2019 IEEE European Symposium on Security and Privacy (EuroS&P)

# Degenerate Fault Attacks on Elliptic Curve Parameters in OpenSSL

Akira Takahashi
Department of Computer Science, DIGIT
*Aarhus University*
Aarhus, Denmark
takahashi@cs.au.dk

Mehdi Tibouchi
*NTT Secure Platform Laboratories*
Tokyo, Japan
mehdi.tibouchi.br@hco.ntt.co.jp

***Abstract*—In this paper, we describe several practically exploitable fault attacks against OpenSSL's implementation of elliptic curve cryptography, related to the singular curve point decompression attacks of Blömer and Günther (FDTC2015) and the degenerate curve attacks of Neves and Tibouchi (PKC 2016).**

**In particular, we show that OpenSSL allows to construct EC key files containing explicit curve parameters with a compressed base point. A simple single fault injection upon loading such a file yields a full key recovery attack when the key file is used for signing with ECDSA, and a complete recovery of the plaintext when the file is used for encryption using an algorithm like ECIES. The attack is especially devastating against curves with $j$-invariant equal to 0 such as the Bitcoin curve secp256k1, for which key recovery reduces to a single division in the base field.**

**Additionally, we apply the present fault attack technique to OpenSSL's implementation of ECDH, by combining it with Neves and Tibouchi's degenerate curve attack. This version of the attack applies to usual named curve parameters with nonzero $j$-invariant, such as P192 and P256. Although it is typically more computationally expensive than the one against signatures and encryption, and requires multiple faulty outputs from the server, it can recover the entire static secret key of the server even in the presence of point validation.**

**These various attacks can be mounted with only a single instruction skipping fault, and therefore can be easily injected using low-cost voltage glitches on embedded devices. We validated them in practice using concrete fault injection experiments on a Rapsberry Pi single board computer running the up to date OpenSSL command line tools—a setting where the threat of fault attacks is quite significant.**

***Index Terms*—OpenSSL, Invalid curve attack, Fault attack, Embedded security, Singular curve, Supersingular curve**

## I. Introduction

### A. Physical Attacks against Cryptographic Devices

As the number of devices holding sensitive information is on the rise, it is essential to actively research and develop cryptographic schemes that remain secure even when deployed in real life conditions; more concretely, we have to take into account the existence of physical attacks against the devices that execute cryptographic algorithms. Physical attacks are very powerful tools that allow adversaries to deviate from traditional security models and ultimately bypass computationally hard problems. One can roughly classify physical attacks into two types. The first one, *side-channel analysis*, consists of passive attacks that attempt to recover secret information from the physical leakage of cryptographic computations, such as the time it takes to carry out certain operations, or the power consumption of the device as the computation is performed. The second one, *fault analysis*, consists of even stronger, active attacks that seek to learn secrets by deliberately tampering with the device to cause malfunction or otherwise unexpected behavior, by modifying the voltage of the power source at carefully chosen points in time, subjecting the device to sudden changes of temperature, etc. [1]. These types of attacks have been experimentally shown to be feasible in realistic settings, and do, in fact, affect the security of numerous cryptographic primitives and protocols in the real world. As the advent of Internet of Things (IoT) is likely to make this threat even more pressing, evaluating the power of physical attacks and proposing appropriate countermeasures before deploying new cryptographic schemes is crucial in preventing sensitive information from getting into the wrong hands. [2]

### B. Implementation Attacks against ECC

Elliptic curve cryptography (ECC) is frequently used nowadays because it offers relatively short key length to achieve good security strength. Elliptic curve-based cryptographic schemes typically operate in the group of rational points of an elliptic curve over a finite field, and their security relies on the hardness of the elliptic curve discrete logarithm (ECDLP) or related problems. Possibly the best-known such schemes are the Elliptic Curve Digital Signature Algorithm (ECDSA) [3], the Elliptic Curve Diffie–Hellman key exchange (ECDH) [4], and the Elliptic Curve Integrated Encryption Scheme (ECIES) [5]. Since ECC schemes are standardized and adopted in many cryptographic libraries like OpenSSL [6], their security against physical attacks, such as fault attacks [7], is of prime concern.

Biehl et al. [8] addressed the first fault attacks against ECC, and various related techniques have been developed in the literature since. One of the most recently discovered attacks, which we extend in this work, is the *singular curve point decompression (SCPD) attack* by Blömer and Günther [9]. The idea of the SCPD attack is quite simple but its effect is highly destructive: if the base point of an elliptic curve (in short Weierstrass form) with $j$-invariant 0 is computed from the compressed form before its use in scalar multiplication, one can completely bypass the ECDLP by injecting


DOI 10.1109/EuroSP.2019.00035

a single instruction skipping fault, and consequently recover the secret scalar. The recovery technique is based on the fact that, after skipping a suitable instruction during point decompression, the perturbed decompressed base point will lie on a singular cubic curve of additive type; in other words, when carried out on that singular curve, the group operations correspond to a group structure isomorphic to $\mathbb{F}_p^+$, the additive group of the base field. Blömer and Günther mounted the SCPD attack on an AVR microcontroller running the Boneh–Lynn–Shacham (BLS) signature scheme [10] instantiated with Barreto–Naehrig (BN) curves [11], and successfully recovered the secret key. However, that signature scheme is not nearly as commonly used as a standardized scheme like ECDSA. In addition, the simple, traditional countermeasure of point validation thwarts the SCPD attack and the authors had to inject an additional fault to eliminate it, which assumes a very powerful adversary and would be much harder to achieve against more complex targets such as Linux-based embedded systems. Therefore, the practical impact of the attack appeared limited.

As a less invasive implementation attack than fault attacks, Antipa et al. [12] initiated the study of the *invalid curve attacks*. These types of attacks usually exploit careless implementations that do not check if the input point satisfies the predefined curve equation. The adversary's basic strategy with invalid curve attacks can be summarized as follows: 1) pick some malicious point $\widetilde{P}$ on a weak curve $\widetilde{E}$ where recovering partial information of the secret scalar is computationally easy, 2) send $\widetilde{P}$ to the scalar multiplication algorithm, and 3) compute partial bits of the secret scalar $k$ by examining an invalid output $[k]\widetilde{P}$. The original invalid curve attacks only targeted curves in short Weierstrass form, and were only applicable against the ECC schemes using point arithmetic that is independent of at least one of the curve parameters, which is not the case for some newer curve models such as high-profile (twisted) Edwards curves [13]. However, Neves and Tibouchi [14] recently presented an extension of the invalid curve attack, which they call *degenerate curve attacks*, and showed that similar attacks can even be exploitable against other curve models including Edwards curves. They also described a Pohlig–Hellman-like technique [15] to mount the attack in a situation where the adversary cannot obtain the raw result of scalar multiplication, but can only get the hash of it. Though such a setting does appear in practical instances of ECDH, their targeted model fails to capture the significant property of real-world protocols: in most ECDH implementations, a shared secret key is not derived from the resulting curve point itself, but from its $x$*-coordinate*. Hence their approach cannot be directly applied to widely deployed ECDH implementations.

### C. Our Contributions

In this paper, we identify fault attack vulnerabilities in OpenSSL's elliptic curve cryptographic algorithms. Our main contributions can be summarized as follows:

- As our first contribution, we present a variant of the SCPD attack and its direct application to OpenSSL's elliptic curve-based digital signature and public key encryption. In particular, we show that OpenSSL allows to construct EC key files containing curve parameters with a compressed base point. A simple single fault injection upon loading such a file yields a full key recovery attack when the key file is used for signing with ECDSA, and a complete recovery of the plaintext when the file is used for encryption using SM2-ECIES. The attack is especially devastating against curves with $j$-invariant equal to 0 such as secp k series standardized by SECG [16], for which the recovery reduces to a single division in the base field. Our variant of the SCPD attack injects a fault into the parameter initialization phase of elliptic curves, while the original method by Blömer and Günther targets point decompression algorithm itself. We stress that the present method strengthens the original because ours does not require an expensive double fault to circumvent the point validity check, which is a widely accepted countermeasure against most invalid curve attacks nowadays.
  We also mention that recovering the secret scalar $k$ in our setting is actually slightly more involved, because OpenSSL (and many other cryptographic libraries) relies on a scalar multiplication algorithm that first rewrites the scalar to fix its bit-length (as a countermeasure against Brumley and Tuveri's remote timing attacks [17]), and as a result, the actual scalar used in the algorithm is congruent to $k$ modulo $n$, but not equal as an integer. We describe a simple technique to deal with this situation.
- Additionally, we apply the present fault attack technique to OpenSSL's implementation of ECDH, by combining it with Neves and Tibouchi's degenerate curve attack. The attack in this part targets usual named curve parameters with nonzero $j$-invariant. In an ECDH key exchange, as opposed to the case of ECDSA and SM2-ECIES, the raw result of scalar multiplication on a degenerate curve is usually not available to an adversary; all she could obtain is some *ciphertext* generated with a shared secret key derived from the resulting point, and therefore our variant of the SCPD attack cannot be easily exploited against it. To overcome this limitation, we employ a Pohlig–Hellman-like technique by Neves and Tibouchi. We first modify their ECDH model to properly capture more realistic protocols, and accordingly describe our version of Pohlig–Hellman-like attack. The attack is typically more computationally expensive than the one against signature and encryption schemes, and requires multiple faulty outputs from the server, but can recover the entire static secret key *even in the presence of an EC public key validation function*.
- We experimentally verified that the above attacks reliably work in a practical situation where physical attacks would be of paramount concern; in particular, we make use of O'Flynn's low-cost voltage glitch fault [18] to mount

the attacks on the following three specific command line operations of OpenSSL when executed in widely used Raspberry Pi single board computer [19]:

- ECDSA signature generation with `dgst -sign` command,
- SM2-ECIES encryption with `pkeyutl -encrypt` command, and
- ECDH key exchange with `pkeyutl -derive` command.

To the best of our knowledge, this is the first work that presents experimental results on practically exploitable fault attacks against cryptographic algorithms executed in Raspberry Pi.

*Organization of the paper:* The remainder of the paper is organized as follows. Section II summarizes relevant mathematical facts, cryptographic schemes, and the SCPD attack by Blömer and Günther. Section III presents our fault injection technique against OpenSSL's ECDSA and SM2-ECIES implementation as well as the experimental results of the attack on OpenSSL command line tools installed in Raspberry Pi. Section IV extends the present attack to ECDH key exchange in OpenSSL. We finally give concluding remarks in Section VI.

### D. Related Works

An invalid curve attack against curves in Weierstrass form was first developed by Antipa et al. [12], and the subsequent works extended it to hyperelliptic curves [20], GLS setting [21] and twisted Edwards curves [14]. Fault attack techniques have been occasionally exploited to force an EC scalar multiplication algorithm to operate on weak curves [22], [23], [24], [25], [9] since the seminal work of Biehl et al. [8]

Brumley et al. [26] discovered a software bug attack on TLS-ECDH implementation of OpenSSL. Jager et al. [27] also practically applied invalid curve attacks to several real-world implementations of TLS-ECDH. Valenta et al. [28] recently performed a broad survey of ECC schemes in the wild and confirmed their resistance to well-known invalid curve attacks.

Side-channel leaks from (EC)DSA in OpenSSL have been cryptanalyzed by a number of papers such as [29], [17], [30], [31], [32]. Tuveri et al. [33] recently carried out various types of side-channel analysis against SM2 cipher suite in the pre-release version of OpenSSL 1.1.1, and accordingly proposed the patchset to fix possible vulnerabilities.

Various techniques of fault analysis and their countermeasure are surveyed in [1], [34], and detailed analysis of clock or voltage glitch fault against ARM processor can be found in Korak and Hoefler's work [35]. Several papers such as Barenghi et al. [36] and Timmers et al. [37], [38] addressed fault analysis of general purpose CPU, though none of them were targeting elliptic curve cryptography. O'Flynn [18], [39] demonstrated that it is possible to cause malfunction in a simple for-loop program executed in Raspberry Pi, by injecting a voltage glitch fault from the ChipWhisperer side-channel and glitch attack evaluation board [40].

## II. Preliminaries

### A. Elliptic Curve Defined over Prime Fields

Let $p$ be a prime satisfying $p > 3$. A short Weierstrass form of an elliptic curve defined over $\mathbb{F}_p$ is given by the following affine equation:

$$E/\mathbb{F}_p : y^2 = x^3 + Ax + B$$

where the coefficient $A$ and $B$ are in $\mathbb{F}_p$. The $\mathbb{F}_p$-rational points of $E$, including the point at infinity $O = (0 : 1 : 0)$, form an abelian group under the following operations:

$$\begin{aligned} -P &\coloneqq (x_P, -y_P) \\ P + Q &\coloneqq \left(\lambda^2 - x_P - x_Q, \lambda(x_P - x_{P+Q}) - y_P\right) \\ \lambda &\coloneqq \begin{cases} \frac{y_P - y_Q}{x_P - x_Q} & \text{if } Q \neq \pm P \\ \frac{3x_P^2 + A}{2y_P} & \text{if } Q = P \end{cases} \end{aligned}$$

where $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$ and $P + Q = (x_{P+Q}, y_{P+Q})$, respectively.

Throughout the entire paper, we assume that the standardized prime curves are defined by the following *domain parameters* $\mathcal{D}$:

$$\mathcal{D} := (p, A, B, P, n, c)$$

where $P \in E(\mathbb{F}_p)$ is a base point of prime order $n$, and $c = \#E(\mathbb{F}_p)/n$ is the curve cofactor.

### B. Singular Curve

We now describe one of the degenerate cases of the group low defined in the previous subsection: the case of a *singular curve*. For more comprehensive and general treatment, see standard textbooks, e.g. [41] and [42]. A point on a curve is said to be *singular* if the partial derivatives of the defining equation of $E$ simultaneously vanish at that point. The curve is said to be singular if there exists at least one singular point on it. Now we consider the following cuspidal singular curve $\widetilde{E}$ defined by a short Weierstrass equation with $A = B = 0$:

$$\widetilde{E} : y^2 = x^3$$

where $(0, 0)$ is the only singular point. The following fact plays a crucial role in the SCPD attack in Section II-G and our variant in Section III.

**Theorem 1.** *Let $\mathbb{F}_p^+$ be the additive group of $\mathbb{F}_p$ and $\widetilde{E}(\mathbb{F}_p)$ be the set of nonsingular $\mathbb{F}_p$-rational points on $\widetilde{E}$ including the point at infinity $O = (0 : 1 : 0)$. Then the map $\phi : \widetilde{E}(\mathbb{F}_p) \to \mathbb{F}_p^+$ with*

$$\begin{aligned} (x, y) &\mapsto x/y \\ O &\mapsto 0, \end{aligned}$$

*is a group isomorphism between $\widetilde{E}(\mathbb{F}_p)$ and $\mathbb{F}_p^+$. Its inverse $\phi^{-1} : \mathbb{F}_p^+ \to \widetilde{E}(\mathbb{F}_p)$ is*

$$\begin{aligned} t &\mapsto \left(1/t^2, 1/t^3\right) \\ 0 &\mapsto O. \end{aligned}$$

### C. Supersingular Elliptic Curve

The other degenerate case we consider in this paper is that of *supersingular elliptic curves*, which can be defined as follows:

**Definition 1** (Supersingular curve)**.** Let $E$ be an elliptic curve defined over $\mathbb{F}_p$, where $q$ is a power of the prime $p$. Then $E$ is called *supersingular* when $\#E(\mathbb{F}_q) \equiv 1 \pmod{p}$.

For $q = p$ and $p \geq 5$, that condition is simply equivalent to $\#E(\mathbb{F}_p) = p+1$ by the Hasse bound. Our attack in Section IV relies on the following claim:

**Proposition 1.** *Suppose $E'$ is an elliptic curve defined over $\mathbb{F}_p$ and defined by the equation*

$$E' : y^2 = x^3 + Ax$$

*where $A \neq 0$. If $p \equiv 3 \mod 4$, then $E'$ is supersingular.*

*Proof.* We denote the quadratic residues of $\mathbb{F}_p$ by $\mathcal{QR}$ and the quadratic non-residue by $\mathcal{QNR}$. Since $p \equiv 3 \mod 4$, $-1 = p-1 \in \mathcal{QNR}$. Hence, if $f(x) \coloneqq x^3 + Ax \in \mathcal{QR}$, then $f(-x) = -(x^3 + Ax) = -f(x) \in \mathcal{QNR}$, and vice versa; in other words, if $f(x) \neq 0$ then exactly one of $\{f(x), f(-x)\}$ is in $\mathcal{QR}$. Let $S$ be the set of $x \in \mathbb{F}_p$ such that $f(x) \in \mathcal{QR}$ and $W$ be the set of roots of $f(x)$. Because for any $x \in \mathbb{F}_p \setminus W$ either $x \in S$ or $-x \in S$ holds, we obtain

$$\#\left((\mathbb{F}_p \setminus W) \cap S\right) = \frac{\#(\mathbb{F}_p \setminus W)}{2}.$$

Finally, the cardinality of $\mathbb{F}_p$-rational points of $E'$ including the point at infinity can be counted as follows:

$$\#E'(\mathbb{F}_p) = 2 \times \#\left((\mathbb{F}_p \setminus W) \cap S\right) + \#W + 1 = p + 1$$

which implies $E'$ is supersingular. □

An important property of supersingular curves is the fact that their group of points maps efficiently into a multiplicative group: this observation is the basis of the MOV attack of Menezes–Okamoto–Vanstone [43].

**Proposition 2** (MOV attack)**.** *Let $E$ be a supersingular curve over $\mathbb{F}_p$, $p \geq 5$. Then there exists an injective, efficiently computable group homomorphism $E(\mathbb{F}_p) \to \mathbb{F}_{p^2}^*$ (which can be expressed in terms of the Weil pairing on $E$). In particular, the discrete logarithm problem on $E$ is no harder than the discrete logarithm problem in the multiplicative group $\mathbb{F}_{p^2}^*$.*

### D. ECDSA

Algorithm 1 specifies the signature generation algorithm of ECDSA. We assume that an approved cryptographic hash function $H : \{0,1\}^* \to (\mathbb{Z}/n\mathbb{Z})^*$ is predefined.

### E. SM2-ECIES

SM2 is a cipher suite recommended by Chinese Commercial Cryptography Administration Office [45] and has been recently supported by OpenSSL version 1.1.1 [46]. Its public key encryption algorithm, which we refer to as SM2-ECIES, is essentially a slightly modified version the ECIES ISO standard [5]. Algorithm 2 specifies the encryption algorithm of SM2-ECIES. Here $|M|$ is the bit-length of a message $M$, and $\mathsf{KDF}$ is a key derivation function which derives a shared secret key $K$ satisfying $|K| = |M|$.

**Algorithm 1** ECDSA signature generation [44]

**Input:** $d \in \mathbb{Z}/n\mathbb{Z}$: secret key, $Q = [d]P$: public key, $M \in \{0,1\}^*$: message to be signed, $\mathcal{D}$: domain parameters
**Output:** a valid signature $(r, s)$

1: $k \leftarrow_{\$} (\mathbb{Z}/n\mathbb{Z})^*$
2: $(x_k, y_k) \leftarrow [k]P$
3: $r \leftarrow x_k \mod n$
4: $h \leftarrow H(M)$
5: $s \leftarrow (h + rd)/k \mod n$
6: **return** $(r, s)$

**Algorithm 2** SM2-ECIES encryption [45]

**Input:** $Q \in E(\mathbb{F}_p)$: public key, $M \in \{0,1\}^*$: message to be encrypted, $\mathcal{D}$: domain parameters
**Output:** ciphertext $(C_1, C_2, C_3)$

1: $k \leftarrow_{\$} (\mathbb{Z}/n\mathbb{Z})^*$
2: $C_1 = (x_k, y_k) \leftarrow [k]P$
3: $(x', y') \leftarrow [k]Q$
4: $K \leftarrow \mathsf{KDF}(x'||y', |M|)$
5: $C_2 \leftarrow M \oplus K$
6: $C_3 \leftarrow H(x'||y'||M)$
7: **return** $(C_1, C_2, C_3)$

### F. Point Compression

Let $P = (x, y) \in E(\mathbb{F}_p)$ be a curve point. Since $y = +\sqrt{x^3 + Ax + B}$ or $y = -\sqrt{x^3 + Ax + B}$, one can recover the $y$-coordinate if its sign (i.e. whether $y$ is even or odd in $\mathbb{F}_p$) is stored alongside the $x$-coordinate; this technique is known as *point compression*. In the hexadecimal format of the compressed point, the leftmost octet contains the information of $y$-coordinate: the octet `0x02` (resp. `0x03`) indicates that the $y$ is even (resp. odd). Moreover, `0x04` indicates that the octet string represents an uncompressed point.

For instance, the secp256k1 curve parameter standardized by SECG in [16, §2] has the following base point in an uncompressed 65-byte hexadecimal string:

```
04 79BE667E F9DCBBAC 55A06295 CE870B07
   029BFCDB 2DCE28D9 59F2815B 16F81798
   483ADA77 26A3C465 5DA4FBFC 0E1108A8
   FD17B448 A6855419 9C47D08F FB10D4B8
```

whereas the compressed form of the above is represented as a 33-byte string as follows:

```
02 79BE667E F9DCBBAC 55A06295 CE870B07
   029BFCDB 2DCE28D9 59F2815B 16F81798
```

Note that the information of $y$-coordinate is compressed to the leftmost octet `02` while $x$-coordinate remains the same.

**Algorithm 3** Point Decompression Algorithm [47, §2.3.4]

**Input:** $x \in \mathbb{F}_p$, $\bar{y} \in \{0\text{x}02, 0\text{x}03\}$, $A$, $B$, $p$
**Output:** $P = (x, y)$: uncompressed base point satisfying $y^2 = x^3 + Ax + B \mod p$

1: $y \leftarrow x^2$
2: $y \leftarrow y + A$ ▷ $A = 0$ for secp k series
3: $y \leftarrow y \times x$
4: $y \leftarrow y + B$ ↯
5: $y \leftarrow \sqrt{y}$
6: **if** $\bar{y} = 0\text{x}02$ **then**
7: $b \leftarrow 0$
8: **else**
9: $b \leftarrow 1$
10: **end if**
11: **if** $y \not\equiv b \mod 2$ **then**
12: $y \leftarrow p - y$
13: **end if**
14: **return** $(x, y)$

### G. Singular Curve Point Decompression Attack

We now describe Blömer and Günther's SCPD attack [9] against elliptic curves of short Weierstrass form.

*Attack model:* We suppose that the compressed base point $P = (x, y) \in E(\mathbb{F}_p)$ is stored in a cryptographic device and assume that the scalar multiplication algorithm receives the decompressed base point as input. The fault attacker is able to modify the base point by injecting a suitably synchronized fault upon point decompression algorithm that leads to an incorrectly reconstructed $y$-coodinate of $P$.

*Instruction skipping fault on base point decompression:* Algorithm 3 is the point decompression routine specified by SECG [47, §2.3.4]. If $A = 0$ (as the BN-curve and secp k series have) and a single instruction skipping fault is injected at line 4, then the resulting $y$-coordinate, denoted by $\widetilde{y}$, is incorrectly reconstructed so that the following holds:

$$\widetilde{y}^2 = x^3 \mod p.$$

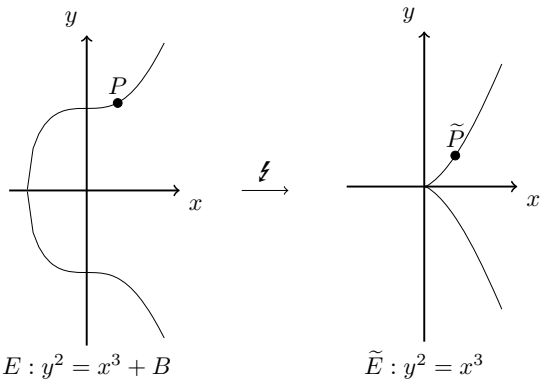
Hence the perturbed faulty base point is reliably on the singular curve $\widetilde{E} : y^2 = x^3$ as depicted in Fig. 1. Let $\widetilde{P} = (x, \widetilde{y})$ be a perturbed base point and $k$ be a secret scalar. Then using the isomorphism $\phi$ in Theorem 1

$$\begin{aligned}\phi([k]\widetilde{P}) &= \phi(\underbrace{\widetilde{P} + \ldots + \widetilde{P}}_{k}) \\ &= \phi(\widetilde{P}) + \ldots + \phi(\widetilde{P}) \\ &= kx/\widetilde{y}.\end{aligned}$$

By applying the inverse $\phi^{-1}$, we obtain

$$\begin{aligned}[k]\widetilde{P} &= \left(\frac{\widetilde{y}^2}{k^2x^2}, \frac{\widetilde{y}^3}{k^3x^3}\right) \\ &= \left(\frac{x}{k^2}, \frac{\widetilde{y}}{k^3}\right).\end{aligned}$$

Fig. 1: Pictorial overview of the Singular Curve Point Decompression Attack when $A = 0$




Hence, assuming that the $x$-coordinate of $[k]\widetilde{P}$, denoted by $\widetilde{x}_k$, is available to an attacker, he can recover the secret scalar $k$ (up to sign) by simply computing a division and a square root modulo $p$:

$$k \equiv \pm\sqrt{\frac{x}{\widetilde{x}_k}} \pmod p.$$

Note, however, that the attack fails if $x^3 + Ax$ is a quadratic non-residue in the base field $\mathbb{F}_p$, because the square root operation at line 5 typically fails in that case (either because the square root algorithm fails on nonquadratic residues, or because the resulting point fails point validation). This implies that, for example, secp192k1 and secp256k1 are susceptible to the SCPD attack, but secp224k1 is not.

## III. Attacking ECDSA and SM2-ECIES in OpenSSL

OpenSSL allows users to generate elliptic curve key files with explicit curve parameter embedded into them (as opposed to the use of a limited set of named curves). The construction of such key file is in fact mentioned in the documentation as a way of achieving backwards compatibility with versions of OpenSSL supporting fewer named curves. Moreover, the curve parameters in such a key file can optionally store the curve base point in *compressed form*.

In this section, we show that the use of such key files can be easily exploited by a fault attacker to mount a variant of the SCPD attack described above. Our variant achieves a full key recovery attack when the key file is used for signing with ECDSA, and a complete recovery of the plaintext when the file is used for encryption using ECIES in OpenSSL. Both attacks were practically validated using concrete fault experiments against a RaspberryPi.

### A. OpenSSL EC Key Files

We first present a concrete situation where OpenSSL generates an EC key pair explicitly containing the domain parameters with a compressed base point. In what follows, we assume

the version 1.1.1 [48], which is the latest release of OpenSSL as of November 2018. Complete command line operations in this part are found in Appendix A Fig. 8. In OpenSSL, EC key operations are mainly dealt with two command line interfaces: `ecparam` and `ec`. While the former is used to generate a secret key, the latter is used to derive an corresponding public key. By default `ecparam` only outputs a secret key and the name of the domain parameter specified by `-name` option. However, the command line tool also allows a user to *explicitly* store the details of parameters into a key file by adding `-param_enc explicit` option. This option is supported mainly for backwards compatibility purposes; for example, not all the target systems know the details of the named curve (such as brainpoolP512t1 for the version below 1.0.2) and a user might want to explicitly pass the full parameter details to others. This use case is in fact described in the official wiki page [49] of OpenSSL. Finally, by adding `-conv_form compressed` option, one can obtain a key file including a compressed base point as part of the parameters. Note that this option also affects the form of a public key point. Fig. 2 displays an example output of the above operations. In order to derive the public key in a compressed form including the same parameter details, one can simply invoke `ec` command on a generated secret key with `-pubout` option. Alternatively, one can derive a key pair of the same form by first creating an EC parameter file, as shown in Fig. 8 of Appendix A.

The signing and encrypting operations using an EC key can be achieved by `dgst` and `pkeyutl`[1], respectively. When these commands are invoked on key files generated as above, OpenSSL's `EC_GROUP_new_from_ecparameters()` function internally constructs the domain parameters $\mathcal{D} = (p, A, B, P, n, c)$ as `EC_GROUP` structure, which essentially works as follows:

(i) Convert the raw byte arrays of $A$, $B$, and $p$ into `BIGNUM` structures, by calling `BN_bin2bn()` utility function.
(ii) Initialize `EC_GROUP` structure with $A, B$ and $p$ as inputs.
(iii) Parse the compressed base point $\bar{P} = (x, \bar{y})$ and convert the raw byte array of $x$-coordinate into a `BIGNUM` structure.
(iv) Call Algorithm 3 on $x, \bar{y}, A, B$, and $p$ to initialize the uncompressed base point $P = (x, y)$.
(v) Perform the validity check of $P$ i.e. check if $y^2 = x^3 + Ax + B \mod p$ holds. Return error if the check fails.
(vi) Convert the raw byte arrays of the group order $n$ and the curve cofactor $c$ into `BIGNUM` structures.
(vii) Store $P = (x, y)$, $n$, and $c$ of `BIGNUM` forms into `EC_GROUP` structure.

Additionally, when the compressed public key $\bar{Q} = (x_Q, \bar{y}_Q)$ is loaded, `o2i_ECPublicKey()` function performs the same operations in (iii) – (v) to initialize `EC_KEY` structure using the domain parameters constructed as above.

[1]We remark that the targeted OpenSSL version in this work does not support the use of SM2 in *command line tools* as is, though the codebase for SM2 has been fully implemented; however, OpenSSL Management Committee plans to add this feature in a post 1.1.1 release [50] and it can be simply achieved by calling `EVP_PKEY_set_alias_type()` right after loading a key file inside `pkeyutl`.

Fig. 2: Generation of EC key files in OpenSSL, including a compressed base point

```
$ openssl ecparam -out sk.pem -name secp256k1\
  -genkey -conv_form compressed -param_enc explicit

$ openssl ec -in sk.pem  -noout -text
read EC key
Private-Key: (256 bit)
priv:
    08:45:c9:52:d8:b9:b3:3b:c3:c5:a2:ef:2d:a9:46:
    32:53:f8:a8:75:68:6d:22:31:b4:d9:fc:de:f5:f3:
    b4:f0
pub:
    02:94:78:28:99:e4:b3:06:53:0d:d3:43:a8:29:12:
    1b:db:5b:72:b0:33:f0:76:88:d9:e8:4e:c5:c6:85:
    66:26:4d
Field Type: prime-field
Prime:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
    ff:fc:2f
A:    0
B:    7 (0x7)
Generator (compressed):
    02:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
    0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
    f8:17:98
Order:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
    36:41:41
Cofactor:  1 (0x1)
```

### B. Our Attack

We now describe our variant of the SCPD attack that can be easily achieved in practice against OpenSSL whenever it loads an EC key file including a base point and a public key in compressed forms. For simplicity, we focus on the attack against curves with $j$-invariant equal to 0 (i.e., $A = 0$), but the attack also generalizes to the curves with nonzero $j$-invariant. Indeed, in that case, the faulty curve becomes supersingular according to Proposition 1, and hence the MOV attack of Proposition 2 applies and reduces the discrete logarithm problem on the curve to a discrete logarithm in the field $\mathbb{F}_{p^2}^*$, which is easy to solve for sizes of $p$ up to at least 256 bits, and tractable even up to 384 bits or so. Note that this supersingular case is not covered by the paper of Blömer and Günther [9], but our attack applies to it nonetheless.

*Attack model:* In our model, the attacker injects a single instruction skipping fault upon the invocation of `BN_bin2bn()` on a parameter $B$ in (i), and can force the resulting `BIGNUM` form of $B$ to be 0, instead of the correct value e.g. $B = 7$ for secp256k1. As a consequence, `EC_GROUP` structure is incorrectly initialized, which causes the point decompression at (iv) to output the invalid uncompressed point $\tilde{P} =$

$(x, \widetilde{y})$ such that $\widetilde{y}^2 = x^3$ holds. Note that the validity check at (v) cannot detect the faulty uncompressed point because the check function also receives the incorrect parameter $B = 0$, and therefore conceives of $\widetilde{P}$ as a "valid" point on the cuspidal singular curve $\widetilde{E} : y^2 = x^3$. Finally, `EC_GROUP_new_from_ecparameters()` returns the following domain parameters:

$$\widetilde{\mathcal{D}} = (p, 0, 0, \widetilde{P}, n, c)$$

When the above faulty domain parameters are used for scalar multiplication of $\widetilde{P}$, we also assume that the attacker has access to the $x$-coordinate of $[k]\widetilde{P}$ (just as the original SCPD attack assumes).

Furthermore, the decompression of public keys is incorrectly performed using the domain parameters $\widetilde{\mathcal{D}}$; accordingly, a faulty uncompressed public key $\widetilde{Q} = (x_Q, \widetilde{y_Q})$ inevitably lies on $\widetilde{E}$ as well.

*Realization of the model:* We now identify the specific instruction in `BN_bin2bn()` which would cause the return value to become 0 when skipped. Fig. 3 shows the source code of `BN_bin2bn()` in OpenSSL 1.1.1. The function takes a raw byte array and its length as inputs, and outputs the corresponding `BIGNUM` object. Whenever `BN_bin2bn()` tries to create the `BIGNUM` object for parameter $B$, it first initializes the return value to 0 by calling `BN_New()` function at line 9. Here we target the conditional branch instruction at line 17; if that line is skipped, `BN_bin2bn()` immediately aborts and returns the `BIGNUM` object containing the value 0.

The high-level description above omits a subtle practical detail; in practice, a fault skips *CPU instructions*, which do not necessarily match a specific line in C code, especially when taking compiler optimizations into account. Hence the target instructions to achieve the desired outcome would vary depending on the actual machine code generated by the compiler. In Appendix B Fig. 10 we present the complete ARM assembly code of `BN_bin2bn()` as generated by the built-in GCC of the Raspberry Pi Linux distribution (with identical compiler options, including full optimization, as specified in the OpenSSL Makefile), and marked possible vulnerable instructions with the comment "`@SKIP!!`". In particular, we can observe that the `bne` instruction at line 35 of Fig. 10 corresponds to line 17 of the original C code in Fig. 3; if it is skipped, the execution proceeds straight up to line 42 of Fig. 10, which corresponds to the `return` at line 19 of Fig. 3. Interestingly, we also found several other instructions (at lines 30, 64, and 77) that would all cause the return value to become 0 when skipped. For space reasons, we omit the specific discussion of these other cases.

*Recovery of the secret scalar $k$:* In OpenSSL's scalar multiplication function `ec_scalar_mul_ladder()`, the scalar $k \in [1, n-1]$ is first rewritten to be $\hat{k} = k + \lambda n$ with $\lambda \in \{1, 2\}$ such that the resulting scalar's bit-length is exactly 1-bit larger than that of the group order $n$, in order to thwart a remote timing attack by Brumley and Tuveri [17].

Fig. 3: `BN_bin2bn()` conversion function from `crypto/bn/bn_lib.c` in OpenSSL 1.1.1 [48]

```
1  BIGNUM *BN_bin2bn(const unsigned char *s, int len,
       BIGNUM *ret)
2  {
3      unsigned int i, m;
4      unsigned int n;
5      BN_ULONG l;
6      BIGNUM *bn = NULL;
7
8      if (ret == NULL)
9          ret = bn = BN_new();
10     if (ret == NULL)
11         return NULL;
12     bn_check_top(ret);
13     /* Skip leading zero's. */
14     for ( ; len > 0 && *s == 0; s++, len--)
15         continue;
16     n = len;
17     if (n == 0) {
18         ret->top = 0;
19         return ret;
20     }
21     i = ((n - 1) / BN_BYTES) + 1;
22     m = ((n - 1) % (BN_BYTES));
23     if (bn_wexpand(ret, (int)i) == NULL) {
24         BN_free(bn);
25         return NULL;
26     }
27     ret->top = i;
28     ret->neg = 0;
29     l = 0;
30     while (n--) {
31         l = (l << 8L) | *(s++);
32         if (m-- == 0) {
33             ret->d[--i]= l;
34             l = 0;
35             m = BN_BYTES - 1;
36         }
37     }
38     bn_correct_top(ret);
39     return ret;
40 }
```

More concretely, the function actually computes $[\hat{k}]P$ instead of $[k]P$, where

$$\hat{k} = \begin{cases} k + 2n & \text{if } \lceil \log(k+n) \rceil = \lceil \log n \rceil \\ k + n & \text{otherwise.} \end{cases}$$

Though $[k]P = [\hat{k}]P$ indeed holds when the valid base point is used, it is not the case anymore when the function takes the invalid base point. Hence, recalling the discussion in Section II-G the fault attacker can recover $\hat{k}$ up to sign from $\widetilde{x}_{\hat{k}}$, where $(\widetilde{x}_{\hat{k}}, \widetilde{y}_{\hat{k}}) = [\hat{k}]\widetilde{P}$, and eventually obtain four candidates of $k$ as follows:

$$k \in \left\{ \pm\sqrt{\frac{x}{\widetilde{x}_{\hat{k}}}} - n \pmod p, \pm\sqrt{\frac{x}{\widetilde{x}_{\hat{k}}}} - 2n \pmod p \right\}.$$

*Recovery of ECDSA secret key:* Once the faulty ECDSA signature pair $(\widetilde{r} = \widetilde{x}_{\hat{k}} \mod n, \widetilde{s} = (h + \widetilde{r}d)/k \mod n)$ is obtained, we first compute candidates of the nonce $k$ as

described above[2]. To complete the attack, it suffices to use the well-known fact that the knowledge of $k$ in an ECDSA signature directly exposes the secret key $d$ as:

$$d = (\widetilde{s}k - h)/\widetilde{r} \mod n.$$

Furthermore, although we have several candidates for the correct $k$, it is easy to find the actual secret key: compute all candidates for $d$, and keep the one that corresponds to the public verification key.

*Recovery of SM2-ECIES plaintext:* In SM2-ECIES, an attacker has access to both coordinates of $[\hat{k}]\widetilde{P} = (\widetilde{x}_{\hat{k}}, \widetilde{y}_{\hat{k}})$ as they are parts of the ciphertext, and can thus determine $\hat{k}$ uniquely:

$$\hat{k} = \frac{\widetilde{y}\widetilde{x}_{\hat{k}}}{x\widetilde{y}_{\hat{k}}}.$$

Using the fact that the public key is also incorrectly decompressed, i.e. $\widetilde{Q} = (x_Q, \widetilde{y_Q})$ satisfies $\widetilde{y_Q}^2 = x_Q^3$, the faulty seed for KDF can be reconstructed as follows:

$$(\widetilde{x}', \widetilde{y}') = [\hat{k}]\widetilde{Q} = \left(\frac{x_Q}{\hat{k}^2}, \frac{\widetilde{y_Q}}{\hat{k}^3}\right).$$

Finally, the derived key $\widetilde{K} = \mathsf{KDF}(\widetilde{x}'||\widetilde{y}', |C_2|)$ can be used to obtain the plaintext $M$ by computing $\widetilde{K} \oplus C_2$.

*Comparison with the original SCPD attack:* If the original SCPD attack described in Section II-G was applied to OpenSSL, it would target the point decompression routine in (iv), and the faulty uncompressed base point $\widetilde{P}$ can be immediately detected by subsequent point validitation, since the program still knows the genuine parameter $B$ at this stage. Accordingly, the original SCPD attack required a *second*, suitably synchronized fault to skip the validity check function as well. Such a double fault attack is quite challenging to achieve in practice, especially on larger devices than the AVR target of Blömer and Günther, due to process scheduling issues and frequent interrupts. On the other hand, since our attack incorrectly reconstructs the *whole domain parameters* at an earlier stage, the validity check function does not know the genuine value of $B$ and eventually executes the assertion of $\widetilde{y}^2 \stackrel{?}{=} x^3$, which of course always passes. As a result, our variant can be realized using a simple, single fault injection. We were able to validate it experimentally at low cost on a large, multiprocess embedded system running a general purpose operating system: namely, the Raspberry Pi running Linux.

An important remark along those lines is that, since the faulty generator obtained in our attack does not have the expected order, the domain parameters on which the computations occur would not pass the full public key validation specified in the SECG standard [47, §3.2.2.1]. Thus, if *full public key validation* was always carried out by OpenSSL upon loading a key file, our attack would also required a double fault. However, although the key validation primitive is indeed implemented in OpenSSL (in library function `EC_KEY_check_key()`), it is not called by default, probably due to its substantial computational cost, and key recovery is thus possible with a single fault.

[2]Though the attacker can only exploit the *residue* $\widetilde{r}$ of the resulting $x$-coordinate modulo $n$, we can ignore the probability that $\widetilde{x}_k > n$ due to the Hasse bound and therefore do not need to distinguish between $\widetilde{r}$ and $\widetilde{x}_k$ in practice.

Fig. 4: Overview of the experimental setup

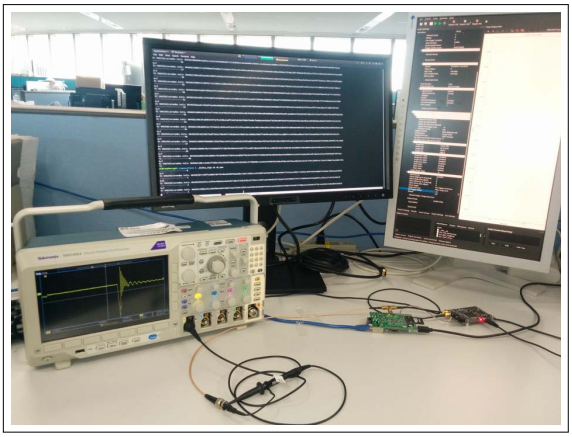
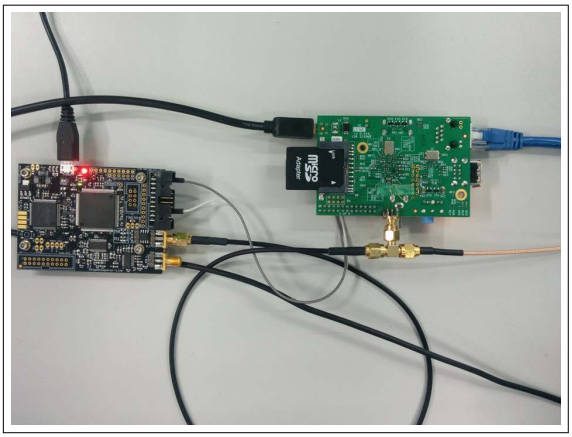


Fig. 5: The ChipWhisperer-Lite evaluation board, connected to Raspberry Pi Model B.



### C. Voltage Glitch Attack Experiment

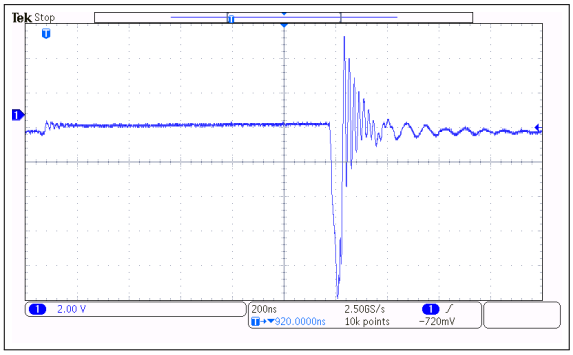
We successfully carried out the above attacks on OpenSSL 1.1.1 installed in Raspberry Pi single board computer [19].

*Experimental setup:* Our device under test, Raspberry Pi Model B, has the following features:

- ARM11-based 32-bit single core processor at 700 MHz clock frequency
- Debian-based Linux OS called Raspbian Stretch
- GCC 6.3.0

The attack was conducted on the ChipWhisperer-Lite side-channel and glitch attack evaluation board [40]. To inject

Fig. 6: Voltage trace of VCC glitch




a voltage glitch from ChipWhisperer into Raspberry Pi, we soldered one side of wire onto the VCC side of a decoupling capacitor, and the other side onto the SMA connector attached to a GND test point of the device. All the command line operations are performed through SSH over the Ethernet connection, as the Ethernet connection usually has good protection against voltage transients. The above setup is suggested in O'Flynn's PhD thesis [39, §8.3.2] and the official ChipWhisperer tutorial [51]. Figs. 4 and 5 show our experimental setup.

We then compiled OpenSSL 1.1.1 using its default Makefile and the built-in GCC toolchain in Raspbian Stretch. The OpenSSL source code was left untouched, except for the addition of the following instructions:

- At the beginning of the `BN_bin2bn()` function: WiringPi [52] library's `digitalWrite()` function, which allows us to transmit a trigger signal to ChipWhisperer through GPIO pins of the Raspberry Pi.
- After `digitalWrite()`: 30 `nop` instructions to further facilitate the synchronization of the injected glitch.
- After the `load_key()` and `load_pubkey()` functions in `pkeyutl.c`: `EVP_PKEY_set_alias_type(pkey, SM2_EVP_PKEY_SM2)` to enable the SM2-ECIES encryption operation by default on the command line tool (as opposed to just a library function).

The first two modifications were applied by first generating the assembly code (i.e. Fig. 10) of the original `crypto/bn/bn_lib.c`, and then adding the corresponding CPU instructions, so that all the other conditions remain the same as in the original code.

*Attack result:* Before mounting the attack, we generated an EC key pair over secp256k1 parameters containing compressed points using the command line options introduced in Section III-A. We then connected to Raspberry Pi through SSH and invoked `dgst` and `pkeyutl` commands to generate ECDSA signature and SM2-ECIES ciphertext, respectively. The ChipWhisperer inserted a single voltage glitch with a high-power MOSFET right after the trigger signal is transmitted to it. After some trial and error, we found that enable-only glitches repeated 127 times at offset 10 clock cycles cause reliably reproducible misbehavior of Raspberry Pi, and we were able to observe that the parameter $B$ was set to 0 with the success probability $\approx 0.1$. Fig. 6 shows a fault waveform inserted into Raspberry Pi. Table I summarizes the experimental results after 1000 trials of fault injection against `dgst` command, where each entry corresponds to the following situations:

TABLE I: Experimental results of voltage glitch fault attacks against `BN_bin2bn()` function running in Raspberry Pi.

| Success | No effect | Program crash | OS crash | Total |
|---|---|---|---|---|
| 95 | 813 | 89 | 3 | 1000 |

- Success: $B$ was successfully set to 0.
- No effect: the command output the valid signature without any error.
- Program crash: the command crashed with some exception e.g. segmentation fault.
- OS crash: Linux OS crashed and completely stopped responding.

As reported by O'Flynn, such voltage glitches rarely crashed the OS and network connection either. Using the faulty outputs of `dgst` and `pkeyutl`, we successfully recovered the ECDSA's secret key and SM2-ECIES's plaintext with the help of SageMath [53].

*Instruction skipping:* Though we were able to observe the desired faulty output, it does not necessarily imply that either of the targeted instructions marked in Fig. 10 was actually *skipped*. To be more specific, we cannot rule out the possibility of other faulty effects such as a double-execution of a certain instruction, as was reportedly achieved by Korak and Hoefler [35] in the attack against ARM Cortex-M0's arithmetical instructions. For example, we can confirm (by manually modifying the assembly code) that a double execution of the `subs` instruction at line 26 would also lead to the 0 return value. In our low-cost setup, it is difficult to reliably synchronize a specific instruction-level glitch against programs running in a non real-time OS like Linux, and we thus remark that our experiment may not perfectly match the attack model described in Section III-B. Nevertheless, the fact that the same goal (of setting the value of $B$ to zero) can be achieved in multiple ways only reinforces the relevance of our fault attack.

*Attacking a fully unmodified library:* One can ask whether it would have been possible to attack a fully unmodified version of OpenSSL using the same approach. We argue that the answer is yes, although at the cost of a significantly more expensive experimental setup (which would be a stretch for typical academic budgets, but not for a less resource-constrained attacker).

More precisely, note that the activation of SM2-ECIES on the command line is simply a matter of convenience: the attack could be mounted without it on code using the OpenSSL

library functions instead of the command line tools (and that change is irrelevant to the ECDSA attack anyway). Therefore, the only meaningful change that we carried out is the addition of a manual GPIO trigger and subsequent `nop` instructions in order to help synchronize the injection of the glitch.

This too can be entirely eliminated using well-known automatic triggering techniques, such as sum-of-absolute-differences (SAD) matching of waveforms acquired from side-channel emanations of the device. The main challenge in applying such techniques to our setting is the high CPU frequency of the Raspberry Pi, which calls for high-resolution capturing equipment and very fast response time triggering hardware. Off-the-shelf solutions exist (e.g. Riscure's icWaves toolkit [54]), but they are far pricier than our $250 experiment. A more advanced triggering technique has recently been demonstrated using moderate resolution side-channel traces, even against ARM-based, high-frequency targets running Linux [55]. The corresponding setup fits better within academic budgets, but requires custom-made hardware and specialized expertise, and hence was somewhat impractical for our purposes.

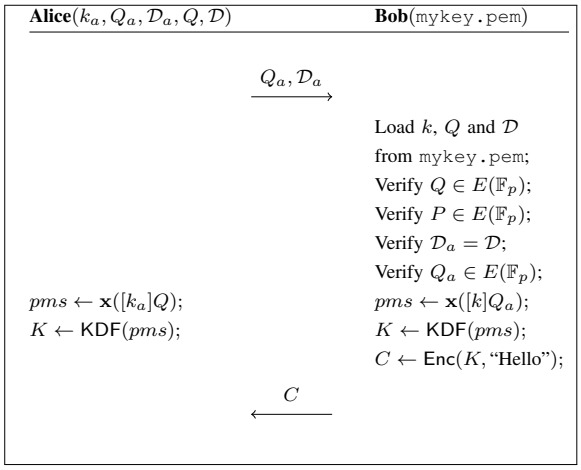
## IV. Attacking ECDH in OpenSSL

In the previous section, we assumed that the result of the scalar multiplication on a degenerate curve is available to an adversary. In an ECDH key exchange, however, this is usually not the case and the adversary can only get some ciphertext generated with a shared secret key derived from the resulting point. Hence, the adversary cannot apply any algebraic operation to the resulting point, unlike the SCPD attack. To overcome this limitation of the SCPD attack, we employ a Pohlig–Hellman-like technique used in traditional invalid curve attacks; specifically, the attack and its target ECDH model in this section are inspired by Neves and Tibouchi's approach [14]. The attack below requires multiple instances of the faulty output by the server, but we show that it can be mounted in moderate time complexity and with only small amount of queries.

### A. Degenerate Fault Attack against Hashed ECDH

*Target ECDH protocol:* We consider the attack against an abstract model of ephemeral-static ECDH key exchange presented in Fig. 7, where Alice is a client holding her ephemeral key pair and Bob is a server who loads his static key pair as well as the domain parameters from locally stored EC key file `mykey.pem`. This protocol is a variant of the one considered in [14], with a few tweaks to increase its practical relevance: our variant does carry out point validation on the server side, and the session key is derived from the $x$-coordinate of the common point computed by the two parties (as opposed to the whole point), as is common in all practical elliptic curve based key exchange protocols, including the TLS handshake with elliptic curves. In Section IV-B, we show how this abstract protocol can be concretely realized using the OpenSSL command line tools, and attacked accordingly using our fault injection techniques.

Fig. 7: EC Diffie–Hellman Protocol with hashed server output, where $k_a$ and $k$ are secret keys chosen from $\mathbb{Z}/n\mathbb{Z}$, $Q_a = [k_a]P$ and $Q = [k]P$ are public keys, $\mathcal{D}_a$ and $\mathcal{D}$ are domain parameters, and $\mathbf{x}(P)$ denotes the $x$-coordinate of a point $P$.




*Target curve parameters:* Our attack target is the server whose domain parameters satisfy $A \neq 0$ and $p \equiv 3 \mod 4$, which hold for many standardized curves like secp r series (except secp224r1) [16] and Brainpool curves [56]. We also assume that Bob's EC key file contains the base point in a compressed form, just as we did in the previous section. These assumptions will allow us to exploit non-prime order of the supersingular curve introduced in Section II-C if the initialization of parameter $B$ is incorrectly done.

*Overview of the attack:* We first describe a high-level overview of our attack against hashed ECDH. Here we assume that Alice is an adversary and tries to steal Bob's secret key $k$ by interacting with him $N$ times. The basic strategy of Alice is to perform a combined attack which uses both the degenerate curve attack of Neves and Tibouchi and the fault injection technique presented in the previous section.

The degenerate case we fundamentally rely upon is a supersingular curve $E' : y^2 = x^3 + Ax$ that has non-prime order $p + 1$ from Proposition 1. Following Neves and Tibouchi's approach, for each query $i$, Alice sends an invalid public key $Q_i \in E'(\mathbb{F}_p)$ of small order $\ell_i$, where $\ell_i$ is a prime (power) factor of $p + 1$, and carries out an exhaustive search in the subgroup $\langle Q_i \rangle$ to find $k \mod \ell_i$ upon receiving an invalid ciphertext $C_i = \mathsf{Enc}(\mathsf{KDF}(\mathbf{x}([k]Q_i)), \text{“Hello”})$ from Bob. After sufficiently many queries Alice computes $k \mod L$ using the Chinese Remainder Theorem (CRT), with $L = \prod_i \ell_i$, and can finally recover the whole $k$ using Pollard's kangaroo (or lambda) algorithm [57] in $O(\sqrt{(p+1)/L})$ time complexity.

However, Neves and Tibouchi only considered careless protocols where point validation on the server side is absent; in our targeted protocol of Fig. 7, Bob performs a number of validity checks upon loading his own key file and receiving

Alice's public key and domain parameters. Hence, we circumvent all these checks via a single instruction skipping fault described in Section III-B, whenever Bob loads his key file. The reader should note that it is crucial to inject a fault *during the initialization phase of parameter* $B$ (i.e. execution of `BN_bin2bn()` function to load $B$ in OpenSSL), not against the validity check functions themselves; otherwise the attacker would need to inject multiple faults to skip them.

Moreover, Neves and Tibouchi's hashed ECDH model fails to capture the significant property: in most ECDH implementations, a shared secret key $K$ is not derived from the resulting curve point $[k]Q_a$ itself, but from its *x-coordinate*, which is often referred to as *premaster secret (pms)*. In this more realistic, but restricted setting, Alice's exhaustive search can only determine $k \mod \ell_i$ *up to sign* for each query, and she thus would have to perform the subsequent Pollard's kangaroo algorithm on exponentially many instances. We avoid this problem by making a single additional query that can be used for checking the correctness of CRT's outputs.

*Attack algorithm and analysis:* We now give a complete description of the attack in Algorithm 4. Again, we stress that a single fault injection in 2) can circumvent all the validity checks of domain parameters and public keys.

**Theorem 2.** *The time complexity of Algorithm 4 is* $O(\sqrt{\ell_0} + N\ell_1 + 2^N \log^2 L)$ *and the space complexity is* $O(1)$.

*Proof.* For each small factor $\ell_i$ the exhaustive search takes $O(\ell_i)$ steps and the total time complexity in 2) is $O(N\ell_1)$. Executing the CRT for each $(k_1, \ldots, k_N)$ takes $O(\log^2 L)$ time, and there are in total $2^N$ patterns of an input. Hence the total time complexity of 4) is $O(2^N \log^2 L)$. Finally Pollard's kangaroo takes $O(\sqrt{\ell_0})$ time. All the subroutines in the above attack are constant space algorithms. □

*Query optimization and complexity estimates:* In Algorithm 4, we simply sorted the prime (power) factors of $p+1$ in descending order and automatically assigned one query to each factor; however, this often yields nonoptimal total time complexity. We present a straightforward approach for query optimization by describing the concrete attack against prime192v1 as an example. We also give the result of optimal complexity estimates for other curve parameters in Table II.

First of all, prime192v1's $p+1$ can be factored as follows:

$$\begin{aligned} p+1 =& 2^{64} \times 67280421310721 \times 6700417 \times 274177 \\ & \times 65537 \times 641 \times 257 \times 17 \times 5 \times 3 \end{aligned}$$

Since $N = 10$ and the largest factors are $\ell_0 = 2^{64}$ and $\ell_1 = 67280421310721 \approx 2^{46}$, the total time complexity would be $O(2^{46})$ without any query optimization.

However, in an actual attack, we can "merge and divide" some queries to equalize the time complexities for each exhaustive search and Pollard's kangaroo. On the one hand, the exhaustive searches for order 3, 5, 17, and 257 subgroups are computationally cheap, and therefore they can be "merged" into other queries, so that the new queries send invalid points of slightly larger composite order to Bob, with which exhaustive search is still feasible. On the other hand, the search in order $2^{63}$ subgroup[3] can be actually "divided" into multiple exhaustive searches in a much smaller subgroup, e.g. order $\rho := 2^{21}$ subgroup; concretely, we first pick order $\rho^3$ point $\widetilde{Q}_{\rho^3}$ as well as $\widetilde{Q}_{\rho^2} = [\rho]\widetilde{Q}_{\rho^3}$ and $\widetilde{Q}_{\rho} = [\rho]\widetilde{Q}_{\rho^2}$, and also rewrite $k$ as follows:

$$k = \bar{\kappa}\rho^3 + \kappa_2\rho^2 + \kappa_1\rho + \kappa_0$$

**Algorithm 4** Attack on hashed ECDH with point validation

**Input:** Bob's public key $Q$ and domain parameters $\mathcal{D}$
**Output:** Secret key $k$ such that $Q = [k]P$

1) Parse the domain parameters $\mathcal{D} = (p, A, B, P, n, c)$ of Bob's public key, and construct invalid domain parameters $\widetilde{\mathcal{D}_a} = (p, A, 0, \widetilde{P}, n, c)$ which defines
   - supersingular curve $E' : y^2 = x^3 + Ax$ of order $p+1 = \prod_{i=0}^{N} \ell_i$ where $\ell_i$ is a prime (power) factor of $p+1$ and $\ell_0 > \ell_1 > \ldots > \ell_N$, and
   - invalid base point $\widetilde{P}$ on $E'(\mathbb{F}_p)$ that shares its $x$-coordinate with the valid base point $P$.
2) For each small factor $\ell_i$ in $\{\ell_1, \ldots, \ell_N\}$:
   i Pick an invalid point $\widetilde{Q_i} \in E'(\mathbb{F}_p)$ of order $\ell_i$ and send $\widetilde{Q_i}$ together with $\widetilde{\mathcal{D}_a}$ to Bob.
   ii Upon Bob loading his EC key file, inject a single instruction skipping fault to set his parameter $B$ to 0, and consequently force the base point to be decompressed to $\widetilde{P} \in E'(\mathbb{F}_p)$ (See Section III-B).
   iii Upon receiving a faulty ciphertext $\widetilde{C_i}$ from Bob, perform an exhaustive search in the small subgroup $\langle \widetilde{Q_i} \rangle$ to find $k_i \in [0, \ell_i)$ such that
   $$\mathsf{Enc}(\mathsf{KDF}(\mathbf{x}([k_i]\widetilde{Q_i})), \text{"Hello"}) = \widetilde{C_i}.$$
   Note that this procedure only finds $k \mod \ell_i$ up to sign i.e. Alice always obtains two solutions $\pm k \mod \ell_i$.
3) Pick the additional invalid point $\widetilde{Q_L} \in E'(\mathbb{F}_p)$ of order $L = \prod_{i=1}^{N} \ell_i$ , send $\widetilde{Q_L}$ together with $\widetilde{\mathcal{D}_a}$ to Bob, and receive the corresponding ciphertext $\widetilde{C_L}$.
4) For each candidate combination of $(k_1, \ldots, k_N) \in \{k \mod \ell_1, -k \mod \ell_1\} \times \ldots \times \{k \mod \ell_N, -k \mod \ell_N\}$:
   i Compute the following using the CRT:
   $$(k_1, \ldots, k_N) \mapsto k' \in \mathbb{Z}/L\mathbb{Z}.$$
   ii Verify the correctness of CRT's output by performing the following check:
   $$\mathsf{Enc}(\mathsf{KDF}(\mathbf{x}([k']\widetilde{Q_L})), \text{"Hello"}) \stackrel{?}{=} \widetilde{C_L}.$$
   If $k'$ passes the above check, it means that $k'$ satisfies either $k' \equiv k \mod L$ or $k' \equiv -k \mod L$, of which the former is derived from the correct candidate combination $(k_1, \ldots, k_N) = (k \mod \ell_1, \ldots, k \mod \ell_N)$.
5) At this stage, Alice already knows two candidates of $k'$, and one of them satisfies $k = k' + Lk''$ for some $k'' < \lfloor n/L \rfloor \approx (p+1)/L = \ell_1$. If Pollard's kangaroo algorithm with inputs $Q - [k']P$ and the base $[L]P$ can find a solution $k''$, then finally outputs $k' + Lk''$.

[3]We lose 1-bit in this search because prime192v1 has no $2^{64}$ order subgroup due to the quadratic non-residue curve parameter $A$.

where $\kappa_i < \rho$ and $\bar{\kappa} < \lfloor n/\rho^3 \rfloor$. The first query uses $\widetilde{Q}_\rho$ and performs exhaustive search in $\langle \widetilde{Q}_\rho \rangle$ to recover $\kappa_0 \equiv k \mod \rho$ up to sign; the second query uses $\widetilde{Q}_{\rho^2}$ to recover $\kappa_1$, which still reduces to the exhaustive search in $\langle \widetilde{Q}_\rho \rangle$ because $[k]\widetilde{Q}_{\rho^2} = [\kappa_1\rho]\widetilde{Q}_{\rho^2} + [\kappa_0]\widetilde{Q}_{\rho^2} = [\kappa_1]\widetilde{Q}_\rho + [\kappa_0]\widetilde{Q}_{\rho^2}$ and the latter term is already known. Note that after the second query we can trivially rule out the wrong candidate of $\kappa_0$ because the exhaustive search would find no solution in that case. Likewise we can recover $\kappa_2$ by using $\widetilde{Q}_{\rho^3}$ in the third query. As a consequence three iterations of exhaustive search in $\langle \widetilde{Q}_\rho \rangle$ are sufficient to find $\kappa_2\rho^2 + \kappa_1\rho + \kappa_0 = k \mod \rho^3$ (up to sign).

To achieve these query optimizations, we can, for example, reconstruct the factors of $p+1$ as follows:

$$\begin{aligned}
\ell'_0 &= 67280421310721 \times 2 \\
\ell'_1 &= 6700417, \quad \ell'_2 = 17 \times 257 \times 641 \\
\ell'_3 &= 3 \times 5 \times 65537, \quad \ell'_4 = 274177 \\
\ell'_5 &= \rho^3 = 2^{63}
\end{aligned}$$

which leads to 7 queries that send invalid points $\widetilde{Q}_1, \ldots, \widetilde{Q}_4, \widetilde{Q}_\rho, \widetilde{Q}_{\rho^2}$, and $\widetilde{Q}_{\rho^3}$ of corresponding orders (+ 1 additional query with the point $\widetilde{Q}_{L'}$ of order $L' = \prod_{i=1}^{5} \ell'_i$). Because $\ell'_0 \approx 2^{47}$, $\ell'_1 \approx 2^{23}$ and $L' \approx 2^{145}$, we obtain the total time complexity of $O(\sqrt{\ell'_0} + 7\ell'_1 + 2^5 \log^2 L') = O(2^{25.5})$, which is much less than the nonoptimal queries.

### B. Application to OpenSSL

*Attack on manual ECDH key exchange in OpenSSL:* We now describe a practical scenario that realizes the ECDH protocol of Fig. 7 using OpenSSL command line tools. Complete command line operations in this part can be found in Appendix A Fig. 9. Suppose Bob holds a static EC key containing explicit curve parameters as well as the compressed base point, which we described in Section III-A. When Bob manually performs the ECDH key exchange with Alice, he may use `pkeyutl -derive` command with Alice's public key file and his own key file as inputs, so that he can obtain the premaster secret $\mathbf{x}([k]Q_a)$. Accordingly, Bob can generate the master secret key $K$ and ciphertext $C$ for "Hello" message using an appropriate cryptographic hash function (e.g. SHA256) as KDF and symmetric encryption algorithm (e.g. AES-256-CBC), respectively; of course, these can be achieved via basic OpenSSL commands.

Here Alice's attack strategy is quite simple; since `pkeyutl -derive` invokes `BN_bin2bn()` function when loading an EC key file, she can directly apply the fault attack in Section III-B to force Bob's curve parameter $B$ to have 0, whenever Bob tries to derive a premaster secret using Alice's malicious public key.

*Experimental results:* We successfully mounted the above attack on OpenSSL 1.1.1 installed in Raspberry Pi. We targeted prime192v1 as Bob's curve parameters and first found an order $(p+1)/2$ point at $x = 260$ on supersingular curve $E' : y^2 = x^3 - 3x$; then we prepared 8 public keys which contain invalid domain parameter $B = 0$ and low order points $\widetilde{Q}_1, \ldots, \widetilde{Q}_4, \widetilde{Q}_\rho, \widetilde{Q}_{\rho^2}, \widetilde{Q}_{\rho^3}$, and $\widetilde{Q}_{L'}$. Upon loading these public keys during the execution of `pkeyutl -derive`, a single voltage glitch fault was inserted into `BN_bin2bn()` function, just as we did in Section III-C. When the fault successfully caused the domain parameters to have $B = 0$, all the subsequent validity checks passed and the command output a faulty $pms$ without raising any error; otherwise, the command aborted immediately with error messages. Then we invoked `dgst` command to hash a faulty $pms$, and to derive a master key $K$; finally we used it to encrypt "hello" message with `enc` command.

After collecting 8 faulty ciphertexts returned from the above operations, we performed the exhaustive search, CRT with the correctness check, and Pollard's kangaroo algorithm described in Algorithm 4 with the help of SageMath [53]. The exhaustive searches for 7 instances took an hour and computing the CRT for $2^5$ candidates of $k'$ was done in few seconds, using a standard desktop computer equipped with Intel Core i5-4460 CPU and Ubuntu 18.04. Finally, we executed Pollard's kangaroo on two candidates of $k' \equiv k \mod L'$, which took 90 minutes, and successfully found the correct secret key $k$.

### C. Applicability to TLS

Although that option is probably not commonly used in practice, the TLS standard does support certificates for elliptic curve cryptography using explicit curve parameters, as well as point compression for the group generator as part of those parameters (see [58, §5.4], particularly the definition of the `ECPoint` structure). It is therefore natural to ask to what extent the attack described in this section applies to the TLS handshake when using elliptic curve cryptography cipher suites.

Note first that one has to posit a rather powerful attacker against the TLS server, than can initiate adversarial TLS handshakes while at the same time injecting faults on the server itself. This can correspond to settings in which the server is an embedded device, in which faults can be injected remotely (e.g., using exploits such as Rowhammer.js [59]).

Even so, the abstract model described in Fig. 7 does not quite map to any of the ECDH-derived key exchange protocols defined in TLS as summarized, e.g., in [58, §2]. However, one can modify the attack to match the "fixed ECDH" setting (i.e., the case of a static server key), albeit at the cost of significant increase in query complexity. Moreover, although the attack does in principle require a static key on the server side, it can interestingly also apply to certain practical implementations of "ephemeral ECDH".

*Fixed ECDH:* The TLS-ECDH setting closest to the one we consider is "fixed ECDH", where the server uses a static key and there is no client authentication. In that case, the TLS handshake takes a form very close to Fig. 7, but with one crucial difference: namely, the client has to send its `ClientFinished` message encrypted under the key obtained from the key exchange (which is already known since the server's public key is static). This is a problem for our

TABLE II: Complexity estimates of the attack against ECDH over standardized curves

| Curve | $p$ | $\lvert\ell'_0\rvert$ | $\lvert\ell'_1\rvert$ | Time | # Queries | $\sqrt{x^3+Ax} \stackrel{?}{\in} \mathbb{F}_p^*$ |
|---|---|---|---|---|---|---|
| prime192v1 | $2^{192}-2^{64}-1$ | 47 | 23 | $O(2^{25.5})$ | 8 | Yes |
| prime256v1 | $2^{224}(2^{32}-1)+2^{192}+2^{96}-1$ | 94 | 46 | $O(2^{48.3})$ | 6 | Yes |
| secp384r1 | $2^{384}-2^{128}-2^{96}+2^{32}-1$ | 188 | 36 | $O(2^{93.5})$ | 7 | Yes |
| secp521r1 | $2^{521}-1$ | 1 | 1 | $O(1)$ | 521 | No |
| brainpoolP192r1 | — | 63 | 49 | $O(2^{50.0})$ | 4 | Yes |
| brainpoolP224r1 | — | 140 | 50 | $O(2^{69.7})$ | 3 | No |
| brainpoolP256r1 | — | 164 | 68 | $O(2^{81.7})$ | 3 | Yes |
| brainpoolP384r1 | — | 181 | 88 | $O(2^{90.1})$ | 4 | Yes |
| brainpoolP512r1 | — | 151 | 125 | $O(2^{126.3})$ | 4 | Yes |

attack, because although the client does know the valid public key of the server, he does not know the perturbed key on the fauly curve $E'$, and therefore cannot easily encrypt its `ClientFinished` message.

There is a simple workaround to this limitation, however: since the derived keys tried by the client are obtained from points of small orders $\ell_1, \ldots, \ell_N$ on $E'$, the client can simply repeat its handshake attempts with key candidates derived from all the possible multiples of those points of small orders, until the server is able to decrypt the `ClientFinished` message and replies. That case corresponds to a correct guess, and therefore reveals the secret key of the server modulo $\ell_1, \ldots, \ell_N$ as before. This is essentially the approach taken in the original invalid curve attacks by Antipa et al. [12]

The time complexity of that variant of the attack is the same as for the attack in the previous section: the only difference is that the exhaustive search phase (Step 2iii of Algorithm 4) is now carried out using online queries to the server instead of offline computations (at a significant, but constant, overhead).[4] The *query* complexity, on the other hand, does obviously increase sharply.

*Ephemeral ECDH:* Clearly, our attack on ECDH requires several faulty executions of the key exchange protocol to recover a secret, and therefore should not apply to "ephemeral ECDH" key exchange (`ECDHE_RSA` and `ECDHE_ECDSA` in TLS), where the server is supposed to use a fresh secret for every session in order to ensure perfect forward secrecy. In practice, however, many TLS servers reuse those ephemeral keys for a long period of time in order to reduce the computational overhead of new encrypted connections: Springall et al. [60] found that, as of 2016, around 15% of the ECDHE domains in the Alexa Top Million practiced some form of key reuse, some of them for months at a time! This type of key reuse is even the default behavior of some popular TLS implementations (e.g., the bug report to fix this issue in NSS, submitted in 2015, appears to remain open at the time of this writing [61]).

In such a setting, the same attack as above applies directly, and recovers the supposedly-ephemeral-but-actually-reused secret of the server (and hence allows to decrypt all TLS sessions the adversary can record until the subsequent key update).

[4]Similarly, note that the check done in Step 4ii of Algorithm 4 can be carried out using $2^N$ handshake attempts, which is negligible.

## V. Countermeasures

The vulnerabilities we pointed out stem from the fact that OpenSSL command line tools support loading a compressed base point from external EC key files. Hence, as a straightforward countermeasure we suggest that the `ecparam` command line interface deprecate `-conv_form compressed` option, so that the generation of such vulnerable EC key files will never occur. More generic, low-level countermeasures against fault attacks can be found in e.g. [34, §5]. We also mention general advice on the use of compressed curve points when one implements ECC:

- To thwart the kinds of attacks described in Section III and Section IV, one should *never* store the base point in compressed form.
- On the other hand, the use of *public key points* in compressed form per se is not a problem; in fact, it is *advisable* to use them because this assures the resulting point is on the curve, so that one can prevent other invalid curve attacks.

## VI. Concluding Remarks

This paper brought the SCPD attack and the degenerate curve attacks closer to practice, and identified fault attack vulnerabilities in OpenSSL's implementation of ECDSA, SM2-ECIES and ECDH. We stress that the attacks on the first two schemes over secp k series (except 224k1) are particularly devastating because the adversary would be able to recover a secret key (resp. plaintext) from a single faulty signature (resp. ciphertext) with almost no computational cost. Note that while we have not directly witnessed the use of such unusual EC keys as Fig. 2 in the wild, there are reasons to believe that they could exist (beyond Heninger's conjecture that "given samples from enough cryptographic implementations, any outrageous vulnerability is likely to be present" [62]). Indeed, the construction of these key files is explicitly described in the OpenSSL official documentation (see III-A) and their use is permitted by the original elliptic curve extensions to TLS [58].

Beyond OpenSSL, we point out that there is a possibility that other cryptographic libraries, especially the ones for embedded systems, are using base points in compressed forms. Such values do appear in SECG's recommended elliptic curve domain parameters [16], and there is a plausible reason why practitioners might want to use them; since reducing code size

is a major concern for embedded implementations, compressing the base point, which results in a code size reduction of 24–64 bytes, can potentially justify the use of compressed base points when program memory is at a premium[5]. Though we fortunately confirmed that the several well-known libraries such as mbed TLS [64], wolfSSL [65], Crypto++ [66] and libsecp256k1 [67] do not implement compressed base points, some implementations for non-production use do include them in reality; for instance, the base point of secp256k1 in a compressed form appears in [68, Appendix D.2]. In particular, the ECDSA over secp256k1 curve, on which we mounted the attack in Section III-C, is nowadays a high-profile target owing to its use in the Bitcoin protocol [69]. Therefore, future research should consider the potential effect of the SCPD attack on hardware Bitcoin wallets, including in-house implementations. All in all, the lesson of this paper is quite simple: *do not store your base points in compressed form!* It would also seem advisable to always avoid the use of explicit curve parameters in ECC implementations and only rely on a reasonable set of named curves.

## Acknowledgement

Akira Takahashi was supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC), and the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC). We thank anonymous reviewers for valuable comments and suggestions.

## References

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[2] A. Takahashi, "A Study on Attacks against Nonces in Schnorr-like Signatures," Master's thesis, Kyoto University, 2018, https://akiratk0355.github.io/file/thesis-master-takahashi.pdf.

[3] P. Gallagher, *Digital Signature Standard (DSS)*, NIST, 2013, fIPS PUB 186–4.

[4] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[5] V. Shoup, "A Proposal for an ISO Standard for Public Key Encryption," Cryptology ePrint Archive, Report 2001/112, 2001.

[6] OpenSSL Management Committee, "OpenSSL: Cryptography and SSL/TLS Toolkit," https://www.openssl.org/.

[7] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)," in *EUROCRYPT '97*, ser. LNCS, vol. 1233. Springer, 1997, pp. 37–51.

[8] I. Biehl, B. Meyer, and V. Müller, "Differential Fault Attacks on Elliptic Curve Cryptosystems," in *CRYPTO 2000*, ser. LNCS, vol. 1880. Springer, 2000, pp. 131–146.

[9] J. Blömer and P. Günther, "Singular Curve Point Decompression Attack," in *FDTC 2015*. IEEE, 2015, pp. 71–84.

[10] D. Boneh, B. Lynn, and H. Shacham, "Short Signatures from the Weil Pairing," *Journal of Cryptology*, vol. 17, no. 4, pp. 297–319, 2004.

[11] P. S. L. M. Barreto and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order," in *SAC 2005*, ser. LNCS, vol. 3897. Springer, 2005, pp. 319–331.

[12] A. Antipa, D. R. L. Brown, A. Menezes, R. Struik, and S. A. Vanstone, "Validation of Elliptic Curve Public Keys," in *PKC 2003*, ser. LNCS, vol. 2567. Springer, 2003, pp. 211–223.

[5]The justification of this scenario closely resembles the discussion in [63, §3.2]. We note that it is significantly more plausible in our setting of an ECDSA implementation than in an implementation of pairing-based cryptography.

[13] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, "Twisted Edwards Curves," in *AFRICACRYPT 2008*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jun. 2008, pp. 389–405.

[14] S. Neves and M. Tibouchi, "Degenerate curve attacks: Extending invalid curve attacks to Edwards curves and other models," *IET Information Security*, vol. 12, no. 3, pp. 217–225, 2018.

[15] S. C. Pohlig and M. E. Hellman, "An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance," *IEEE Transactions on Information Theory*, vol. 24, no. 1, pp. 106–110, Jan. 1978.

[16] *SEC 2: Recommended Elliptic Curve Domain Parameters*, Standards for Efficient Cryptography Group (SECG), 2010, version 2.0.

[17] B. B. Brumley and N. Tuveri, "Remote Timing Attacks Are Still Practical," in *ESORICS 2011*, ser. LNCS. Springer, Berlin, Heidelberg, Sep. 2011, pp. 355–371.

[18] C. O'Flynn, "Fault Injection using Crowbars on Embedded Systems," Cryptology ePrint Archive, Report 2016/810, 2016.

[19] Raspberry Pi Foundation, "Raspberry Pi: A small and affordable computer that you can use to learn programming," https://www.raspberrypi.org/.

[20] K. Karabina and B. Ustaoglu, "Invalid-curve attacks on (hyper)elliptic curve cryptosystems," *Advances in Mathematics of Communications*, vol. 4, no. 3, pp. 307–321, 2010.

[21] T. Kim and M. Tibouchi, "Invalid Curve Attacks in a GLS Setting," in *IWSEC 2015*, ser. LNCS, vol. 9241. Springer, 2015, pp. 41–55.

[22] M. Ciet and M. Joye, "Elliptic Curve Cryptosystems in the Presence of Permanent and Transient Faults," *Designs, Codes and Cryptography*, vol. 36, no. 1, pp. 33–43, Jul. 2005.

[23] J. Blömer, M. Otto, and J.-P. Seifert, "Sign Change Fault Attacks on Elliptic Curve Cryptosystems," in *FDTC 2006*, ser. LNCS, L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, Eds., vol. 4236. Springer Berlin Heidelberg, 2006, pp. 36–52.

[24] P.-A. Fouque, R. Lercier, D. Réal, and F. Valette, "Fault attack on elliptic curve Montgomery ladder implementation," in *FDTC 2008*, L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J.-P. Seifert, Eds. IEEE, 2008, pp. 92–98.

[25] T. Kim and M. Tibouchi, "Bit-Flip Faults on Elliptic Curve Base Fields, Revisited," in *ACNS 2014*, ser. LNCS, vol. 8479. Springer, 2014, pp. 163–180.

[26] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical Realisation and Elimination of an ECC-Related Software Bug Attack," in *CT-RSA 2012*, ser. LNCS, O. Dunkelman, Ed., vol. 7178. Springer Berlin Heidelberg, 2012, pp. 171–186.

[27] T. Jager, J. Schwenk, and J. Somorovsky, "Practical Invalid Curve Attacks on TLS-ECDH," in *ESORICS 2015*, ser. LNCS, vol. 9326. Springer, 2015, pp. 407–425.

[28] L. Valenta, N. Sullivan, A. Sanso, and N. Heninger, "In Search of CurveSwap: Measuring Elliptic Curve Implementations in the Wild," in *Euro S&P 2018*. IEEE, Apr. 2018, pp. 384–398.

[29] B. B. Brumley and R. M. Hakala, "Cache-Timing Template Attacks," in *ASIACRYPT 2009*, ser. LNCS. Springer, Berlin, Heidelberg, Dec. 2009, pp. 667–684.

[30] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way," in *CHES 2014*, ser. LNCS, vol. 8731. Springer, 2014, pp. 75–92.

[31] J. van de Pol, N. P. Smart, and Y. Yarom, "Just a Little Bit More," in *CT-RSA 2015*, ser. LNCS, K. Nyberg, Ed., vol. 9048. Springer International Publishing, 2015, pp. 3–21.

[32] C. Pereida García, B. B. Brumley, and Y. Yarom, ""Make Sure DSA Signing Exponentiations Really Are Constant-Time"," in *CCS 2016*. New York, NY, USA: ACM, 2016, pp. 1639–1650.

[33] N. Tuveri, S. ul Hassan, C. P. Garcia, and B. B. Brumley, "Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study," in *ACSAC 2018*. New York, NY, USA: ACM, 2018, pp. 147–160.

[34] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov. 2012.

[35] T. Korak and M. Hoefler, "On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms," in *FDTC 2014*. IEEE, Sep. 2014, pp. 8–17.

[36] 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*, vol. 86, no. 7, pp. 1864–1878, Jul. 2013.
[37] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM Using Fault Injection," in *FDTC 2016*. IEEE, Aug. 2016, pp. 25–35.
[38] N. Timmers and C. Mune, "Escalating Privileges in Linux Using Voltage Fault Injection," in *FDTC 2017*. IEEE, Sep. 2017, pp. 1–8.
[39] C. O'Flynn, "A Framework for Embedded Hardware Security Analysis," Ph.D. dissertation, Dalhousie University, 2017.
[40] C. O'Flynn and Z. D. Chen, "ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research," in *COSADE 2014*, ser. LNCS, E. Prouff, Ed., vol. 8622. Springer, 2014, pp. 243–260.
[41] J. H. Silverman, *The Arithmetic of Elliptic Curves*, 2nd ed., ser. Graduate Texts in Mathematics. Springer-Verlag New York, 2009.
[42] L. C. Washington, *Elliptic Curves: Number Theory and Cryptography*, 2nd ed. Chapman & Hall/CRC, 2008.
[43] A. Menezes, T. Okamoto, and S. A. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field," *IEEE Transactions on Information Theory*, vol. 39, no. 5, pp. 1639–1646, 1993.
[44] D. Johnson, A. Menezes, and S. A. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.
[45] S. Shen and X. Lee, *SM2 Digital Signature Algorithm*, IETF, 2014, draft-shen-sm2-ecdsa-02.
[46] "OpenSSL 1.1.1 series release notes," https://www.openssl.org/news/openssl-1.1.1-notes.html, 2018.
[47] *SEC 1: Elliptic Curve Cryptography*, Standards for Efficient Cryptography Group (SECG), 2009, version 2.0.
[48] "OpenSSL version 1.1.1," https://github.com/openssl/openssl/tree/OpenSSL_1_1_1, 2018.
[49] "OpenSSL wiki: Command line elliptic curve operations," https://wiki.openssl.org/index.php/Command_Line_Elliptic_Curve_Operations, accessed on November 1st, 2018.
[50] "OpenSSL issue #6719," https://github.com/openssl/openssl/issues/6719, 2018.
[51] "ChipWhisperer Tutorial A3 VCC Glitch Attacks," https://wiki.newae.com/Tutorial_A3_VCC_Glitch_Attacks#Glitching_More_Advanced_Targets:_Raspberry_Pi, accessed on November 1st, 2018.
[52] G. Henderson, "Wiring Pi: GPIO interface library for the Raspberry Pi," http://wiringpi.com/.
[53] The Sage Developers, "SageMath, the Sage Mathematics Software System (Version 7.5.1)," 2017, http://www.sagemath.org.
[54] Riscure, "icWaves: A security test tool for side channel analysis and fault injection testing," https://www.riscure.com/product/icwaves/.
[55] A. Beckers, J. Balasch, B. Gierlichs, and I. Verbauwhede, "Design and implementation of a waveform-matching based triggering system," in *COSADE*, ser. LNCS, F. Standaert and E. Oswald, Eds., vol. 9689. Springer, 2016, pp. 184–198.
[56] J. Merkle and M. Lochter, *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*, IETF, {RFC 5639}.
[57] J. M. Pollard, "Monte Carlo methods for index computation (mod p)," *Mathematics of Computation*, vol. 32, no. 143, pp. 918–924, 1978.
[58] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, "Elliptic curve cryptography (ECC) cipher suites for transport layer security (TLS)," Internet Requests for Comments, RFC Editor, RFC 4492, May 2006, http://www.rfc-editor.org/rfc/rfc4492.txt.
[59] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA 2016*, ser. LNCS, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., vol. 9721. Springer International Publishing, 2016, pp. 300–321.
[60] D. Springall, Z. Durumeric, and J. A. Halderman, "Measuring the security harm of TLS crypto shortcuts," in *IMC 2016*, P. Gill, J. S. Heidemann, J. W. Byers, and R. Govindan, Eds. ACM, 2016, pp. 33–47.
[61] H. Kario, "Bug 1166338: Don't reuse ECDHE key by default," NSS Bugzilla Bugtracker, May 2015, https://bugzilla.mozilla.org/show_bug.cgi?id=1166338.
[62] N. Heninger, "Fun with the hidden number problem," Talk at the AMS Special Session on the Mathematics of Cryptography, Mar. 2019, https://public.csusm.edu/ssharif/hawaii/heninger.pdf.
[63] A. Takahashi, M. Tibouchi, and M. Abe, "New Bleichenbacher Records: Fault Attacks on qDSA Signatures," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 3, pp. 331–371, 2018.
[64] ARM Limited., "mbed TLS," https://tls.mbed.org/.

Fig. 8: Complete command line operations in Section III

```
# Generate a key pair including compressed base point
openssl ecparam -out sk.pem -name secp256k1 -genkey \
        -conv_form compressed -param_enc explicit
openssl ec -in sk.pem -pubout -out pk.pem

# Alternative way: generate a parameter file, then derive a
↪  key pair
openssl ecparam -out ec_param.pem -name secp256k1 \
        -param_enc explicit
openssl ecparam -out sk.pem -in ec_param.pem -genkey \
        -conv_form compressed
openssl ec -in sk.pem -pubout -out pk.pem

# Sign/verify with ECDSA
openssl dgst -sha256 -sign sk.pem file.txt > sigma.sig
openssl dgst -sha256 -verify pk.pem \
        -signature sigma.sig file.txt

# Encrypt/decrypt with SM2 ECIES (assuming
↪  EVP_PKEY_set_alias_type(pkey, EVP_PKEY_SM2) is set)
openssl pkeyutl -encrypt -in file.txt -pubin \
        -inkey pk.pem -out cipher
openssl pkeyutl -decrypt -in cipher -inkey sk.pem
```

Fig. 9: Complete command line operations in Section IV

```
# Bob generates a key pair including compressed base point
openssl ecparam -out sk.pem -name prime192v1 -genkey \
        -conv_form compressed -param_enc explicit
openssl ec -in sk.pem -pubout -out pk.pem

# Derive ECDH premaster secret with Alice's public key as
↪  input
openssl pkeyutl -derive -inkey sk.pem \
        -peerkey pk_alice.pem -out pms.bin

# Generate master secret key with SHA256
openssl dgst -sha256 -binary pms.bin  > K.bin

# Encrypt hello message with AES-256-CBC
echo "hello" | openssl enc -aes256 -k K.bin -e -out C.bin
```

[65] "wolfSSL," https://www.wolfssl.com/.
[66] "Crypto++ Library 7.0," https://www.cryptopp.com/.
[67] "libsecp256k1," https://github.com/bitcoin-core/secp256k1.
[68] D. Wang, "Secure Implementation of ECDSA Signatures in Bitcoin," Master's thesis, University College London, Sep. 2014, http://www.nicolascourtois.com/bitcoin/thesis_Di_Wang.pdf.
[69] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009, http://www.bitcoin.org/bitcoin.pdf.

## APPENDIX A
## OPENSSL COMMAND LINE OPERATIONS

See Fig. 8 and Fig. 9.

## APPENDIX B
## ASSEMBLY CODE FOR `BN_BIN2BN()` FUNCTION

See Fig. 10. We observed that the instructions with the comment "`SKIP !!`" cause the return value of `BN_bin2bn()` function to have 0 when skipped.

Fig. 10: Complete assembly code for `BN_bin2bn()` function, generated by GCC 6.3.0 in Raspberry Pi

```
        .arch armv6
        .align  2
        .global BN_bin2bn
        .syntax unified
        .arm
        .fpu vfp
        .type   BN_bin2bn, %function
BN_bin2bn:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        push    {r4, r5, r6, r7, r8, r9, r10, lr}
        subs    r8, r2, #0
        mov     r4, r0
        mov     r6, r1
        movne   r10, #0
        beq     .L351
.L330:

        cmp     r6, #0
        ble     .L332
        ldrb    r3, [r4]
        cmp     r3, #0
        bne     .L332
        add     r3, r4, #1
.L334:
        subs    r6, r6, #1
        mov     r4, r3
        beq     .L333
        ldrb    r2, [r3]
        add     r3, r3, #1      @ SKIP!!
        cmp     r2, #0
        beq     .L334
.L332:
        cmp     r6, #0
        bne     .L335   @ SKIP!!
.L333:
        mov     r9, r8
        mov     r3, #0
        str     r3, [r8, #4]
.L329:
        mov     r0, r9
        pop     {r4, r5, r6, r7, r8, r9, r10, pc}
.L335:
        sub     r5, r6, #1
        mov     r0, r8
        lsr     r7, r5, #2
        add     r7, r7, #1
        mov     r1, r7
        bl      bn_wexpand(PLT)
        and     r5, r5, #3
        subs    r9, r0, #0
        beq     .L352
        mov     r2, #0
        mov     r3, r2
        add     r6, r4, r6
        mov     r0, r2
        str     r7, [r8, #4]
        str     r2, [r8, #12]
.L337:
        ldrb    r1, [r4], #1
        cmp     r5, #0
        sub     r5, r5, #1
        orr     r3, r1, r3, lsl #8
        beq     .L338           @ SKIP!!
        cmp     r4, r6
        bne     .L337
        mov     r0, r8
        bl      bn_correct_top(PLT)
        mov     r9, r8
.L353:
        mov     r0, r9
        pop     {r4, r5, r6, r7, r8, r9, r10, pc}
.L338:
        ldr     r2, [r8]
        sub     r7, r7, #1
        cmp     r4, r6
        str     r3, [r2, r7, lsl #2] @ SKIP!!
        mov     r5, #3
        mov     r3, r0
        bne     .L337
        mov     r0, r8
        bl      bn_correct_top(PLT)
        mov     r9, r8
        b       .L353
.L351:
        bl      BN_new(PLT)
        subs    r8, r0, #0
        movne   r10, r8
        bne     .L330
        mov     r9, r8
        b       .L329
.L352:
        mov     r0, r10
        bl      BN_free(PLT)
        b       .L329
        .size   BN_bin2bn, .-BN_bin2bn
```