\mathbb{F}_p$
    Compute $P_{x,\tilde{x}^2,y}(a) = (\frac{3\tilde{x}^2+a}{2y})^3 + 3x\frac{3\tilde{x^2}+a}{2y} - y$
    Compute $a = Root(P)$
    if $a \neq \emptyset$ then
        $b = y^2 - x^3 - a.x$
        $q = SEA(p, a, b)$
    Pick a random $T_e$ in $[0..2^\sigma - 1]$
    Compute $G' = (x', y') = T_e^{-1}.G$
Ensure: Public=$(p, a, b, G', q)$ and Secret=$T_e, x, y$
```

Remark : It is the same for a window method with fixed exponent if the length of T_e is multiple of the window size. The construction also works for the Montgomery Ladder and a well chosen T_e (for example with the three first bits of T_e being "101" both chains will take value $2P$ and $4P$ successively).

Example of bug. A first example of such bug could be a hardware bug in the multiplier for two secret values A and B. For a 64 bit architecture, the probability of apparition is 2^{-128}. This bug can be used to meet the requirement of a faulted square by setting the multiprecision integer value to $x = A + 2^{64}B + 2^{128}.R$ (R being any value), so that in the implementation of the squaring a multiplication of A and B will occur. In most implementations, the Montgomery representation is used for efficiency sake. If so x should be changed so that its representant includes the faulted values. The error could also occurs at software level (in the Montgomery multiplication itself for instance). With such trap, we show in Section 4 how $\mathcal{A}$ can forge an ECKCDSA signature for any public key that any trapped verifier material will accept as valid.

3.4 Real word examples

Nasty Montgomery Multiplier. The bug may be introduced at hardware level as suggested in [16], but the error may also occurs at software level. A multiple precision library is a complex tool, and we provide an example of a bugged implementation

** Other conditions like twist rho-order, Embedding degree may be added here due to an error in the carry propagation at the Montgomery multiplication level. The error is touchy and occurs with the multiplication of four words of 64 bits of value 0xFFFFFFFFFFFFFFFF occurs. This specific value seems easy to reach as special moduli are used in most of NIST curves, leading to a detection of the bug, but the use of the Montgomery Representation hides the particularity.

This bug is a misimplementation, hard to detect by a simple inspection and it would be easy to pledge so for the trapper. The probability to reach such a case is negligible. In fact it was detected using the same algorithm on smaller machine words (8 bits) where the probability of occurrence is high.

Open Source Cryptographic library bugs. Recently, bugs have been identified using coverage tools by the security researcher Böck [17]. Table 2 summarizes those bugs. They perfectly fit our description : they concern modular computations, with a very low probability of black box detection. Any of these bugs could have been used with one of the trapping design we introduced in this paper. The fact that the discovery of those bugs are very recent (while the library are old, popular and commonly used) highlights how hard it is to debug a cryptographic library, and how easy it would be to deny a bad intention by a saboteur. The security analysis of those bugs only describe the vulnerability of fixed ECDH. This paper shows that signatures may also be a concern.

Library	Reference	Function	Date	Severity	Description
OpenSLL	[CVE-2014-3570]	Squaring	11/2014	Low	Bug with probability 2^{-128} (resp 2^{-64}) on X86_X64 architecture (resp MIPS 32 bits)
OpenSLL	[CVE-2015-3193]	**Montgomery Squaring**	10/2015	Moderate	Bug with probability 2^{-128} on X86_X64 architecture
NSS (Firefox, Mozilla)	[CVE-2016-1938]	Montgomery exponentiation	01/2016	Medium	Bug with probability 2^{-128} on X86_X64 architecture
Nettle (GNUTLS)	[CVE-2015-8805]	ecc256_modq	02/2016	Critical	Carry error propagation for P256 curve over all architectures

Table 2. Open Source Recent Known Bugs

3.5 Trapped Parameters.

Using the nasty Montgomery multiplier bug, we demonstrate the trapping and construct an example using the same prime field as the curve P384. The Magma code used to generate the tricky curve $E(\widetilde{\mathbb{F}_p})$ and its result is given in annex A.2.

4 Trapped ECKCSA

The ECKCDSA is one of the three elliptic curve signature protocol described in the ISO 15946-2[4]. In this section we describe how $\mathcal{A}$ can exploit the previous trapped curve to forge an ECKCDSA signature for any trapped device. The attack requires a kind of time memory tradeoff such that the computational effort for $\mathcal{A}$ is $O(\sigma.2^{\frac{\sigma}{2}})$, where $(\sigma = log_2(T_e))$. Its memory requirement is $O(2^{\frac{\sigma}{2}})$. In the real life, an attacker could set $\sigma = 80$, leading to a probability of detection of the trap of 2^{-80}, for an attack cost of $O(2^{46})$. The computations are done offline and the attacker only need to submit a single message to the victim device.

4.1 ECKCDSA Description

ECKCDSA is a Korean Signature Standard which doesn't require modular inversion for signing and verifying.It is commonly used as a replacement of ECDSA . To check the validity of a signature (r, s) of message M concateneted to constant certificate information z by a signer with associated public key Q, the verifier process the following algorithm. For more security, it is safer to check that points Q and R are on the curve.

Algorithm 2 ECKCDSA Signature Verification

```
Require: $E(\mathbb{F}_p) : p, a, b, G(x, y), q$, $M$,$z$, $Q$, $(r, s)$.
Ensure: Flag_sigvalid
 1: Set Flag_sigvalid=FALSE
 2: if $(r, s) \notin [1..q-1]^2$ then
 3:     Exit.
 4: $e = r \oplus h(z \mid\mid M)$
 5: $R = sQ + eG$
 6: if $R = P_\infty$ then
 7:     Exit.
 8: if $r \neq h(\lfloor R \rfloor_x)$ then
 9:     Exit.
10: Set Flag_sigvalid=TRUE
```

4.2 Forging message using the secret trap

Algorithm 3 ECKCDSA Message Forge

```
Require: $E(\mathbb{F}_p) : p, a, b, G(x, y), q, z, \mathbf{T_e}$
$P_0 = Q$, $s_0 = 1$, finished=FALSE
for $i \in [1..2^{\frac{s}{2}}]$ do
    $s_i = i + 1$
    $P_i = P_{i-1} + Q$
    $r_i = h(\lfloor P_i \rfloor_x)$
    Sort $r_i$ according to its $\sigma$ LSB bits
repeat
    Select a message $m_j$ and compute $h(z \mid\mid m_j)$
    if $\exists i, r_i = T_e \oplus h(z \mid\mid m_j) \mod 2^\sigma$ then
        $(M, r, s, finished) = (m_j, r_i, s_i, \texttt{TRUE})$
until finished=TRUE
Ensure: Forged signature$(M, r, s)$.
```

Discussion. The aim of this forge is to find a couple (r_i, m_j) such that during the Step 4 of the verification the exponent e will include the trapped exponent T_e. In order to do so we set $(r_i, s_i) = (\lfloor i.Q \rfloor_x, i)$, m_i's are random. As e is the xor of the result of a scalar multiplication r and a hash h, it can be seen as the result of two requests to a random oracle. Finding a specific value of the σ LSB (or MSB) bits can be done by an average of $2^{\frac{\sigma}{2}}$ requests of r and h. This is what means the stop condition (`finished`) of the algorithm. The convergence of the algorithm relies on the birthday paradox, that ensures that for $2^{\frac{\sigma}{2}}$ messages a valid couple (r_i, m_j) exists. The asymptotic complexity is given by the sort of the $2^{\frac{\sigma}{2}}$ elements of the list of r_i's. In practice the most time consuming computation is given by the point addition over Z/qZ. The choice of s_i's as successive elements is motivated to compute efficiently by point addition. A random choice of s_i's would require a far more expensive scalar multiplication. To speed-up the computation it is also possible to choose the m_j's such that only the last block of the message vary, so the hash computation only require one call to the `hash_update` and `hash_final` functions of the hash primitive. Note that the probability of success is $\frac{1}{2}$ because the result of a faulted point multiplication gives P_∞ with probability $\frac{1}{2}$. $\mathcal{A}$ just iterates Algorithm 3.

4.3 Verification by the victim

Now under the previous assumptions, when $\mathcal{V}$ computes the value eG, as $e = T_e \mod 2^\sigma$, during the double and add computation, after the σ first steps, the intermediate result is equal to G', then the rest of the computation are over the bad curve, so with probability one half the result is P_∞. In affine coordinates a simple field may be use to indicate that the point is P_∞, in projective its representation will be such that the (x, y, z) coordinates of G' are such that $z = 0$. In both case the **infinity point has the same representation for any curve**. This means that at step 5 of algorithm 2, we have $\tilde{R} = sQ \in E(\mathbb{F}_p)$. Then by construction $h(\lfloor \tilde{R} \rfloor_x) = r$ and $\mathcal{V}$ accepts the signature as valid.

5 Characterization of the sabotage

We characterize the described sabotage with respect to the criteria defined in [25]. We add a column for the rigidity, which should now be considered as mandatory criteria when designing an elliptic curve. We define it as medium for our ECKCDSA ICBA because only the prime field $\mathbb{F}_p$ is an input of our construction.

5.1 Rigidity and degrees of freedom of the curve generation

The rigidity is a property defined in [3] as the amount of degree of freedom in the generation of the parameter of a curve. A curve is defined either as

- "trivially manipulatable" if no pseudorandom proof is given for the curve at all,
- "somewhat rigid" if some of the parameters are fixed by pseudorandom generation and verifiable,
- "fully rigid" if there is no degree of freedom in the generation.

Currently in the FIPS186 [1], the base point may be chosen by users "to ensure cryptographic separation of networks". Also the ANSX9.62 describe the rigidity as not mandatory, the value `SEED` being optional. All the curves defined in those norms are only "somewhat rigid" as there is no explanation about the mysterious value of the seed. If a property we ignore lies with not negligible probability, it is possible to pick values of `SEED` until the required property is reached.

Our first design works even with "Somewhat rigidity" (like NIST[1] generation), as one can pick a random curve until the order of $|E'_{\pm 256}|$ is smooth. For the second design (Algorithm 1) of curve generation, there is only a limited amount of rigidity (p can be the output of a cryptographic hash function). It also needs to choose the value of the base point to hide the bug. Example of trivially manipulatable curves are the FRP256V1[7] standard published in 2011, and the curve published by the Office of State Commercial Cryptography Administration (OSCCA) [6] in 2010.

The trapping of the curve wouldn't be possible with "full rigidity". A possibility is to used curves like Curve25519 [3]. However this curve uses a very specific field and it may appear safer to have alternatives. Michael Scott already suggested a natural way to do it starting from decimal values of π [27]. A fully rigid original construction is proposed in [12] to generate generic curves from lottery results.

5.2 Attacker and Defender effort

The following table give the rigidity of the curves and the related complexities required by $\mathcal{S}$ and $\mathcal{D}$. The first design requires the same effort for both, while in the second the bug is computationnaly undetectable. If $\mathcal{S}$ is a large organization, it could be assumed that he has a greater computational power than $\mathcal{D}$ but sooner or later the first design bug will be revealed.

Bug&Design	Attacker comp.	Defender comp.	Max. Rigidity
First design	$O(2^{\sigma})$	$O(2^{\sigma})$	Somewhat R.
OpenSSL-P256 OpenSSL-square, OpenSSL-ModExp,Nasty-Montg	$O(2^{28})$ $O(2^{128})$	$O(2^{28})$ $O(2^{128})$	
Second design	$O(2^{\frac{\sigma}{2}})$	$O(2^{\sigma})$	Trivially M.
OpenSSL-P256 OpenSSL-square, OpenSSL-ModExp,Nasty-Montg	$O(2^{14})$ $O(2^{64})$	$O(2^{28})$ $O(2^{128})$	

Table 3. Characterization of both designs

5.3 Comparison of the taxonomy with various historical backdoors

We give a comparison of this work with historical backdoored systems as done [25]. We insist with ECC systems to highlight the difference and similarity.

	Undetectability	Lack of Conspiracy	Deniability	easiness	Moniterability	Severity	Scale	Precision	Control	Rigidity
Debian PRNG [5]	H	H	H	M	M	H	H	M	L	-
Dual EC [2]	M	L	L	M	L	M	L	L	H	L
Trapdoor ECC [28]	H	L	L	L	H	H	H	L	H	L
ECKCDSA ICBA	H	H	L	M	M	H	H	L	H	M
Static DH ICBA	H	H	L	M	M	L	H	L	H	H

Table 4. Taxonomy of various cryptosystems

5.4 Defender Toolbox

Fault Resistant Implementation. An important fact is that the result of the faulted computation is the infinity point. Therefore if the fault verification consists in checking that the point is on the curve, the result is true as it belongs to any curve. Moreover, at this step, most implementation won't test that the result is not the point at infinity because the input condition (e being different from q) guarantees this property in a faultless code. A simple countermeasure is to use any of the Coron [20] countermeasures. One of them is to use a projective equivalent by choosing $x', y', z' = \lambda.x, \lambda.y, \lambda.z$. As the original value x is replaced, the square of x' is computed correctly. For an attacker with no physical access to the device, usually this countermeasure is not necessary. As it has a computational cost, one could be tempted to ignore the threat of faults. The point testing step is enough to prevent remote attack. This work stresses that this countermeasure should be used anyway to prevent unknown trap.

Black box testing. If $\mathcal{D}$ doesn't have access to the code, it should test intensively each set of parameters. The asymmetric cryptography give a rare opportunity to assess the correctness of implementation. For the P256 bug, a **toughness test**, consisting in signing/verifying on random input would have spotted the P256 bug.

Code coverage and Scalability. This work shows that with black box inspection only, $\mathcal{S}$ wins the game against $\mathcal{D}$. A way to increase trust would be to require formally verified implementation of elliptic curve cryptography [22], but it is a very difficult task to obtain and not compatible with high throughput. A tradeoff is to use **code coverage** or scalability. Code coverage consists in providing test vectors to test each branch of a program. However as stated in [17], the branch free implementation of cryptographic libraries (related to side-channel attacks) makes the inspection of code more difficult. Some of the bugs discovered in Table 2 has been found by using the special vectors of a branched implementation in a non branched one. The **scalability** of a library consists in having a generic implementation over machine words chosen from 8 bit to 64 bit architectures. On a 8 bit architecture, the special cases will appear more frequently and carry errors should be detected.

6 Conclusion

This work is a proof of concept of the design of a hardly detectable trapped cryptosystems. There are easy countermeasures, but the important point is that it is easy for a manufacturer to design a white box malicious implementation and nevertheless obtain certification of its products.

The attack implies a very strong model : the bug on the modular multiplication is common, but the elliptic implementation is not. The Montgomery ladder, Coron's countermeasures, Shamir's trick (Strauss ladder) are not compatible with it, thus minimal state of the art implementations. However despite bad implementation, it would pass all FIPS tests. All of this wouldn't be possible with verifiable randomness. The main point is that it is not clear what kind of vulnerabilities we may be missing by using non rigid curve. The mathematical tools required are very basic, but the overall complexity for a defender comes from the assembly of bug, trapped curve, propice protocol. To answer the initial question ”*is there some trap we ignore in the widely used standardized elliptic curve ?*”, this preliminary work stresses that it is plausible that some flaw with the combination of several assumptions hard to relate to each other might exist in our standard. We definitely recommend to avoid the use of the P-series NIST curve [1] except for the validation of code. As for ANSSI and OSCCA, an interesting (and a bit provoking) fact is that knowing the exact specification of this sabotage, it is not possible to say if they are trapped for ECKCDSA or not.

The described method is only one from many possibilities, we are still looking to way to trap more rigid curve (i.e switching from choice of x, y, a, b related to each other to more degrees of freedom). In the next years, the development of Post Quantum protocols will certainly requires constants which will have to be verifiable as well (see [11] for a PQ-trap). An extension of this work is to extend the concept to other kinds of protocols.

Acknowledgement The author wish to thank Olivier Bernard for helpful comments and discussion.

References

1. Digital signature standard. *Federal Information Processing Standards Publication 186-2. 2000.*
2. Recommandation for random number generation. *Special Publication 800-90, 2012.*
3. Safecurves: Choosing safe curves for elliptic-curve cryptography. *http://safecurves.cr.yp.to/index.html.*
4. Information technology - security techniques - cryptographic techniques based on elliptic curves-part 2 : Digital signatures. Technical Report 15946-2, 2002.
5. Debian security advisor : Openssl predictable random number generator. 2008.
6. Public key cryptographic algorithm sm2 based on elliptic curves. *State Commercial Cryptography Administration (OSCCA), China*, page 6 and 13, 2010.
7. Publication d'un paramtrage de courbe elliptique visant des applications de passeport lectronique et de l'administration lectronique franaise. *Agence Nationale de la scurit des systmes d'information.*, pages 2–6, 2011.
8. Awacs 2016. a workshop about cryptographic standards. 2016.
9. Government annouces steps to restore confidence on encryption standards. *The New York Times*, September 10, 2013.
10. Revealed : how us and uk spy agencies defeat internet privacy and security. *The Guardian*, September 5, 2013.
11. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post quantum key exchange - a new hope. *eprint 2015, report1092.*, 2015.
12. T. Baignières, C. Délerablée, M. Finiasz, L. Goubin, T. Lepoint, and M. Rivain. Trap me if you can - million dollar curve. *eprint 2015, report1249. https://eprint.iacr.org/2015/1249.*
13. M. Bellare and Z. Brakerski. Security of symmetric encryption against mass surveillance. In *Crypto 2014*, volume 8616 of *LNCS*, pages 1–19, 2014.
14. D. Bernstein, T. Chou, C. Chuengsatiansup, A. Hulsing, E. Lambooij, T. Lanje, R. Niederhagen, and C. Vredendaal. How to manipulate curve standards : a white paper for the black hat. *Cryptology eprint Archive. report0571.*, 2014.
15. I. Biehl, B. Meyer, and V. Mller. Differential fault attacks on elliptic curve cryptosystems. *Springer, Advances in Cryptology, CRYPTO 2000*, pages 131–146, 2000.
16. E. Biham, Y. Carmeli, and A. Shamir. Bug attacks. *Journal of Cryptology*, 29:1–31, 2015.
17. H. Böck. Fuzzing project report december 2015. 2015.
18. D. Boneh, R.A. DeMillo, and R.Lipton. On the importance of eliminating errors in Cryptographic Computations. In *Journal of Cryptology*, volume 14, pages 101–120, 2001.
19. B.B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. *Springer, Advances in Cryptology, CT-RSA 2012*, 7178:171–186, 2012.
20. J.S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. *Springer, CHES 1999*, pages 292–302, 1999.

21. Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, , and al. The matter of heartbleed. *In ACM Internet Measurement ConferenceIMC, 2014.*
22. S. Fischer. Formal verification of a big integer library. *Workshop on Depandable Software Systems at DATE'08*, 2008.
23. P.A. Fouque, Reynald Lercier, Denis Réal, and Frédéric Valette. Fault attack on elliptic curve with Montgomery Ladder. *FDTC'08*, pages 92–98, 2008.
24. S. Neves and M. Tibouchi. Degenerate curve attacks - extending invalid curve attacks to edwards curves and other models. *Springer, Advances in Cryptology, Public Key Cryptography 2016, PKC2016*, 2016.
25. B. Schneier, M. Fredrikson, T. Kohno, and T. Ristenpart. Surreptitiously weakening cryptographic systems. *eprint 2015, report097. https://eprint.iacr.org/2015/097.*
26. R. Schoof. Counting Points on elliptic curves over finite fields. In *J. Theorie des nombres Bordeaux 7*, Les dix huitiemes journees arithmetiques, pages 219–254, 1995.
27. Mickael Scott. Re: Nist announces set of elliptic curves. 1999.
28. E. Teske. An elliptic curve trapdoor system. *J. Cryptology*, 19:115–133, 2006.
29. A. Young and M. Yung. The Dark Side of "Black-Box" Cryptography, or: Should We Trust Capstone? In *Crypto 1996*, volume 1109 of *LNCS*, pages 89–, 1996.
30. A. Young and M. Yung. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In *Crypto 1997*, volume 1294 of *LNCS*, pages 264–276, 1997.
31. Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In *Eurocrypt 1997*, volume 1233 of *LNCS*, pages 62–, 1997.
32. Adam Young and Moti Yung. Monkey: Black-Box Symmetric Ciphers Designed for MONopolizing KEYs. In *FSE 1998*, volume 1372 of *LNCS*, pages 122–, 1998.

A Annex : Magma

A.1 Design weak curve for P256 OpenSLL 0.9.8g bug

```
/* Generating a weak curve for the OpenSSL 0.9.8g bug*/

function RhoSecurity(q) /*return the greatest divisor binary size*/
 res:=Factorization(q);
 /*largest divisor */
 largest_divisor:=res[#res][1];
 return Log(largest_divisor)/Log(2);
end function;

/*modulus of original P256*/
p:=(2^256-2^224+2^192+2^96-1);

a:=p-3;
b:=5;
b:=19086;//starting value with already weak curve
min:=512;
Fp:=GF(2^256-2^224+2^192+2^96-1);

while(true) do
  /*print "b=",b;*/
  E_Fp := EllipticCurve([Fp!a, Fp ! b]);
  q:=SEA(E_Fp: MaxSmooth := 1);

  if IsPrime(q) then
 print "SEA completed with prime q=",q;
   cp256:= Fp!(b+2^256);
 E_Fp := EllipticCurve([Fp!a, Fp ! cp256]);
   q:=SEA(E_Fp);
 min:=Min(min,RhoSecurity(q));
   print "min for E+=",min, "b=",b, "cp256=",cp256,"q=",q;
 cm256:= Fp!(b-2^256);
 E_Fp := EllipticCurve([Fp!a, Fp ! cm256]);
   q:=SEA(E_Fp);
   min:=Min(min,RhoSecurity(q));
   print "min for E-=",min, "b=",b, "cm256=",cm256,"q=",q;
  end if;

       b:=b+1;
end while;
```

Note : Short magma computations may be executed at `magma.maths.usyd.edu.au/calc`. Copy Paste the previous code to obtain the weak curve E'_{-256} described in the article (use command print to display). A timeout will occur but the first iteration to compute P',E'_{-256} and E'_{+256} is performed. The largest prime subgroup order of E'_{-256} is displayed. Let the code run on a full magma version to compute weaker and weaker curves. We stop the code at iteration b=71575 after 18 hours of computation. Weakest curve was found for b=19086.

A.2 Design a weak curve for any input bug

Example on the Nasty Montgomery Multiplier

- We took the prime field of curve P384

  ```
  p=39402006196394479212279040100143613805079739270465446667
    9482934042457217714968703290472660882589380018616069731123319.
  ```

- We set the value of the base point to the value that fault when squared

  ```
   x=36939380588943957081609007219436164662343008267191841 7
     4605740716166525599560284862385226450010903198127151387561 1635.
  ```

  The underlying reason of the bug is that the Montgomery representant hexadecimal value in base 2^{64} of x is `0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF, 0, 1, 0xF000000000000000`. A carry propagation error leads to the fact that x^2 is (wrongly) equal to $\tilde{x}^2 =$

  ```
  198549103319280409870596578564863768173241930473338755541170
  7807139360972639936186917704858123306422522240711209 6178$.
  ```

- We implemented Algorithm 1 in magma and simply choosed succesive values of y until stop condition is true. The value of the trapped exponent T_e is set to the 80-bit value '666666666666666666666666' (to show how chosen and evil it could be).

Magma Code

```
/* Original P384 curve modulus          */
 p:=39402006196394479212279040100143613805079739270465446667948\
29340424572177149687032904726608825893800186160697311231 9;
/*starting value for y*/
 y:=0;
/*to speed up the iteration over y for demonstration : first iteration is success*/
 y:=1287;

 q:=0;
 Fp:=GF(p);
 PolFp<X>:=PolynomialRing(Fp);
 rac:=[];
/************************************************/
/*Search for a prime order elliptic curve over Fp
/************************************************/
/* INPUT BUG*/
x:= 36939380588943957081609007219436164662343008267191841746057 40\
71616652559956028486238522645001090319812715138756116 35;
/*the faulty square, result of x^2 (see hexadecimal value of x upper)*/
x2bar:=Fp!19854910331928040987059657856486376817324193047333875 55\
41170780713936097263993618691770485812330642252224071120961 78;

flag_primecurve:=false;
while flag_primecurve eq false do
flag_root:=false;
Y:=Fp!y;
print "\n y=",y ;
y:=y+1;

while flag_root eq false do
y:=y+1;

Lamdabar:=(3*x2bar+X)/y;
x3:=Lamdabar^2-2*x;
y3:=-Lamdabar*(x3-x)-y;
rac:=Roots(y3,Fp);
Y:=Fp!y;
print "rac=",rac;
if #rac ne 0 then
flag_root:=true;
anew:=rac[1][1];
bnew:= Y^2-(Fp!x)^3-anew*(Fp!x);
end if;
end while;

E_Fp2 := EllipticCurve([Fp!anew, Fp ! bnew]);
  print("\n SEA");
```

```
q2:=SEA(E_Fp2 :MaxSmooth :=1);

flag_primecurve:=IsPrime(q2);
end while;

/****************************************************************************/
/* value of c for the faulty curve with base point (X3,Y3) and coefficient a*/
/****************************************************************************/
Lamda:=(3*x2bar+anew)/y;
X3:=Lamda^2-2*x;
Y3:=-Lamda*(X3-x)-y;
cnew:= Y3^2-(Fp!X3)^3-anew*(X3);

G:=E_Fp2![x,y];/*Generating point before faulted square*/
evil_backdoor:=666666666666666666666666; /*The backdoor exponent T_e*/
Fq2:=GF(q2);

inv:=Integers()!((Fq2!evil_backdoor)^-1); /*T_e^-1*/
Trapped_G:=inv*G; /* The public base point G'=Te^-1.G*/

E_broken := EllipticCurve([Fp!anew, Fp ! cnew]);/*Curve with base point of order 2*/
G_broken :=E_broken![X3,Y3];
qbroken:=SEA(E_broken);

print "first coefficient of good curve a=",anew;
print "second coefficient of good curve b=",bnew;
print "second coefficient of bad curve c=", cnew;
```

A.3 Result of computations

Copy Paste the previous code at `magma.maths.usyd.edu.au/calc` to obtain the following results. We only displayed a, b and c coefficients. (Other prints may be added in the code).

Cryptographically "Good" Curve $\boldsymbol{E}(\mathbb{F}_{\boldsymbol{p}})$ Magma Version : 2.15_14

```
Prime order of Good curve
q2=394020061963944792122790401001436138050797392704654466667
946201705329262751545450690436602728229344419988978707511011

Base point of Good curve that fault at doubling
x=36939380588943957081609007219436164662343008267191841
7460574071616652559956028486238522645001090319812715138756116 35
y=1289;

Another generating point hidden by evil_backdoor prefix multiplication
x=345905649600701700123808194814761334555599739012455858945 5
197983713765173109049975298644342184403273214488213654203 2
```

```
y=114206713525978474709882504211423428570111721926405016017830
1581724019748878030456681814010279081152012949423980608 5
```

```
First and second coefficients of good curve
anew:=14367930229711479883350915186118982860188203847075728
760652649533264083000146742134994819780135817207484978476863149
bnew:=1094909963319946420405405309030750338745147125012343394397 0
33555231119660085366798239719198307761590854067563906234 1
```

Cryptographically "Weak" Curve $\tilde{E}(\mathbb{F}_p)$

```
Abscisse of point of order 2 over bad curve (y=0)
Composite order of bad curve
qbroken:= 3940200619639447921227904010014361380507973927046544666794538835
868294942161006161888356066445743993111031246243001 6
```

```
Second coefficient of bad curve (first is anew)
cnew:=963075812436490218851581420912108609778000250454 1
2837398201736839034370078776260604078895756605250812239362266709 27
```

```
Abscisse of point of order 2 over bad curve (result of faulted doubling : y=0)
Gxbroken:=37461763926852220391385989298186307975823546614445560438856010 1
283977902513325964714639881505369966706680601062807 16
```

B Annex : ECKCDSA Signature and Key Generation

We provide here the description of ECKCDSA Verification and Key Generation functions for completeness.

Algorithm 4 ECKCDSA Key Generation

```
Require: curve $(E, P)$
Pick random $d$ in $(Z/qZ)^*$
$Q = d^{-1}.P$
Ensure: Private key $d$ and Public Key $Q$
```

Algorithm 5 ECKCDSA Signature Generation

```
Require: private key $d$, certificate information $z$, message $M$, curve $(E, P)$
Pick random $k$ in $(Z/qZ)^*$
$(x_R, y_R) = k.P$
$r = H(x_R)$
$e = H(M, z)$
$\omega = r \oplus e \bmod q$
$s = d(k - \omega)$
if s=0 then
    Goto Step 1.
Ensure: Signature $(r, s)$
```