*Thesis no: MSCS-2015-09*

# On the Applicability of a Cache Side-Channel Attack on ECDSA Signatures

## The Flush+Reload attack on the point multiplication in ECDSA signature generation process

**Sai Prashanth Josyula**

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

**Contact Information:**
Author(s):
Sai Prashanth Josyula
E-mail: sajo14@student.bth.se

University advisor:
Dr. Andrew Moss
Dept. of Computer Science & Engineering

| | | |
|---|---|---|
| Faculty of Computing | Internet : | www.bth.se |
| Blekinge Institute of Technology | Phone : | +46 455 38 50 00 |
| SE–371 79 Karlskrona, Sweden | Fax : | +46 455 38 50 57 |

# Abstract

**Context**. Digital counterparts of handwritten signatures are known as Digital Signatures. The Elliptic Curve Digital Signature Algorithm (ECDSA) is an Elliptic Curve Cryptography (ECC) primitive, which is used for generating and verifying digital signatures. The attacks that target an implementation of a cryptosystem are known as side-channel attacks. The FLUSH+RELOAD attack is a cache side-channel attack that relies on cache hits/misses to recover secret information from the target program execution. In elliptic curve cryptosystems, side-channel attacks are particularly targeted towards the *point multiplication* step. The Gallant-Lambert-Vanstone (GLV) method for point multiplication is a special method that speeds up the computation for elliptic curves with certain properties.
**Objectives**. In this study, we investigate the applicability of the FLUSH+RELOAD attack on ECDSA signatures that employ the GLV method to protect point multiplication.
**Methods**. We demonstrate the attack through an experiment using the curve $secp256k1$. We perform a pair of experiments to estimate both the applicability and the detection rate of the attack in capturing side-channel information.
**Results**. Through our attack, we capture side-channel information about the decomposed GLV scalars.
**Conclusions**. Based on an analysis of the results, we conclude that for certain implementation choices, the FLUSH+RELOAD attack is applicable on ECDSA signature generation process that employs the GLV method. The practitioner should be aware of the implementation choices which introduce vulnerabilities, and avoid the usage of such ECDSA implementations.



**Keywords:** Digital signatures, Elliptic curve cryptography, GLV method, Side-channel attack.

# Acknowledgements

I thank my supervisor, Dr. Andrew Moss for his magnificent supervision. I thank him for his patience and encouragement. This work would not have been possible without his immense knowledge, exceptional guidance and extraordinary support.

I thank my parents, my sister, and my loved ones for their unconditional love and continuous support. Finally, I express my gratitude to the Sanskrit alphabet.

# List of Figures

# List of Tables

# List of Algorithms

# Contents

# Chapter 1

# Introduction

Popular protocols such as Bitcoin, Secure Shell (SSH), and Transport Layer Security (TLS) are employed in applications that are used by many users. These protocols include robust techniques that prevent an attacker from exploiting the services provided to a user. Bitcoin, SSH, TLS, and many other protocols employ cryptography for implementing such robust techniques. Even when the cryptography is secure, the implementations of the schemes used in these popular protocols suffer from many vulnerabilities [3].

Analysis of potential security flaws of cryptographic implementations is an important task of cryptographic research [2]. A cryptosystem is prone to several kinds of attacks. The attacks that target leakage from the implementation of a cryptographic system are called as side-channel attacks. Information leaked outside of the normal result of a cryptographic function is known as side-channel information. A side-channel attack occurs when the attacker uses the side-channel information for cryptanalysis [4][5]. We briefly describe some of the various kinds of side-channel attacks:

- *Timing attacks:* These attacks are based on analyzing the time taken to execute various computations in a cryptographic implementation.
- *Power-monitoring attacks:* These attacks are based on analyzing the power consumption by the underlying hardware during computations involved in a cryptographic implementation.
- *Differential fault analysis:* In these attacks, the attacker induces unexpected environmental conditions (*faults*) into a cryptographic implementation, in order to gain knowledge about its internal states.

Each attack exploits a certain characteristic in the implementation of the cryptographic function.

Digital counterparts of handwritten signatures are known as digital signatures [6]. Digital signature schemes are designed to implement digital signatures. These schemes can provide cryptographic services such as Data Integrity, Data Origin Authentication, and Non-Repudiation [6]. Data Integrity assures that the data has not been altered by unauthorized entities. Data Origin Authentication assures that origin of the data as claimed in the digital signature is the actual source of the data. Non-repudiation assures that an entity does not deny its previous actions or commitments.

Digital signatures can be implemented through various digital signature algorithms. Digital signature schemes that employ Elliptic Curve Cryptography (ECC) have an advantage of generating digital signatures significantly faster than other alternatives. They also provide smaller signature sizes at the same security level as other widely deployed schemes. Hence, Digital signature schemes that use ECC are increasingly being used in practice to implement digital signatures [3]. Elliptic Curve Digital Signature Algorithm (ECDSA) is an ECC primitive, which is used for generating and verifying digital signatures. Using ECC, the digital signature is generated by the ECDSA signature generation algorithm. Such a digital signature is called as an ECDSA signature. To check the validity of the ECDSA signature, it is verified with the ECDSA signature verification algorithm.

Cryptographic mechanisms involved in Elliptic curve cryptosystems depend on arithmetic involving the points of an elliptic curve [1]. Elliptic curve addition is significantly complex compared to the ordinary arithmetic addition. The fundamental operation in elliptic curve cryptosystems is point multiplication [7]. Though there is a significant difference in the addition operation between ordinary arithmetic and elliptic curve arithmetic, the multiplication operations are similar to each other. The elliptic curve multiplication operation, which is also known as scalar multiplication, is the repeated application of point addition. Continuous repetition of a small task by a program introduces a known structure into the overall task performed by the program. In cryptographic applications, such a structure is to be hidden in order to prevent side-channel attacks. In elliptic curve cryptosystems, side-channel attacks are particularly targeted towards the step which computes the point multiplication [8].

# Chapter 2

# Background

Various methods exist to compute the point multiplication, with special methods for accelerating the computation on specific types of elliptic curves [1]. Some of the general methods for point multiplication are Window methods, Montgomery's method, and Comb methods [1]. The Gallant-Lambert-Vanstone method (GLV method) for point multiplication is a special method that speeds up the point multiplication process for elliptic curves with efficiently-computable endomorphisms. These endomorphisms of an elliptic curve are mappings that compute alternative representations for points on the curve. The GLV method exploits a fast endomorphism that is efficiently-computable at runtime [9]. In signature generation algorithm of ECDSA (Algorithm 2.1), the base point is fixed and hence there exist various efficient methods for point multiplication that make use of precomputation tables. Therefore, due to the common use of precomputation tables, the GLV method is not widely used in signature generation algorithm of ECDSA. But it has the potential to be used in the ECDSA signature generation algorithm in the absence of precomputation tables.

ECDSA is typically executed on a machine including a cache. Basic purpose of cache memory is to store contents of the RAM that are frequently referenced by the processor during the execution of a program. Cache memory is implemented as small blocks of fast memory. Cache side-channel attacks rely on the timing difference between accessing cached values and non-cached values [4]. Recently, a cache side-channel attack known as the FLUSH+RELOAD attack [4] was introduced. Prior to the introduction of this attack, the majority of cache side-channel attacks relied upon monopolizing the Level 1 (L1) cache memory. Yarom and Falkner [4] mention the limitations of such cache side-channel attacks.

Recently, B. B. Brumley [9] carried out an instruction cache side-channel attack [10] against implementations of Elliptic Curve Diffie-Hellman (ECDH) and ECDSA. The implementation of ECDH uses the GLV method for point multiplication. The instruction cache attack targets the L1 cache level. For the attack to be successful, the victim process and the spy process must execute on the same execution core of the processor. In their attack, the spy process monitors victim's access to sets of memory lines that map to the same cache set. Given the nature of such a cache side-channel attack, the attacker's options are very limited. The author also carried out the instruction cache side-channel attack against an implementation of ECDSA signature generation process that makes use of precomputation tables.

In general, cache based side-channel attacks monitor access to sets of memory lines that map to the same cache set rather than monitoring specific memory lines. This introduces many false positives in the captured memory accesses. The FLUSH+RELOAD attack monitors access to specific memory lines. It measures if a particular target address is in the cache memory. The ability to monitor specific memory lines reduces the chance of false positives [2]. Categorically, this is a new class of attack. Unlike the attack in [10], the FLUSH+RELOAD attack targets the Last Level Cache (LLC). The LLC is shared between cores, and hence the attack can be accomplished even when the victim process and the spy process execute on different cores of the processor.

In order to perform the FLUSH+RELOAD cache side-channel attack, the attacker exploits a property of Intel implementation of x86 and x86_64 processor architectures [4][2]. The FLUSH+RELOAD attack has been shown to be successful against implementations of certain cryptographic primitives in widely used cryptographic libraries [11][12]. The implementation of RSA in GNU Privacy Guard (GnuPG) library [11] which was shown to be vulnerable to the attack [4] has been patched appropriately in subsequent versions of GnuPG. Most of the side-channel attacks are specific to a particular implementation of a cryptographic algorithm and cannot be generalized [13]. The applicability of the FLUSH+RELOAD attack depends on the algorithm under attack and its implementation details. Further research is required to determine the applicability of the FLUSH+RELOAD attack to various other implementations of different cryptographic systems.

## 2.1 Cache Architecture

Modern processors employ a cache architecture that is usually organized as a hierarchy of cache levels (Level 1 (L1), Level 2 (L2), ...). Accessing cache levels that are close to the processor is faster than accessing cache levels that are further away from the processor.

Multi-level caches fall into two main categories: *inclusive* caches, and *exclusive* caches. In an inclusive cache architecture, all the data present in a lower-level cache (cache levels that are away from the processor) is also present in all the higher-level caches (cache levels that are close to the processor). In an exclusive cache architecture, any data that resides in the cache memory is guaranteed to be in at most one of the caches exclusively.

## 2.2 Cache Side Channel attacks

Being a shared resource that all processes compete for, the cache affects every process and is affected by each of them. The cache architecture causes subtle

**Algorithm 2.1:** ECDSA signature generation

```
Input: Elliptic Curve E, Base point G of Order n, Message m,
       Private-Public key pair (α, Q).
Output: The ECDSA signature (r, s).
Compute e = Hash(m), where Hash is an approved Hash algorithm.
Let b be the bitlength of n. Let h be the b leftmost bits of e.
Generate a cryptographically secure random nonce k ∈ Z_n.
Compute the point (x, y) = [k] G ∈ E.
r = x (mod n).
if r = 0 then
    return to step 3.
else
    Compute s = k^{-1} (h + r · α) (mod n).
    if s = 0 then
        return to step 3.
    else
        return (r, s).
    end
end
```

indirect interaction between processes [14].

While a program is executing, the processor needs to retrieve instruction and data that are necessary for the execution. The instructions and data can be retrieved from main memory or from the cache memory. Retrieval of contents from the main memory takes longer than retrieval from cache memory. Similarly, retrieval of contents from lower cache levels takes longer than their retrieval from the higher cache levels. Cache side channel attacks exploit this difference in access timings.

Tromer et al. [14] describe several side-channel attacks (Evict+Time attack, Prime+Probe, etc.) based on inter-process leakage through the state of cache memory.

## 2.3 Mathematical Background

### 2.3.1 Abstract Algebra Preliminaries

We define the basic concepts in Abstract Algebra that are required in order to understand the elliptic curves. The definitions are influenced by the definitions of the concepts as presented in the Guide to Elliptic Curve Cryptography [1].

**Definition 2.1.** *A Group denoted by* $\{G, +\}$ *consists of a set* $G$ *with a binary operation* $+$ *satisfying the following properties:*

- *Closure: The binary operator* $+$ *is closed on the set* $G$.

  $$\forall\ a, b \in G, \textit{the element } a + b = c \in G.$$

  *Note: '*$+$*' operation is usually called as 'addition', though it is not related to the arithmetic addition.*

- *Associativity: The binary operator* $+$ *is associative on the set* $G$.

  $$a + (b + c) = (a + b) + c\ \forall\ a, b, c \in G.$$

- *Existence of a unique identity element:* $\exists$ *an element* $e \in G$ *such that:*

  $$a + e = e + a = a\ \forall a \in G.$$

  *Note: The additive identity element* $e$ *is usually denoted by* $0$.
  *The additive inverse of* $a$ *is denoted by* $-a$.

- *Existence of inverse elements: For each* $a \in G$, $\exists$ *an element* $b \in G$ *such that:*

  $$a + b = b + a = e.$$

  *Note:* $b$ *is called the inverse of* $a$.

**Definition 2.2.** *An Abelian Group denoted by* $\{G, +\}$ *is a Group which satisfies the following additional property:*

- *Commutativity: The binary operator* $+$ *is commutative on the set* $G$.

  $$a + b = b + a\ \forall\ a, b \in G.$$

**Definition 2.3.** *A Ring denoted by* $\{R, +, \times\}$ *consists of a set* $R$ *and the operator* $+$ *which form an abelian group* $\{R, +\}$, *while the binary operator* $\times$ *satisfies the following additional properties on the set* $R$:

- *Closure: The binary operator* $\times$ *is closed on the set* $G$.

  $$\forall\ a, b \in G, \textit{the element } a \times b = c \in G.$$

  *Note: '*$\times$*' operation is usually called as 'multiplication', though it is not related to the arithmetic multiplication.*

- *Associativity: The binary operator* $\times$ *is associative on the set* $G$:

  $$a \times (b \times c) = (a \times b) \times c\ \forall\ a, b, c \in G.$$

  *Note: The multiplication operation* $\times$ *is frequently shown by concatenation.*

- *Distributivity: The multiplication operator* $\times$ *distributes over the addition operator* $+$.

$$a(b+c) = ab + ac, (a+b)c = ac + bc.$$

**Definition 2.4.** *A Commutative Ring denoted by* $\{R, +, \times\}$ *is a Ring which satisfies the following additional property:*

- *Commutativity: The multiplication operator* $\times$ *is commutative on the set* $R$.

$$ab = ba \ \forall \ a, b \in R.$$

**Definition 2.5.** *An Integral Domain, denoted by* $\{R, +, \times\}$ *is a Commutative Ring whose elements satisfy the following two additional properties:*

- *Existence of a unique identity element:* $\exists$ *an element* $e \in R$ *such that:*

$$ae = ea = a \ \forall a \in G.$$

  *Note: The multiplicative identity element* $e$ *is usually denoted by 1.*
  *The multiplicative inverse of* $a$ *is denoted by* $a^{-1}$.

- *If* $ab = 0$, *then either* $a = 0$ *or* $b = 0$, *where* $0$ *is the additive identity element.*

**Definition 2.6.** *A Field, denoted by* $\{F, +, \times\}$ *is an Integral Domain whose elements satisfy the following additional property:*

- *For each* $a \in F - \{0\}, \exists$ *an element* $b \in F - \{0\}$ *such that:*

$$ab = ba = 1.$$

### 2.3.2 Terminology of Abstract Algebra

1. **Finite Field:** If the set $F$ is finite, then the Field $\{F, +, \times\}$ is said to be a *Finite Field.*

2. **Order of a Finite Field:** The *order* of a Finite Field is the number of elements in the Field.

3. **Existence of a Finite Field:** There exists a Finite Field $\{F, +, \times\}$ of order $q$ if and only if $q$ is a prime power.

$$\exists \text{ a Finite Field } \{F, +, \times\} \iff q = p^m,$$

   where $p$ is a prime number.

4. **Prime Field:** A *Prime Field* is a Finite Field $\{F, +, \times\}$ with an order $q$, where $q$ is a prime number.

5. **Extension Field:** An *Extension Field* is a Finite Field $\{F, +, \times\}$ with order $q = p^m$, where $p$ is a prime number and $m \geq 2$.

   *Note:* Henceforth, we denote the Finite Field $\{F, +, \times\}$ by $\mathbb{F}$.

6. **SubField**: Let $f$ be a subset of $F$. If $\{f, +, \times\}$ is a Finite Field, then it is said to be the *SubField* of $\mathbb{F}$.

7. **Binary Field:** A Finite Field $\mathbb{F}$ of order $2^m$ is called a *Binary Field* or *characteristic-two Finite Field.*

8. **Order of an element:** If $\mathbb{F}$ is a finite multiplicative group of order $n$ and $e \in F$, then the smallest positive integer $t$ such that $et = 1$ is called the order of the element $e$.
   *Note:* Such a positive integer $t$ always exists and is a divisor of $n$.

### 2.3.3 Elliptic Curves

**Definition 2.7.** *An Elliptic Curve $E$ over a Field $\mathbb{F}$ is a non-singular curve defined by an equation:*

$$\boxed{E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6} \tag{2.1}$$

*where $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}$.*

*Note:* Non-singularity ensures that the curve is 'smooth' and that there are no points on the curve with two or more distinct tangent lines.

### 2.3.4 Endomorphism of Elliptic Curves

Let $E$ be an Elliptic Curve defined over a field $\mathbb{F}$. The set of all points on $E$ that lie in any finite extension of $\mathbb{F}$ is also denoted by $E$.

**Definition 2.8.** *An endomorphism $\phi$ of $E$ over $\mathbb{F}$ is a rational map $\phi : E \to E$ such that:*

$$\boxed{\phi(\infty) = \infty \text{ and } \phi(P) = (g(P), h(P)) \ \forall P \in E} \tag{2.2}$$

where $g$ and $h$ are rational functions whose coefficients lie in $\mathbb{F}$.

Gallant et al. [7] present several examples of elliptic curves with endomorphisms. We present the following relevant example [15][7] of an elliptic curve with endomorphism.

**Example:** Consider the elliptic curve

$$E : y^2 = x^3 + b$$

defined over $\mathbb{F}_p$, where $p \equiv 1 \pmod 3$ is a prime.
Let $\beta \in \mathbb{F}_p$ be an element of order 3. Then the map:

$$\boxed{\phi : E \to E \text{ defined by } (x, y) \mapsto (x, \beta y),\ \ \phi : \infty \to \infty}$$

is an endomorphism of $E$ defined over $\mathbb{F}_p$.

If $P \in E(\mathbb{F}_p)$ is a point of prime order $n$, then $\phi$ acts on $P$ as a multiplication map. The characteristic polynomial of $\lambda$ is $z^2 + z + 1$. The integer $\lambda$ is a root modulo $n$ of the characteristic polynomial of $\phi$. Therefore, we have

$$\boxed{\lambda^2 + \lambda + 1 \equiv 0 \pmod n} \tag{2.3}$$

### 2.3.5 Examples of Elliptic Curves

**Example 1:** The elliptic curve

$$E : y^2 = x^3 + 7$$

over the field $\mathbb{F}_p$, where $p = 2^{256} - 2^{32} - 977$ is a prime, and $p \equiv 1 \pmod 6$. This elliptic curve is called the **secp256k1** curve. It can be observed that the curve satisfies all the criteria as mentioned in Section 2.3.4. Other details specific to the secp256k1 curve are mentioned in Appendix C.

**Example 2 [1]:** The elliptic curve

$$E : y^2 = x^3 - 2$$

over the field $\mathbb{F}_p$, where $p = 2^{390} + 3$ is a prime, and $p \equiv 1 \pmod 3$. We can notice that the curve satisfies all the criteria as mentioned in Section 2.3.4 and thus has an an endomorphism.

$$\lambda = \frac{2^{195} - 2}{3}$$

The integer $\lambda$ satisfies $\lambda^2 + \lambda + 1 \equiv 0 \pmod n$ and $\lambda(x, y) = (\beta x, y)$ for all $(x, y)$ in the order-$n$ subgroup of $E(\mathbb{F}_p)$, for $\beta = 2^{389} + 2^{194} + 1$.

## 2.4 GLV decomposition

Acceleration of point multiplication with a scalar multiplier on elliptic curves with efficiently computable endomorphisms consists of the following phases:

- Decomposing the scalar multiplier.
- Performing point multiplication with the decomposed scalars.

By *GLV method*, we mean both decomposing a scalar multiplier, and performing point multiplication with the decomposed scalars using a particular algorithm. Decomposition of the scalar multiplier is called *GLV decomposition.* The point multiplication can be performed on the decomposed scalars in many ways. In OpenSSL, it is performed by *interleaved scalar multiplication* (Algorithm 2.2). The GLV decomposition of a scalar multiplier $k$ into scalars $k_1$ and $k_2$ can be done

**Algorithm 2.2:** Interleaved Scalar Multiplication

`ec_wNAF_mul()`
**Input**: $v$, integers $k_j$, width $w_j$ and points $P_j$, $1 \leq j \leq v$.
`/*` $v = 2$, $k_1$ and $k_2$ are the decomposed scalars. `*/`
**Output**: $\sum_{j=1}^{v} k_j P_j$
`/*` $k_1P_1 + k_2P_2$. `*/`
Compute the modified $w$NAF form of $k_1$ and $k_2$.
$Q \leftarrow \infty$.
**for** $i$ *from* $l-1$ *downto* $0$ **do**
  $Q \leftarrow 2Q$.
  **for** $j$ *from* $1$ *to* $v$ **do**
    **if** $k_j^i \neq 0$ **then**
      `/*` $k_j^i$ represents the $i^{th}$ bit of $k_j$. `*/`
      **if** $k_j^i > 0$ **then**
        $Q \leftarrow Q + k_j^i P_j$
      **else**
        $Q \leftarrow Q - k_j^i P_j$
**return** $Q$

by various methods [7][16][17][18]. Gallant, Lambert and Vanstone [7] describe the GLV decomposition algorithm which takes as input integers $n, \lambda, k \in [1, n-1]$ and returns integers $k_1$ and $k_2$ such that $k \equiv k_1 + k_2\lambda \pmod{n}$.

The GLV decomposition of a scalar multiplier $k$ can be framed into the following problem [7]. Let $G = \mathbb{Z} \times \mathbb{Z}$ and consider the homomorphism $f : G \to \mathbb{Z}_n$ defined by $(i, j) \mapsto (i + \lambda j) \pmod{n}$.
**Problem:** To find a short vector $u\,(k_1, k_2) \in G$ such that $f(u) = k$.
The vector $u'(k, 0)$ satisfies $f(u') = k$, but it is not a short vector. The above problem is solved by breaking it into two subproblems:

- **Subproblem 1: Finding $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$**
  To find linearly independent short vectors $v_1(a_1, b_1), v_2(a_2, b_2) \in G$ such that $f(v_1) = f(v_2) = 0$.

- **Subproblem 2: Finding $\boldsymbol{v}$**
  To find a vector $v(c_1, c_2)$ in the integer lattice generated by $v_1(a_1, b_1)$ and $v_2(a_2, b_2)$ that is close to $u'(k, 0)$.

Then, $u = u'(k, 0) - v(c_1, c_2)$ is a short vector with $f(u) = f(u') - f(v) = k$. The first subproblem can be solved independently of $k$, and hence is generally carried out as a precomputation. This is almost always solved by using the Extended Euclidean Algorithm. So, we label each method by the technique used to solve the Subproblem 2.

### 2.4.1 Method 1: Babai's rounding method

The problem of finding two independent short vectors $v_1\,(a_1, b_1)$ and $v_2\,(a_2, b_2)$, such that $f(v_1) = f(v_2) = 0$ can be solved by the Extended Euclidean Algorithm [7]. A vector $v$ in the integer lattice generated by $v_1$ and $v_2$, that is close to $(k, 0)$ can be found by elementary linear algebra [7]. In particular, Babai's rounding method [19] is used. $v = c_1v_1 + c_2v_2$, where the vectors $v_1\,(a_1, b_1)$, $v_2\,(a_2, b_2)$, and the scalars $c_1$, $c_2$ are defined in the Algorithm 2.3.

**Algorithm 2.3:** Balanced length-two representation of a multiplier [1]

`ec_GFp_glv_decompose()`
**Input**: Integers $n, \lambda, k \in [0, n-1]$.
**Output**: Integers $k_1, k_2$ such that $k = k_1 + k_2\lambda \pmod{n}$ and $|k_1| = |k_2| \approx \sqrt{n}$.
Run the Extended Euclidean Algorithm with $(n, \lambda)$ as input.
The Extended Euclidean algorithm produces a sequence of equations of the form $s_in + t_i\lambda = r_i$, where $(r_0, t_0, s_0) = (n, 0, 1)$ and $(r_1, t_1, s_1) = (\lambda, 1, 0)$, and the non-negative remainders $r_i$ are strictly decreasing.
Let $m$ be the greatest index for which $r_m \geq \sqrt{n}$.
$(a_1, b_1) \leftarrow (r_{m+1}, -t_{m+1})$.
**if** $(r_m^2 + t_m^2) \leq \left(r_{m+2}^2 + t_{m+2}^2\right)$ **then**
  $(a_2, b_2) \leftarrow (r_m, -t_m)$.
**else**
  $(a_2, b_2) \leftarrow (r_{m+2}, -t_{m+2})$.
Compute $c_1 = \lfloor b_2k/n \rceil$ and $c_2 = \lfloor -b_1k/n \rceil$.
**return** $(k_1, k_2)$.

This method involves a number of multiplications, divisions, rounding operations, and computations in the rationals [18]. The values specific to the secp256k1 curve are presented in Appendix C.

### 2.4.2 Method 2: Extended Euclidean Algorithm

Brumley and Nyberg [18] present a method for decomposition of integers with a desired ratio of balance between the size. Their method allows an unbalanced representation of the scalar $k$ and also provides strict bounds on the decomposed scalars, $k_1$ and $k_2$. The Extended Euclidean Algorithm is run on the parameters $(n, \lambda)$ and the following integers are obtained:

- A sequence of positive quotients $q_1, \ldots, q_m$.
- Decreasing sequence of positive remainders $r_1, \ldots, r_m$.
- Sequence of integers $s_1, \ldots, s_{m+1}$.
- Sequence of integers $t_1, \ldots, t_{m+1}$.

With $r_0 = n, r_1 = \lambda, s_0 = 1, s_1 = 0, t_0 = 1, t_1 = 1$, and the integers obtained from running the Extended Euclidean Algorithm on $(n, \lambda)$, the following conditions are satisfied:

$$\begin{aligned} r_{i-1} &= q_i r_i + r_{i+1}, \\ t_{i+1} &= t_{i-1} - q_i t_i, \\ r_i &\equiv t_i \lambda + s_i n \end{aligned}$$

for all $i = 1, \ldots, m$.
Let the index $m > 0$ be fixed but arbitrary. Given $k \in [0, n-1]$ , we iteratively divide $k$ by $r_1, \ldots, r_m$:

$$\boxed{\begin{aligned} & k = a_1 r_1 + j_1 \\ & j_1 = a_2 r_2 + j_2 \\ & \cdots \\ & j_{m-1} = a_m r_m + j_m \end{aligned}}$$

where, $0 \leq j_i < r_i$.

$$\boxed{\begin{aligned} k_1 &= j_m \\ k_2 &= \sum_{i=1}^{m} a_i r_i \\ k_1 + k_2\lambda &\equiv k \pmod{n} \end{aligned}}$$

The algorithm presented by Brumley and Nyberg [18] has a non-trivial typing error; they mention the Line 9 of Algorithm 2.4 as $k_2 \leftarrow k_2 + qt_1$. We implemented Algorithm 2.4 and ensured that it is in accordance with the rest of the work presented in [18].

**Algorithm 2.4:** Integer decomposition using Extended Euclidean algorithm

**Input**: Integers $n, \lambda, k \in [0, n-1]$, bound index $m$.
**Output**: Integers $k_1, k_2$ such that $k = k_1 + k_2\lambda \pmod{n}$ and $0 \leq k_1 < A_m, |k_2| < B_m$.

/* $A_m = r_m$, $B_m = |t_m| + |t_{m+1}|$. */

1 $k_0 \leftarrow k, k_1 \leftarrow 0, k_2 \leftarrow 0, r_0 \leftarrow n, r_1 \leftarrow \lambda, t_0 \leftarrow 0, t_1 \leftarrow 1$.
/* $(quotient, remainder) \leftarrow$ `DivRem`$(dividend, divisor)$. */
2 **for** $i = 1$ *to* $m$ **do**
3     $(a, k_1) \leftarrow \text{DivRem}(k_0, r_1)$.
4     $(b, c) \leftarrow \text{DivRem}(r_0 - k_0, r_1)$.
5     $r_0 \leftarrow r_1, r_1 \leftarrow k_1 + c, q \leftarrow a + b, k_0 \leftarrow k_1$.
6     **if** $r_1 \geq r_0$ **then**
7         $r_1 \leftarrow r_1 - r_0$.
8         $q \leftarrow q + 1$.
9     $k_2 \leftarrow k_2 + at_1$.
10     $(t_0, t_1) \leftarrow (t_1, t_0 - t_1 q)$.
11 **return** $(k_1, k_2)$.

**Lemma 2.1.** *[18] Suppose that sequences $r_i, q_{i+1}, t_i$ and $u_i, i \geq 0$ have been obtained by running the Extended Euclidean Algorithm for integers $n$ and $\lambda$. Then, for any $i \geq 1, j$ such that $0 \leq j < q_{i+1}$, and $\kappa$ such that $0 < |\kappa| < |t_i| + (j+1)|t_{i+1}|$, there exist integers $a$ and $b$ such that*

$$\boxed{\kappa\lambda \ (mod\ n) = a(r_i - jr_{i+1}) + br_{i+1}} \tag{2.4}$$

*where* $0 \leq a \leq |t_{i+1}|$ *and* $0 \leq b \leq |t_i| + j|t_{i+1}|$. *Moreover,* $a = 0$ *and* $b = 0$ *do not hold simultaneously, and similarly,* $a = |t_{i+1}|$ *and* $b = |t_i| + j|t_{i+1}|$ *do not hold simultaneously.*

### 2.4.3 Method 3: Curves with Special Parameters

Brown et al. [20] introduce special elliptic curves which they term as *compact curves.* These curves can be considered as 'special' cases of GLV curves which allow us to obtain an efficient GLV decomposition [21]. Cohen and Frey [21] point out that the compact curves are 'very special' and thus are susceptible to specialized attacks in spite of their positive applications like efficient GLV decomposition.

Let us consider the example curve presented in Section 2.3.5. It is a compact curve that allows the decomposition of $k$ into $k_1$ and $k_2$ to be obtained with little effort compared to Algorithm 2.3 and Algorithm 2.4.

Let $P(x, y) \in$ order-$n$ subgroup of $E(\mathbb{F}_p)$ and $k \in [0, n-1]$ be the scalar that is to be decomposed into scalars $k_1$ and $k_2$. To find a balanced length-two representation of $k$, we write

$$k = 2^{195}k'_2 + k'_1, \text{ where } k'_1 < 2^{195}.$$

$$\boxed{\begin{aligned} kP &= (2^{195}k'_2 + k'_1)P \\ &= ((3\lambda + 2)k'_2 + k'_1)P \\ &= \underbrace{(2k'_2 + k'_1)}_{k_1} P + \underbrace{(3k'_2)}_{k_2} \lambda P \\ &= k_1(x, y) + k_2(\beta x, y) \end{aligned}}$$

In this example, the scalar $k$ of bitlength $\approx 384$ bits is decomposed into $k_1$ and $k_2$ with each of them approximately half the bitlength of the input scalar $k$.

### 2.4.4 Other methods

**Theory of $\boldsymbol{\mu}-$ Euclidean Algorithm:** Park et al. [16] describe an alternate method to decompose a scalar $k$ using the theory of $\mu-$Euclidean algorithm.

**Lattice Basis Reduction Algorithms:** The GLV decomposition of the scalar multiplier $k$ can be carried out by the use of lattice basis reduction algorithms. In spite of the applicability of lattice basis reduction algorithms, they are not employed since other methods are easier to implement [7].

## 2.5 $w$NAF representation

**Definition 2.9.** *[1] Let $w \geq 2$ be a positive integer. The $w$NAF representation of a positive integer $k$ is a sequence of signed digits $k_{l-1} \dots k_0$ where:*

1. $k = \sum_{i=0}^{l-1} k_i 2^i$.
2. *Each non-zero digit is odd and less than $2^{w-1}$ in absolute value.*
3. *Among any $w$ consecutive digits, at most one is non-zero.*
4. $k_{l-1} \neq 0$.

Algorithm 2.5 can be used to convert a positive integer $k$ into its $w$NAF form. The following properties of the $w$NAF form are enumerated in the Guide to Elliptic Curve Cryptography [1]:

- A positive integer $k$ has a unique $w$NAF.
- The length of $w$NAF form of $k$ is at most one more than the length of the binary representation of $k$.
- The average density of non-zero digits among all $w$NAFs of length $l$ is approximately $1/(w+1)$.

**Algorithm 2.5:** Conversion to $w$NAF form

**Input**: positive integer $k$ and window width $w$.
**Output**: $w$NAF form of $k$.

$l \leftarrow 0$.
**while** $k \geq 1$ **do**
  /* $k$ (mods $2^w$) denotes integer $u$ satisying $u \equiv k \pmod{2^w}$ and $-2^{w-1} \leq u < 2^{w-1}$. */
  **if** $k$ *is odd* **then**
    $k_l \leftarrow k$ (mods $2^w$).
    $k \leftarrow k - k_l$.
  **else**
    $k_l \leftarrow 0$.
  $k \leftarrow k/2$.
  $l \leftarrow l + 1$.
**return** $k_{l-1}, k_{l-2} \dots k_1, k_0$.

**Definition 2.10.** *Let $w$ be a positive integer. The modified $w$NAF representation of a scalar $k$ is a sequence of signed digits $k_{l-1} \ldots k_0$ where:*

1. $k = \sum_{i=0}^{l-1} k_i 2^i$.

2. *Each non-zero digit is odd and less than $2^w$ in absolute value.*

3. *Among any $w + 1$ consecutive digits, at most one is non-zero with the exception that the most significant digit may be only $w - 1$ zeros away from the next non-zero digit.*

4. $k_{l-1} \neq 0$.

Henceforth, we represent the modified $w$NAF form of $k$ by $w$NAF($k$). The `compute_wNAF` function used in OpenSSL computes the modified $w$NAF form of the scalar. The following are some of the noteworthy properties of the modified $w$NAF form:

- A scalar $k$ has a unique modified $w$NAF.
- The length of modified $w$NAF form of $k$ is at most one more than the length of the binary representation of $k$.
- The number of trailing zero digits in the modified $w$NAF form of a scalar are equal to the number of trailing zero bits in the binary form of that scalar.

## 2.6 GLV method in Practice

Elliptic curves with efficiently computable endomorphisms can employ GLV method for point multiplication (Line 4 of Algorithm 2.1). In the GLV method for computing $kG$, scalar $k$ is first decomposed into two scalars $k_1$ and $k_2$ using one of the methods for GLV decomposition. The decomposed scalars are then converted into their modified $w$NAF form. The point multiplication $kG$ is obtained by carrying out interleaved scalar multiplication (Algorithm 2.2) of $k_2P + k_1G$, where the point $P = \lambda G$.

In practice, this computation of $kG$ is slightly modified in OpenSSL code. Instead of computing the value of $kG$, the code computes $(k + x \cdot n)G$, where $x \in \{1, 2\}$ is chosen such that $\lfloor \log_2 (k + x \cdot n) \rfloor = \lfloor \log_2 (n) \rfloor + 1$ [2]. The modified step thus generates a fixed size scalar from the original scalar $k$. This provides protection against the Brumley and Tuveri remote timing attack [22]. For the secp256k1 curve, the probability that $x = 2$ is less than $2^{-125}$ [2]. Hence, we can assume that the GLV decomposition is applied to the scalar $k + n$.

## 2.7 Research questions

The FLUSH+RELOAD attack has been successfully carried out on the implementation of ECDSA signature generation in OpenSSL library [12][2]. Theoretically, the use of GLV method reduces the effectiveness of the FLUSH+RELOAD attack, though it does not completely thwart the attack [2][23]. But, this has not been confirmed through a practical investigation by applying the FLUSH+RELOAD attack on the GLV method. Hence, there is a need to investigate the applicability of the FLUSH+RELOAD attack on the GLV method and subsequently on ECDSA signatures that are generated using this method.

The aim of our work is to answer the following research questions:

- R.Q.1: How would the FLUSH+RELOAD side channel attack be applicable to GLV method for point multiplication?
- R.Q.2: How does the use of GLV method in ECDSA signature generation affect the amount of information obtained from the FLUSH+RELOAD attack?
- R.Q.3: How does the encoding of the scalar used in the GLV method defend the method against the FLUSH+RELOAD attack?

# Chapter 3

# Experimental Setup

We perform an experiment to discover answers to the research questions. The goal of our experiment is to determine how much sensitive information can be obtained about the scalars $k_1$ and $k_2$ from each ECDSA signature generation instance by applying the Flush+Reload attack across various instances. In Chapter 5, we present an elaborate discussion about analyzing the obtained side-channel information.

## 3.1 Experimental Design

Prior to setting up the experimental environment, we first select the dependent and independent variables.

- **Independent variables:** The independent variable represents the random nonce in each instance of the ECDSA signature generation process.
- **Dependent variable:** The dependent variable represents the output plot.

During the design phase of our experiment, we considered the checklist [24] regarding experimental design.

### 3.1.1 Hardware Environment

The hardware environment used for the experiment consists of a HP Pavilion g4 Notebook PC. The underlying processor is an Intel Core i5 2430M with two processing units (execution cores) and 3 MB LLC (Last Level Cache). The processor belongs to the Intel Sandy Bridge architecture [25], and has a clock speed of 2.4 GHz. Table 3.1 provides a concise summary of the hardware environment.
**Cache Architecture:** The cache hierarchy of the Core i5 2430M processor consists of three cache levels: L1, L2, and L3. Each core consists of 32KB L1 instruction cache, 32KB L1 data cache and 256KB unified L2 cache. Both the execution cores share a 3MB unified L3 cache. In the Intel Sandy Bridge architecture, each cache line is 64 bytes in size. The L1 and L2 caches are 8-way set-associative, and the L3 cache is 12-way set-associative.

| **Hardware Specification** | **Environment variable** |
|---|---|
| Processor Architecture | `Intel Sandy Bridge (x86_64)` |
| Processor name | `Intel Core i5 2430M` |
| Processor Clock speed | `2.40 GHz` |
| Execution cores | `2 cores` |
| Last Level Cache | `3 MB` |

Table 3.1: Summary of the Hardware environment

| **Level** | **Capacity** | **Associativity (ways)** | **Line size (bytes)** | **Inclusive** |
|---|---|---|---|---|
| L1 Data | 32 KB | 8 | 64 bytes | - |
| L1 Instruction | 32 KB | 8 | N/A | - |
| L2 (Unified) | 256 KB | 8 | 64 bytes | No |
| L3 (Unified) | 3 MB | 12 | 64 bytes | Yes |

Table 3.2: Cache Parameters

The L3 cache on the Intel Sandy Bridge architecture is *inclusive*. If a certain data is present in the L1 cache of a core, then it will also be present in the L2 cache of the core, and in the unified L3 cache which is common to both the cores. Thus, when data is evicted from the L3 cache, the copies of that data will be evicted from the L1 cache and the L2 cache too. Table 3.2 provides a concise summary of the cache parameters. Figure 3.1 depicts the cache architecture of Core i5 2430M processor.

### 3.1.2 Software Environment

We tested our experiment on Ubuntu 14.04 (Trusty Tahr). The underlying kernel is Linux kernel 3.13. The code has been written using the library `openssl 1.0.1l` and `glibc-2.19-1`. The code has been compiled with `gcc 4.8.2`. The graphing utility, `gnuplot 5.0` is used to generate output plots. Table 3.3 provides a concise summary of the cache parameters.

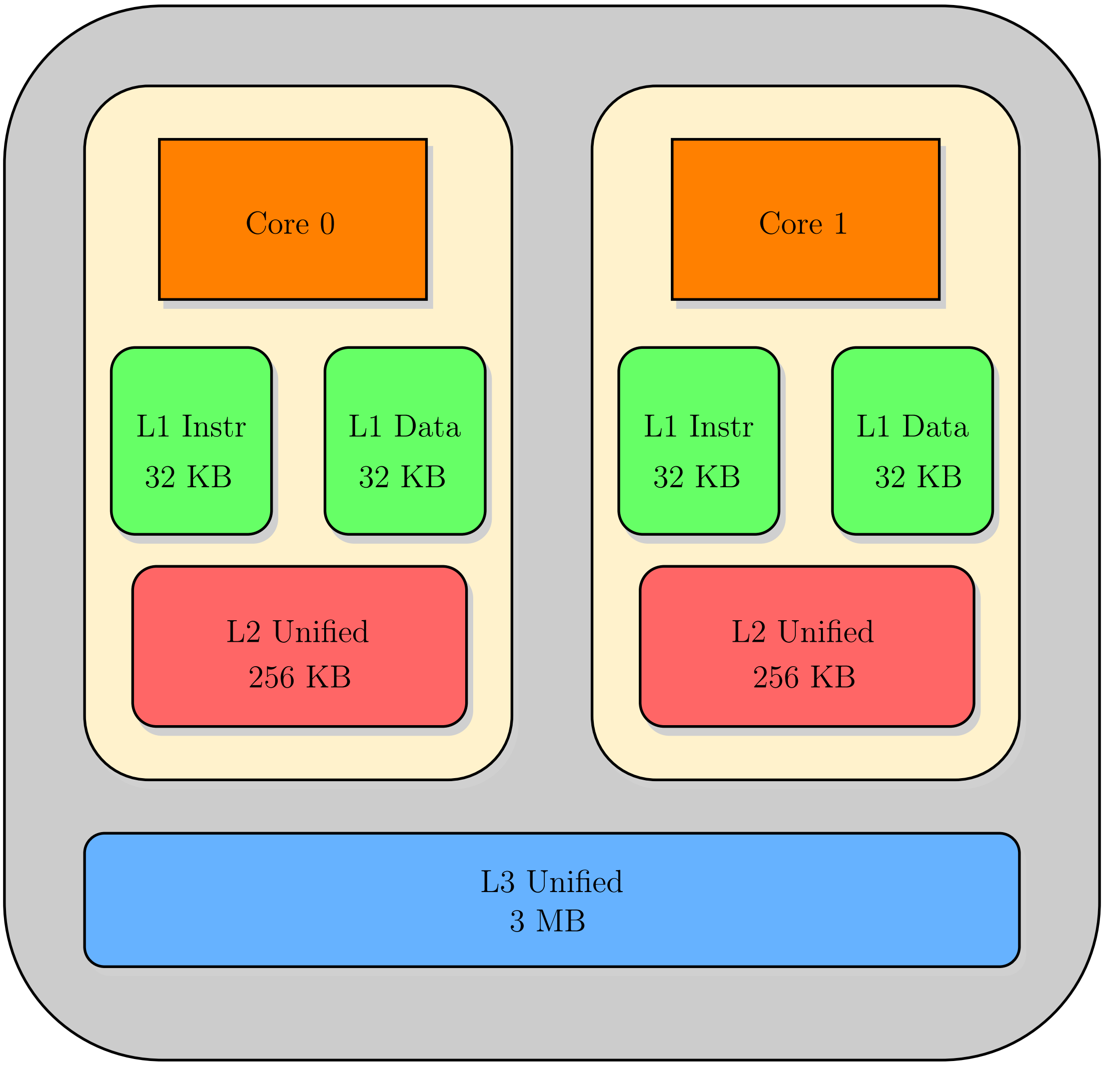



Figure 3.1: Intel Sandy Bridge Cache Architecture

| **Software Specification** | **Environment variable** |
|---|---|
| Operating system | `Ubuntu 14.04` |
| Kernel | `Linux kernel 3.13` |
| Libraries | `openssl 1.0.1l`, `glibc-2.19-1` |
| Graphing utilities | `gnuplot 5.0` |

Table 3.3: Summary of the Software environment

## 3.2 OpenSSL

OpenSSL [12] is a general purpose cryptography library for the implementation of cryptographic protocols. The library is implemented in C language and is often used to implement Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, apart from many other cryptographic standards. The OpenSSL library provides support for Elliptic Curve Cryptography (ECC). Though OpenSSL supports a number of GLV curves, it does not exploit the fast endomorphism for efficient point multiplication [9]. Recently, B. B. Brumley [9] provided an integration of the GLV method into the OpenSSL code base through a source-code patch [26].

We apply the patch [26] to `openssl-1.0.1l` source code. We then build the patched version of `openssl-1.0.1l` with debugging symbols. These symbols are not loaded when the code is executed and thus do not have an effect on the performance of the code.

We implement a C program that generates ECDSA signatures using the GLV method for point multiplication. We refer to this program as the *glv program*. The glv program carries out an ECDSA signature generation process on a randomly generated message hash. The following parameters are kept constant across all the instances of the glv program:

1. The elliptic curve parameters as mentioned in Appendix C.
2. The public key $Q$, and private key $\alpha$.

In each instance or run of the glv program, we perform the following:

1. Generate a random message hash.
2. Generate a cryptographically secure random integer (nonce).
3. Generate ECDSA signature for the message hash using the GLV method for point multiplication.

The base point is one of the required parameters for point multiplication step in ECDSA signing. The value of base point remains same across many executions of the glv program, each execution of the program corresponding to a new digital signature. The nonce is the other parameter for the point multiplication step in ECDSA signing. If the nonce of a signature generation session is disclosed, then the secret key $\alpha$ can be recovered with ease [27]. Hence, a cryptographically secure random nonce is to be used for each digital signature. The spy process is made to run in background with the glv process. While the processor executes instructions of the glv process, the spy process appropriately collects the side-channel information.

The GLV method in our OpenSSL build incorporates the use of interleaved scalar multiplication on the modified $w$NAF form of the decomposed scalars $k_1$ and $k_2$. The implementation is demonstrated in Listing 3.2.

Listing 3.2: Code snippet from the file ec_mult.c

```
for (k = max_len - 1; k >= 0; k--)
{
    if (!r_is_at_infinity)
    {
        if (!EC_POINT_dbl(group, r, r, ctx)) goto err;
    }

    for (i = 0; i < totalnum; i++)
    {
        if (wNAF_len[i] > (size_t)k)
        {
            int digit = wNAF[i][k];
            int is_neg;

            if (digit)
            {
                is_neg = digit < 0;

                if (is_neg)
                    digit = -digit;

                if (is_neg != r_is_inverted)
                {
                    if (!r_is_at_infinity)
                    {
                        if (!EC_POINT_invert(group, r, ctx))
                         ↪  goto err;
                    }
                    r_is_inverted = !r_is_inverted;
                }

                /* digit > 0 */

                if (r_is_at_infinity)
                {
```

```
690                     if (!EC_POINT_copy(r, val_sub[i][digit >>
                         ↪  1])) goto err;
691                     r_is_at_infinity = 0;
692                 }
693                 else
694                 {
695                     if (!EC_POINT_add(group, r, r,
                         ↪  val_sub[i][digit >> 1], ctx)) goto
                         ↪  err;
696                 }
697             }
698         }
699     }
700 }
```

Let us call the **for** loop in line 656 as the *outer* **for** loop. The variable `maxlen` represents the maximum length among $w\mathrm{NAF}(k_1)$ and $w\mathrm{NAF}(k_2)$. The value of the variable `r_is_at_infinity` takes the value of 1, only for the first iteration of the outer `for` loop. Let us call the `for` loop in line 663 as the *inner* `for` loop. Since we are decomposing the scalar multiplier $k$ into two scalars, the value of the variable `totalnum` (in line 663) is 2. Let $k_1$ and $k_2$ be the decomposed scalars. In the Listing 3.2, $w\mathrm{NAF}(k_2)$ is stored in the array `wNAF[0]`, and $w\mathrm{NAF}(k_1)$ is stored in the array `wNAF[1]`.

The inner `for` loop has two iterations for each outer `for` loop iteration. The `if` statement at line 665 is executed twice for each iteration of the outer `for` loop. The `if` statement at line 670 is executed if the $w$NAF length of the decomposed scalar is greater than the variable `k` (note that `k` corresponds to the outer `for` loop variable).

The `EC_POINT_add` function calls the `ec_GFp_simple_add` function. Let us represent this *group add* operation by *add*. Similarly, the `EC_POINT_dbl` function calls the `ec_GFp_simple_dbl` function. Let us represent this *group double* operation by *dbl*.

The machine code created from source lines 656 to 700 covers the virtual memory address range `0x805becb` to `0x805bff0`. This memory range spans across five cache lines (each cache line is 64 bytes). We designate the cache lines by $A$, $B$, $C$, $D$, and $E$. The mapping of source code lines are depicted in Figure 3.2.

For each of the decomposed scalars (which are stored in their modified $w$NAF form), the `if` statement at line 695 (in Listing 3.2) is executed for each non-zero digit.

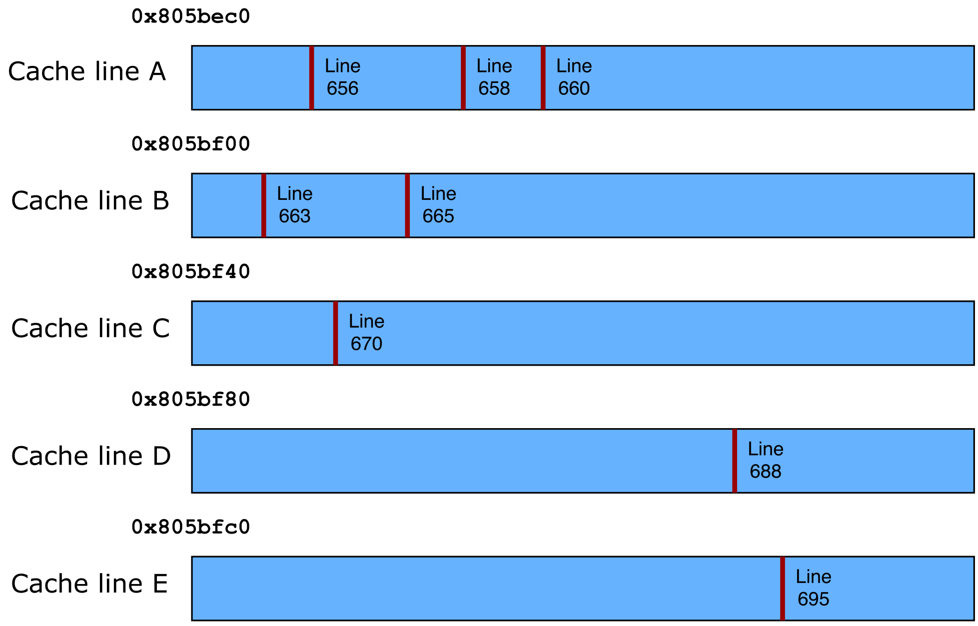



Figure 3.2: Mapping of source code to memory

## 3.3 The Flush+Reload Attack

The Flush+Reload attack [4] is a cache side-channel attack that exploits a property of Intel implementation of x86 and x86_64 processor architectures [2]. The attack also relies on shared memory pages between the spy process and the victim process. The spy process can ensure that a specific memory line is evicted from the entire cache hierarchy. The attack consists of a specified number of rounds. Each round of the attack consists of three phases: Flush, Wait and Reload.

- **Flush phase:** During this phase, the memory line that is being monitored is evicted from the cache hierarchy with the `clflush` instruction.
- **Wait phase:** During this phase, the spy program waits for a certain number of clock cycles mentioned by the attacker. During this wait time, the victim process may access the memory line which was evicted during the Flush phase of the round.
- **Reload phase:** During this phase, the spy process accesses the memory line and measures the access time. The victim process may or may not have accessed the memory line during the Wait phase of the round. If the memory line has been accessed by the victim process, then it will be

available in the cache and hence time measured by the spy process will be short. Otherwise, the memory line access by the spy process takes a long time.

**Algorithm 3.1:** FLUSH+RELOAD algorithm [2]

`probe()`<br>
**Input**: The probed address *adrs*.<br>
**Output**: `true` if the victim process accessed the address line during the WAIT phase.<br>
evict(*adrs*)<br>
wait()<br>
*time* ← current_time()<br>
*tmp* ← read(*adrs*)<br>
*readTime* ← current_time() - *time*<br>
**return** *readTime* < **threshold**

The FLUSH+RELOAD attack evicts selected memory lines from all the caches in the hierarchy and measures the time taken by the processor to reload the memory lines [4]. The attack thus detects the use of specific memory lines by a certain computation.

### 3.3.1 Implementation

To implement the spy program, we referred the work of Yarom and Falkner [4]. We also referred to an implementation of the FLUSH+RELOAD attack available in github [28]. In our implementation, we made necessary modifications to the available implementation.

The spy process uses the FLUSH+RELOAD technique to monitor the execution of the `if` statement branches in line 660 and line 695. The time taken to complete the RELOAD phase in each round of the attack reveals whether the memory line containing the `if` statement has been accessed by the glv process during the WAIT phase of that round. This information reveals whether the $w$NAF form of a scalar has a zero digit or a non-zero digit.

The following factors [29] influence the applicability of the FLUSH+RELOAD attack on ECDSA signature generation process:

- The mapping of source code of the glv program to memory lines.
- The sequence of accesses to these memory lines by the glv process.
- The ability of the spy process to accurately capture the sequences.

### 3.3.2 Processor Optimizations

The Intel Sandy Bride architecture implements certain processor optimizations which can potentially affect our experiment if certain mitigation strategies are not employed. We present such optimizations and strategies used to mitigate their effect on our experiment:

- **Speculative Execution:** When the processor uses speculative execution [30], both arms of a conditional branch are followed before evaluating the condition. When the branch condition is evaluated, the processor utilizes the precomputation corresponding to the correct arm of the conditional branch. The precomputation corresponding to the other arm of the branch is disposed.

  In our case, the processor may start evaluating line 690 and line 695 before evaluating line 688. This might affect our experiment if we had placed probes on both line 690 and line 695. But, we do not probe line 690. Thus, our choice of probe lines mitigates the effect of Speculative Execution on our experiment.

- **Adjacent Cache Line Prefetch:** The hardware prefetching mechanism in the processor employs adjacent cache line prefetching (also known as spatial prefetching) [31]. The spatial prefetcher strives to complete every cache line fetched with the pair line that completes it to a 128-byte aligned chunk [25].

  Consider two cache lines that form a pair of 128-byte aligned chunk. Probing one of the cache lines of the pair triggers a prefetch of the other cache line. For example, when a cache miss occurs on the memory line $A$, the spatial prefetching optimization may cause the prefetching of memory line $B$ along with fetching memory line $A$ into the cache. This implies that the spy process cannot be used to probe two cache lines that form a pair.

  In our experiment, we probe line 660 and line 695. As can be seen from Figure 3.2, these lines do not form a 128-byte aligned chunk.

### 3.3.3 Limitations of the attack

The following limitations of the Flush+Reload attack affect the ability of the attack:

- **Memory access overlap:** The attack may miss the capture of a memory access if the victim's access to the memory overlaps with the Reload phase. Figure 3.3 depicts the cases which cause a memory access overlap.

- **Temporal resolution:** The Flush+Reload attack can observe the order of memory accesses only if they are sufficiently separated in time [29]. This is due to the limited temporal resolution of the attack.

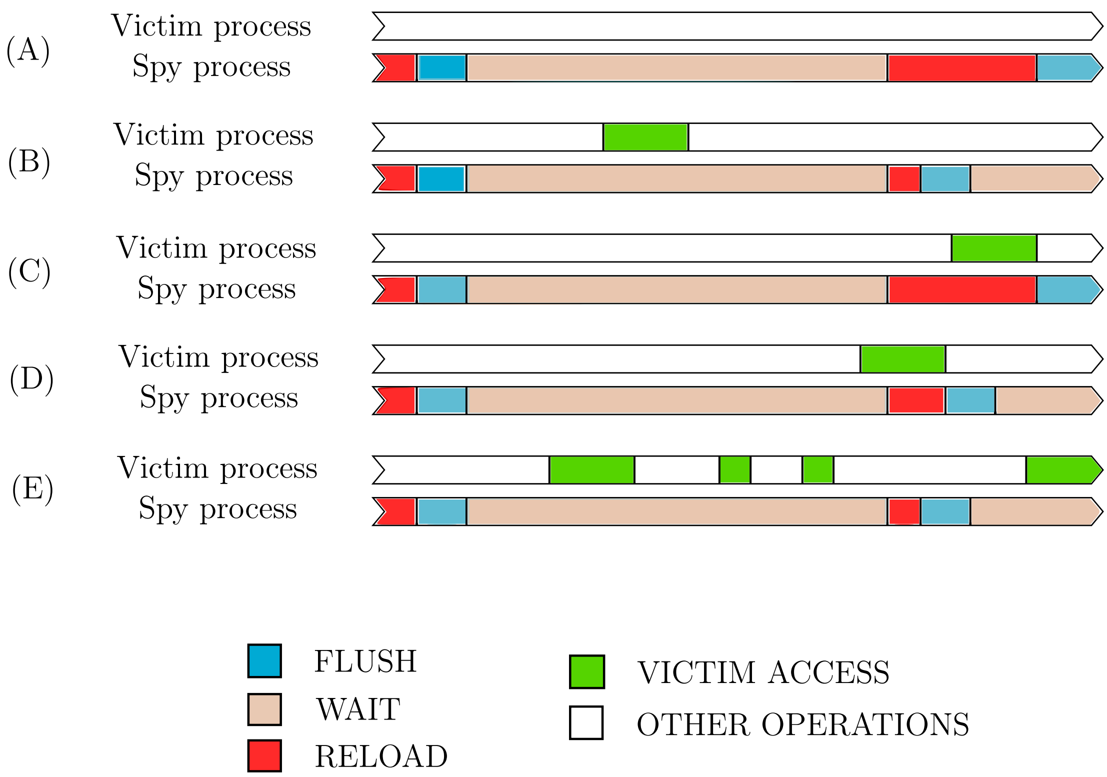



Figure 3.3: The FLUSH+RELOAD attack [4]

Possible cases in single round of the FLUSH+RELOAD attack.
(A) - Victim process does not access the probed memory.
(B) - Victim process accesses the probed memory.
(C), (D) - Memory access overlap.
(E) - Few memory accesses are lost due to insufficient attack resolution.

## 3.4 Discussion of attack parameters

In each round of the attack, the time taken by the spy process to complete the RELOAD phase for that particular round determines whether the memory was accessed by the victim process during the WAIT phase of the round. The following are the factors that are affected by the duration of the WAIT phase:

- **Memory access overlaps:** When the memory access by the victim process overlaps the RELOAD phase, the spy process cannot capture the memory access [4]. The probability of an overlap depends on the duration of the WAIT phase. An increase in the duration of the WAIT phase decreases the probability of a memory overlap.

- **Attack resolution:** The spy process cannot distinguish between multiple accesses to the same memory line during a round of the FLUSH+RELOAD attack. The resolution of the attack corresponds to the ability of the spy process to capture such memory accesses. The attack resolution is inversely proportional to the duration of the WAIT phase. Thus, increasing the duration of the WAIT phase reduces the resolution of the attack. Figure 3.3 depicts the case where memory accesses are lost due to insufficient attack resolution.

Due to the limited temporal resolution of the attack, the spy process also cannot determine the order of memory accesses to different memory lines that occur in a round of the attack.

The duration of the WAIT phase can be increased in order to reduce the number of memory overlaps. But, increasing the duration of the WAIT phase increases the probability of missing a memory access due to decreased attack resolution.

Hence, choosing the length of the WAIT phase presents a tradeoff between high attack resolution and reduced number of memory access overlaps [29]. Thus we need to carefully calibrate the length of the WAIT phase so that victim accesses are not lost.

## 3.5 Heuristics for tuning WAIT phase duration

Consider the calculation of $k_1P + k_2Q$ by the interleaving method according to Algorithm 2.2. The following observations are noteworthy:

- Let $l_1$ and $l_2$ denote the lengths of $w$NAF($k_1$) and $w$NAF($k_2$) respectively. Let $max\{l_1, l_2\}$ denote the maximum among $\{l_1, l_2\}$. The number of double operations involved in the calculation of $k_1P+k_2Q$ is equal to $max\{l_1, l_2\}-1$.

- In the calculation of $k_1P + k_2Q$, each non-zero digit in the modified $w$NAF form of a decomposed scalar corresponds to an *add* operation.

The bitlength of the random nonce $k$ is 256. In a balanced length-two representation, $k_1$ and $k_2$ are approximately the same length. When the bitlength of $k$ is 256, then the bitlength of $k_1 \approx 128$ and the bitlength of $k_2 \approx 128$. Since the length of $w\text{NAF}(k_1)$ and $w\text{NAF}(k_2)$ may be atmost one digit longer than their respective binary representations, we can say that $l_1 \approx 128$ and $l_2 \approx 128$. Thus, the number of double operations in the interleaved multiplication is also approximately equal to 128.

We calibrate the WAIT phase duration to ensure that there is an empty slot during the execution of an *add* or a *dbl* operation. This improves the resolution of the attack and allows the spy process to correctly distinguish consecutive doubles [2]. We notice that for a WAIT phase of around 2200 clock cycles, the spy records approximately 128 *dbl* operations consistently across 500 instances. When compared to the ground truth, we observe that the spy process is highly accurate in capturing the *add* and *dbl* operations corresponding to the least significant bits.

## 3.6 Attacking OpenSSL ECDSA

By observing the sequence of memory accesses through the spy process, we can retrieve sensitive information about the digits of the scalars $w\text{NAF}(k_2)$ and $w\text{NAF}(k_1)$. For elucidatory purpose, we categorize the digits of the modified $w$NAF form of the decomposed scalars as follows:

1. **Most Significant Digits:** The most significant digits of $w\text{NAF}(k_2)$ and $w\text{NAF}(k_1)$ correspond to the initial iterations of the outer `for` loop. In order to obtain the values of $k_2$ and $k_1$, we performed a balanced decomposition of the scalar multiplier $k$. Though the lengths of $w\text{NAF}(k_2)$ and $w\text{NAF}(k_1)$ are close to each other, in most of the cases the lengths may not be actually equal to each other. Due to this specific reason, it is evident that the information obtained from the first few iterations of the outer `for` loop is ambiguous in nature.

   In a similar experiment, Yarom and Benger [29] provide empirical evidence that the missed memory access captures tend to cluster towards the most significant digits. We noticed a similar trend in our experiment where the captured information about the most significant digits of $k_1$ and $k_2$ is undetailed and indistinguishable.

2. **Intermediate Digits:** The intermediate digits that have a value of zero can be identified through the information captured by the attack. In spite of recognizing intermediate zero digits, it is unfeasible to accurately trace their actual position in the modified $w$NAF form of the scalar.

3. **Least Significant Digits:** A least significant digit can either be negative, positive, or zero. We briefly discuss the cases pertaining to a least significant

digit in the modified $w$NAF form of each decomposed scalar:

- **Negative Digit:** The variable `is_neg` which is defined in line 668 is responsible for changing the program flow when a negative digit is encountered in the modified $w$NAF form of the decomposed scalar. The memory accesses in an iteration of the outer `for` loop do not follow a predictable pattern in case of a negative digit (see Listing 3.2). The `EC_POINT_invert` function is called only if the value of the variable `is_neg` is equal to the value of the variable `r_is_inverted`.
- **Positive Digit:** When the digit is positive, the memory accesses happen in a fixed pattern. But detecting the presence of a positive digit does not reveal the value of the digit. The positive digit may be any digit in the set $\{1, 3, 5, 7\}$.
- **Zero Digit:** When the digit is zero, the memory accesses happen in a fixed pattern. There is a direct correspondence between the least significant zeroes in the $w$NAF form of a scalar and the least significant zeroes in its binary form. Thus, capturing trailing zeroes in the $w$NAF form of a scalar reveals the number of trailing zeroes in the binary form of the scalar.

Our attack detects the presence of a non-zero digit with a very high probability. The absence of a non-zero digit implies the presence of a zero digit. Thus, the presence of a zero digit can be detected with very high probability. We focus on observing the sequence of memory accesses which reveal the least significant zero digits in the $w$NAF form of the decomposed scalars.

*Henceforth, our discussion will be only regarding the last few iterations of the outer `for` loop, as these iterations correspond to the least significant digits of $wNAF(k_2)$ and $wNAF(k_1)$.*

- For a zero digit, the *dbl* function is called in the outer `for` loop and no function is called in the inner `for` loop. Hence, the memory line $A$ is always accessed for every zero digit in the $w$NAF form of the decomposed scalar.
- The memory line $E$ is always accessed for every non-zero digit in the modified $w$NAF form of the decomposed scalar. This memory line is never accessed for a zero digit.

Memory lines that contain function calls are accessed by the process both when the function is called and when the function returns [2]. For example, memory line $A$ (corresponding to cache line $A$) is accessed when the *dbl* function is called as well as when the function returns. Similarly, the memory line E (corresponding to cache line $E$) is accessed when the *add* function is called and when the function returns. Probing such memory lines reduces the chance of missing accesses that occur due to memory access overlaps [2]. Hence, we probe the memory line $A$ (containing call to the *dbl* function; line 660 of Listing 3.2) and the memory line E (containing call to the *add* function; line 695 of Listing 3.2). Since we built OpenSSL with debugging symbols, we can find the memory lines containing the call sites. We used the `gdb break` command to find the address of relevant memory lines. The spy process monitors the cache hits/misses and allows us to determine whether the line 695 is executed in a particular round of the attack.

We retrieve the last values of $k_i^j$ processed by Algorithm 2.2. In our experiment, we have $i = \{1, 2\}$ and thus the decomposed scalars $k_1$ and $k_2$. Figure 4.9 shows few instances where sensitive information has been obtained from the attack.

## 3.7 Experimental Validity

We draw conclusions from the results yielded by the experiment. Valid conclusions are dependent on valid results. A valid experimental design is necessary for obtaining valid results. Hence we validate the experimental design by addressing the threats to validity.

$$Experiment \rightarrow Results \rightarrow Conclusions$$

### 3.7.1 Statistical Conclusion Validity

We ensure that we have employed adequate sampling procedures and appropriate statistical tests.

**Experiment:** The independent variable is the 256 bit random nonce $k$. Our sample set consists of a random sample of 500 nonces. We captured 500 signature generation instances, each corresponding to one of the randomly generated cryptographically secure nonce $k$.

**Notation:** Henceforth, we adopt the following notation:

- Let $LSB_b(n)$ denote the $b$ least significant bits of the binary representation of $n$. Let $LSB_b(n) = 0$ denote that the binary representation of number $n$ has $b$ least significant zero bits.

- Let $LSB_b(k_1k_2) = 0$ denote that $LSB_b(k_1) = 0$, $LSB_b(k_2) = 0$ and at most one of $\{LSB_{b+1}(k_1), LSB_{b+1}(k_2)\}$ is equal to 0.

- Let $\tau_t$ denote the set containing all the possible values of the independent variable $k$. Let $\nu_t \subset \tau_t$ denote the set containing values of $k$ for which $LSB_b(k_1k_2) = 0$, for all admissible values of positive integer $b$.

- Let $\tau$ denote the set of all values of the independent variable $k$ considered in the experiment. Let $\nu = \tau \cap \nu_t$.

- Let $\nu_b = \{k \mid k \in \nu, LSB_b(k_1k_2) = 0\}$, where $b$ is a positive integer. Note that $\nu_1 \cup \nu_2 \cup \nu_3 \cup \cdots = \nu$ and $\nu_i \cap \nu_j = \emptyset$, for $i, j \in \{1, 2, 3, \ldots\}$, $i \neq j$.
- Let $n(A)$ denote the cardinality of the set $A$.

*Ground Truth:* Of the 500 randomly generated nonces, we noticed that 108 of the nonces $\in \nu_t$. These nonces produced decomposed scalars with trailing zero bits. The distribution of nonces $\in \nu_t$ in the sample set is depicted in Figure 3.5.

$$n(\tau) = 500, \text{ and } n(\nu) = 108$$

Of the 500 nonces, we identified output plots corresponding to 99 nonces that display trailing zeroes in both $k_1$ and $k_2$.

- 69 plots reveal the information that the corresponding decomposed scalars $k_1$ and $k_2$ have one trailing zero bit.
- 24 plots reveal the information that the corresponding decomposed scalars $k_1$ and $k_2$ have two trailing zero bits.
- 6 plots reveal the information that the corresponding decomposed scalars $k_1$ and $k_2$ have three trailing zero bits.

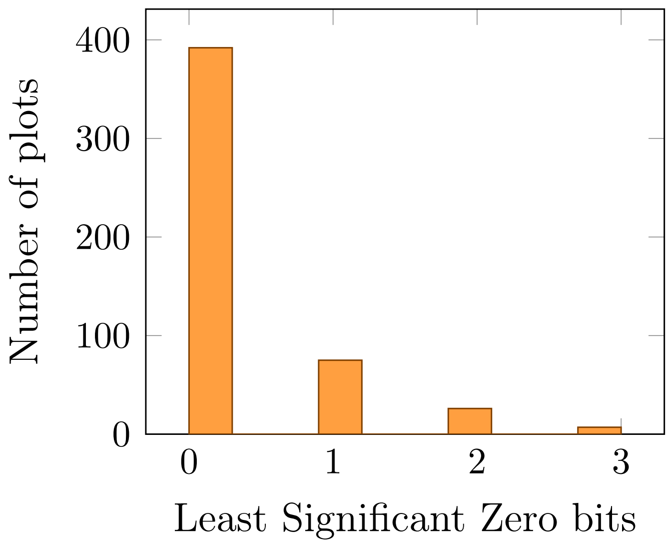



Figure 3.4: A frequency histogram

The distribution of the 108 nonces in the set $\nu$ is depicted by the histogram. $n(\nu_1) = 75, n(\nu_2) = 26, n(\nu_3) = 7.$

Thus, out of 500 randomly generated nonces, 108 cause identifiable leakage. Henceforth, we refer to any nonce in the set $\nu_t$ as a 'leaky' nonce. The nonces in

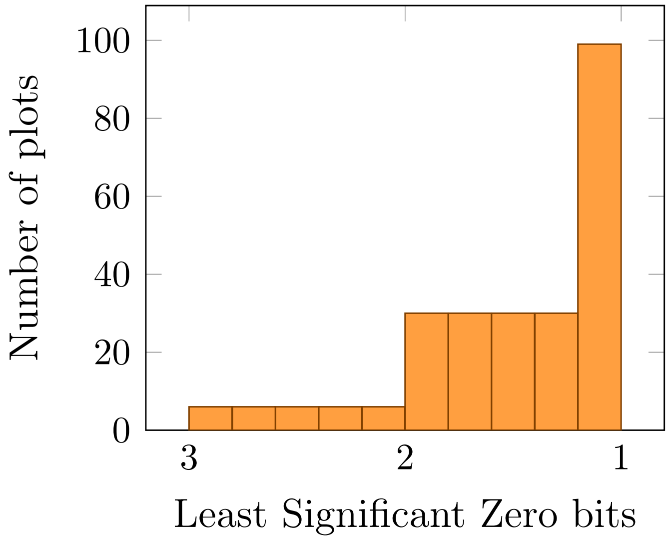



Figure 3.5: A cumulative frequency histogram

The distribution of the identified 99 nonces is depicted by the cumulative frequency histogram. Let $\nu'$ be the set containing these 99 nonces.
$n(\nu'_1) = 69, n(\nu'_2) = 24, n(\nu'_3) = 6.$

the set $\nu_1$ reveal only 1 bit of the decomposed scalars. The sets $\nu_2$, $\nu_3$ contains nonces which cause significant leakage (2 bits, 3 bits respectively). We can determine the proportion of 'leaky' nonces in the set $\tau$ by the following.

$$\frac{n(\nu)}{n(\tau)} = \frac{108}{500} = 0.216$$

From the experiment, we can say that with a 95% confidence level that for a randomly chosen nonce, there is 21.6% chance that it is 'leaky'. The confidence level is 3.61%.

**Binomial Experiment:** An experiment often consists of repeated trials, where each trial may result in two possible outcomes that may be labeled *success* or *failure* [32].

**Definition 3.1.** *[32] A Binomial experiment (also known as Bernoulli process) is an experiment that satisfies the following properties:*

1. *The experiment consists of a fixed number of repeated trials.*
2. *The outcome of each trial may be classified as a success or a failure.*
3. *The probability of success (p) remains constant across all the trials.*
4. *The outcome of a trial does not affect the outcome of subsequent trials i.e., the repeated trials are independent.*

*Statistical Terminology:*

- **Bernoulli trial:** In a Bernoulli process, each of the series of independent repeated trials is called a Bernoulli trial [32].

  *Note:* A Bernoulli trial is also known as a *Binomial trial.*

- **Binomial random variable:** The number of successes ($X$) in $n$ Bernoulli trials is called a Binomial random variable [32].

- **Binomial distribution:** The probability distribution of the Binomial random variable is called the Binomial distribution [32].

- **Binomial test:** According to the Cambridge Dictionary of statistics [33], a Binomial test is a procedure for making inferences about the probability of success ($p$), in a series of independent repeated Bernoulli trials.

- **Clopper-Pearson interval:** The Clopper-Pearson interval is a confidence interval for the probability of success in a series of independent repeated Bernoulli trials [33].

An instance of the FLUSH+RELOAD attack is equivalent to a Bernoulli trail with *success* and *failure* defined as follows:

- *success:* The plotted graph displays exactly $b$ trailing double operations, where $LSB_b(k_1k_2) = 0$.

- *failure:* The plotted graph does not display exactly $b$ trailing double operations, where $LSB_b(k_1k_2) = 0$.

In order to measure the success rate of the FLUSH+RELOAD attack in our experimental setup (i.e., the ability of the attack in identifying the leakage), we frame the following statistical problem [34]:
*Aim of the problem:* To find the value of the success probability $p$.
*Required observations:* Results of $n$ Bernoulli trials; $s$ successes and $n - s$ failures.

We perform a Binomial experiment with 400 repeated trials. Each trial results in either *success* or *failure.* To perform this experiment, we choose an element $k$ uniformly and at random from the set $\nu_3$. We notice that for the randomly chosen $k$, $k_1$ has 3 trailing zero bits while $k_2$ has 4 trailing zero bits.

$$LSB_3(k_1) = 0 \text{ and } LSB_4(k_2) = 0.$$

$$\text{Therefore, } LSB_3(k_1k_2) = 0.$$

The outcomes of 400 trials contained 336 successes. The 95% confidence interval calculated from the Clopper-Pearson method is as follows:

$$0.80033 \leq p \leq 0.87455$$

This gives us an estimation of the effectiveness of the attack in capturing sensitive information when 'leaky' nonces are involved in the signature generation process.

### 3.7.2 Context Validity

We build the patched version of `openssl-1.0.1l` with debugging symbols. In a typical scenario, these debugging symbols are not available to the attacker. But a determined attacker could use reverse engineering [35] techniques in the absence of the availability of these symbols.

Chapter 4

# Results

The spy process outputs the collected side-channel information to a plain text document (`out.txt`). We ensure that the side-channel information is output to the document only after the entire information is collected. This is necessary, so that the file output operations do not interfere with the probing. The `gnuplot` script (Appendix E) outputs the plots from the available side-channel information.

## 4.1 Processing the captured information

Listing 4.1 presents a fragment of the `out.txt` file for a random instance of the glv program. In this section, we discuss the steps involved in processing the captured side-channel information (`out.txt`) to derive output plots (Figures 4.1 to 4.8).

$$\underbrace{\textbf{\textit{Captured Information}}}_{\texttt{out.txt}} \longrightarrow \underbrace{\textbf{\textit{Initial Plot}}}_{\text{Figures 4.1,4.2}} \longrightarrow \underbrace{\textbf{\textit{Threshold Plot}}}_{\text{Figures 4.3,4.4,4.5}} \longrightarrow \underbrace{\textbf{\textit{Processed Plot}}}_{\text{Figures 4.6,4.7,4.8}}$$

For each instance of the glv program, the spy process accomplishes 50,000 rounds of the Flush+Reload attack. The information captured in each round spans $r$ rows of `out.txt`, where $r$ is the number of address lines being probed. Since we probe the lines 660 and 695 (Listing 3.2), the value of $r$ is 2. Therefore, the file `out.txt` consists of 100,000 rows.

The fragment of `out.txt` presented in Listing 4.1 shows the captured information corresponding to the first 10 rounds of the attack on an instance of the glv program. The addresses of the line 660 (containing call to the *dbl* function) and the line 695 (containing call to the *add* function) are labeled *0* and *1* respectively.

We describe in detail the steps employed to derive output plots (Figures 4.1 to 4.8) from the `out.txt` file.

1. **Captured Information:** The side-channel information captured from a single round of the attack corresponds to two rows in the file `out.txt`. Each row contains three fields delimited by whitespace characters. The first field contains the round number (time slot number) of the attack. The second field contains the label of the probed address line. The third field contains

the number of clock cycles taken by the spy process to load the corresponding address line. This value is determined in the Reload phase of each round of the attack.

2. **Initial Plot:** The `gnuplot` utility interprets the file row-wise and plots points according to the plotting script. These points constitute the plot depicted in Figure 4.1. Each plotted point corresponds to a row in the file `out.txt`. Each row comprising the triplet (*round number*, *label*, *access time*) is plotted as $(x, y) = (\textit{round number}, \textit{access time})$, with the value of *label* determining the type of point ( ● or ■ ). Thus, the values on the $x-$axis correspond to the values in the first column of the file (time slot numbers) and the values on the $y-$axis correspond to the values in the third column of the file (access times in clock cycles). The symbol ● denotes the access to line 660 (containing the *dbl* operation) by the spy process (in the Reload phase), and the symbol ■ denotes its access to line 695 (containing the *add* operation).

   The Initial Plot containing the captured information is depicted in Figure 4.1. Figure 4.2 shows an enlarged view of the portion of Figure 4.1 that contains crucial side-channel information.

3. **Threshold Plot:** It is evident that the interleaved scalar multiplication phase in an instance of the glv program is discernible from the Initial Plot (Figure 4.1). Nevertheless, establishing a pre-defined threshold value across all the instances assists us in convenient graphing of the plots. The threshold value in the Flush+Reload attack is architecture dependent. In order to determine the threshold value, we follow an approach identical to the one adopted by Yarom and Falkner [4].

   We estimate the average L1 cache access time and accordingly determine the threshold value. To measure the L1 access times, we reused the code in Appendix D by removing the `clflush` instruction from it. From 100,000 measurements, we deduce that the average L1 cache access time is approximately 50 clock cycles.

   In the Flush+Reload attack, data accessed by the victim process is read from the L3 cache [4]. Therefore, the average L1 access time underestimates the probe time (recorded by the spy process in the Reload phase) for an address line that has been accessed by the victim process (during the Wait phase). Taking this into consideration, and based on the measurement results, we set the threshold value to 120 clock cycles.

   Figures 4.3, 4.4 and 4.5 collectively depict the Threshold Plot. These figures also present an expanded view of the side-channel information contained in Figure 4.2.

4. **Processed Plot:** From the discussion presented in Section 3.3, it is evident that the Flush+Reload attack detects the victim access to probed address lines. The presence of a point at $(x, y)$ in the Threshold Plot implies that the particular address line (denoted by the type of the point) was accessed by the victim during the Wait phase of the $x^{th}$ round of the attack. Hence, the information about the actual access times ($y-$coordinates) contained in the Threshold Plot is of little significance.

   We can unambiguously interpret the Threshold Plot to detect the victim accesses that occur during the Wait phases. But, it is strenuous for an analyst to interpret the results from hundreds of such plots. To visually aid the analysis of the results, we process the side-channel information contained in the Threshold Plot and generate a Processed Plot containing only the required 'binary' information (whether the probed address line was accessed by the victim or not). We 'round' the access times corresponding to the line 660 (denoted by ● ) to a constant value (50). We also 'round' the access times corresponding to the line 695 (denoted by ■ ) to another distinct constant value (40). We call the points $(0, 50)$ and $(0, 40)$ on the $y-$axis, the *reference points*. The aim of our reference points is to enable an easier visual interpretation of the results. Any pair of points on the $y-$axis which fulfill this aim are a potential alternative for the chosen reference points.

   The symbol ● denotes that the victim process has accessed line 660 (containing the *dbl* operation) during the Wait phase, and the symbol ■ denotes its access to line 695 (containing the *add* operation). The absence of a symbol in a round of the attack implies that the corresponding address line was not accessed by the victim process during the Wait phase of that round.

   Figures 4.6, 4.7 and 4.8 collectively depict the Processed Plot. Note that the reference points in these figures are $(0, 58)$ and $(0, 55)$.

Listing 4.1: First few rows of the `out.txt` file

```
# Round    Address line   Clock cycles
#------------------------------------
    0            0           22472
    0            1            384
    1            0            368
    1            1            368
    2            0            400
    2            1            392
    3            0            384
    3            1            336
    4            0            336
    4            1            360
    5            0            368
    5            1            368
    6            0            392
    6            1            384
    7            0            344
    7            1            376
    8            0            392
    8            1            368
    9            0            368
    9            1            360
```

## 4.2 Interpretation of the output plots

The trailing zeroes of the binary representation of a scalar are retained in its modified $w$NAF representation. Therefore, if the modified $w$NAF form of a scalar is found to have certain number of trailing zeroes, then its corresponding binary form has an equal number of trailing zeroes. From the attack, we identify instances of ECDSA signature generation process which contain least significant zero bits for the scalars $k_1$ and $k_2$. Figure 4.9 presents relevant fragments of the threshold plots corresponding to three successful trials of our binomial experiment. For the value of $k$ that is chosen in the binomial experiment, the corresponding values of the decomposed scalars are as follows:

$$k_1 = \texttt{0x579EDFC4407787B568D57C7EB0FD8EE8}$$

$$k_2 = \texttt{0x1AAF8BC7EB62988D3158F8003EB65330}$$

The scalar $k_1$ has 3 trailing zero bits while the scalar $k_2$ has 4 trailing zero bits; $LSB_3(k_1k_2) = 0$. This information about the decomposed scalars can be obtained

| **Sequence of operations for $\boldsymbol{k_i}$** | **LSB sequence** | **Number of trailing zeroes in $\boldsymbol{k_i}$** |
|---|---|---|
| $\ldots A$ | 1 | 0 |
| $\ldots AD$ | 10 | 1 |
| $\ldots ADD$ | 100 | 2 |
| $\ldots ADDD$ | 1000 | 3 |

Table 4.1: Preliminary interpretation of the obtained results

$A$ - *add* operation, $D$ - *dbl* operation, $i \in \{1, 2\}$.
It can be noticed that each of the identified 3 trailing *dbl* operations in the Figures 4.9a, 4.9b, and 4.9c corresponds to a *dbl* operation for $k_1$ and a *dbl* operation for $k_2$.

from our attack (Figures 4.9a, 4.9b, 4.9c). In order to find answers to our research questions, we analyze the obtained results in Chapter 5.

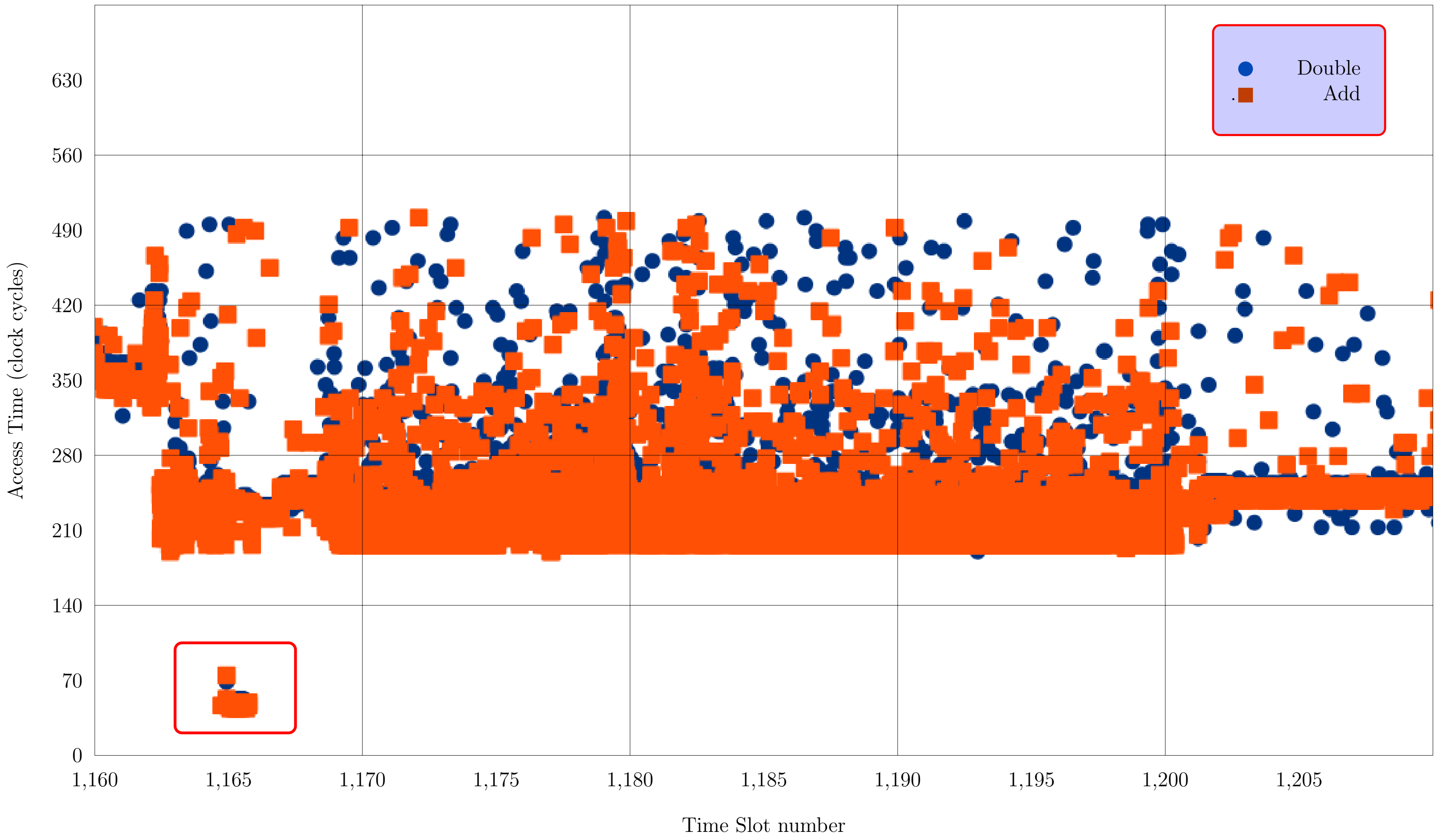



Figure 4.1: Overall view of the captured information.

The spy process runs in background with an instance of the glv program. The depicted signature generation process corresponds to one of the 500 instances of the glv program. This instance employs a nonce in the set $\tau - \nu$.

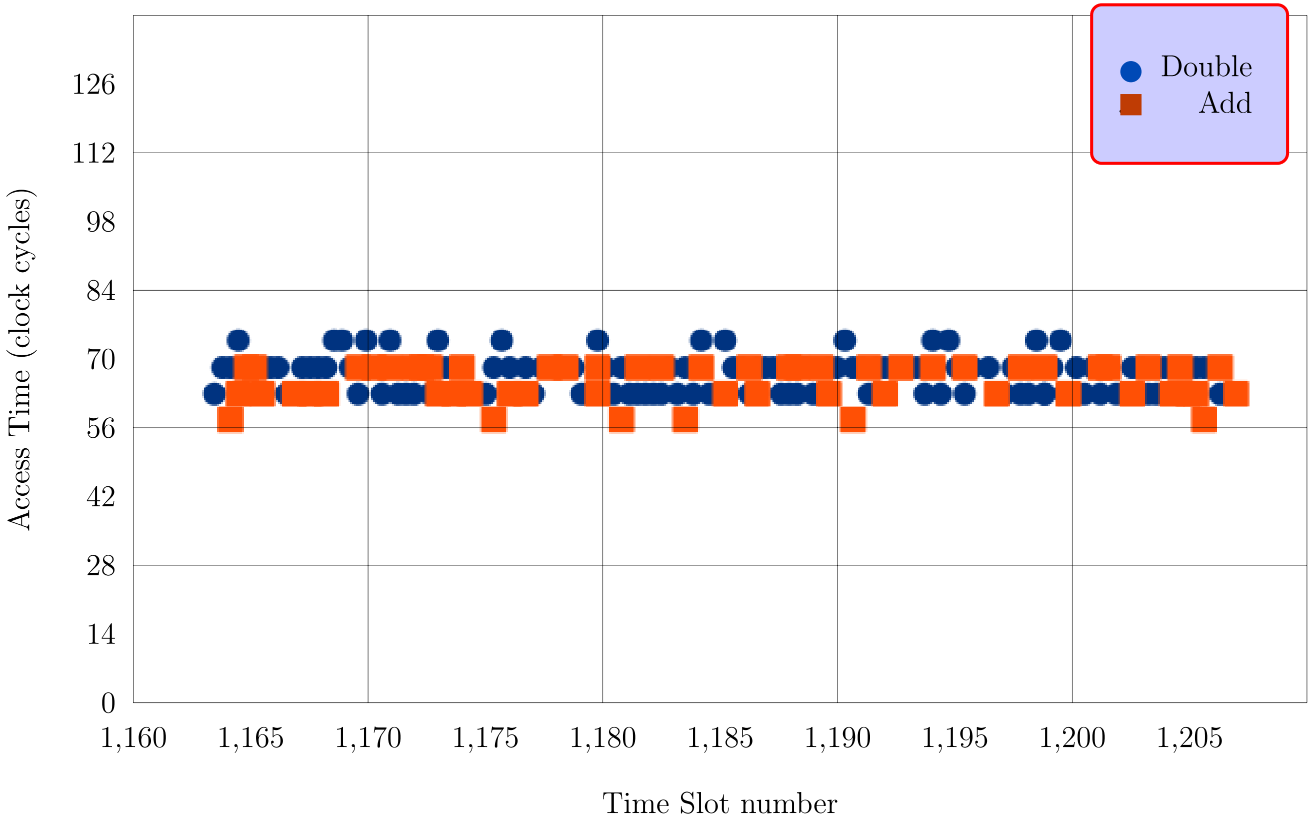



Figure 4.2: An enlarged view of the captured side-channel information.

The spy process captures side-channel information from the interleaved point multiplication of the decomposed scalars, $(k_2\lambda G + k_1 G)$.

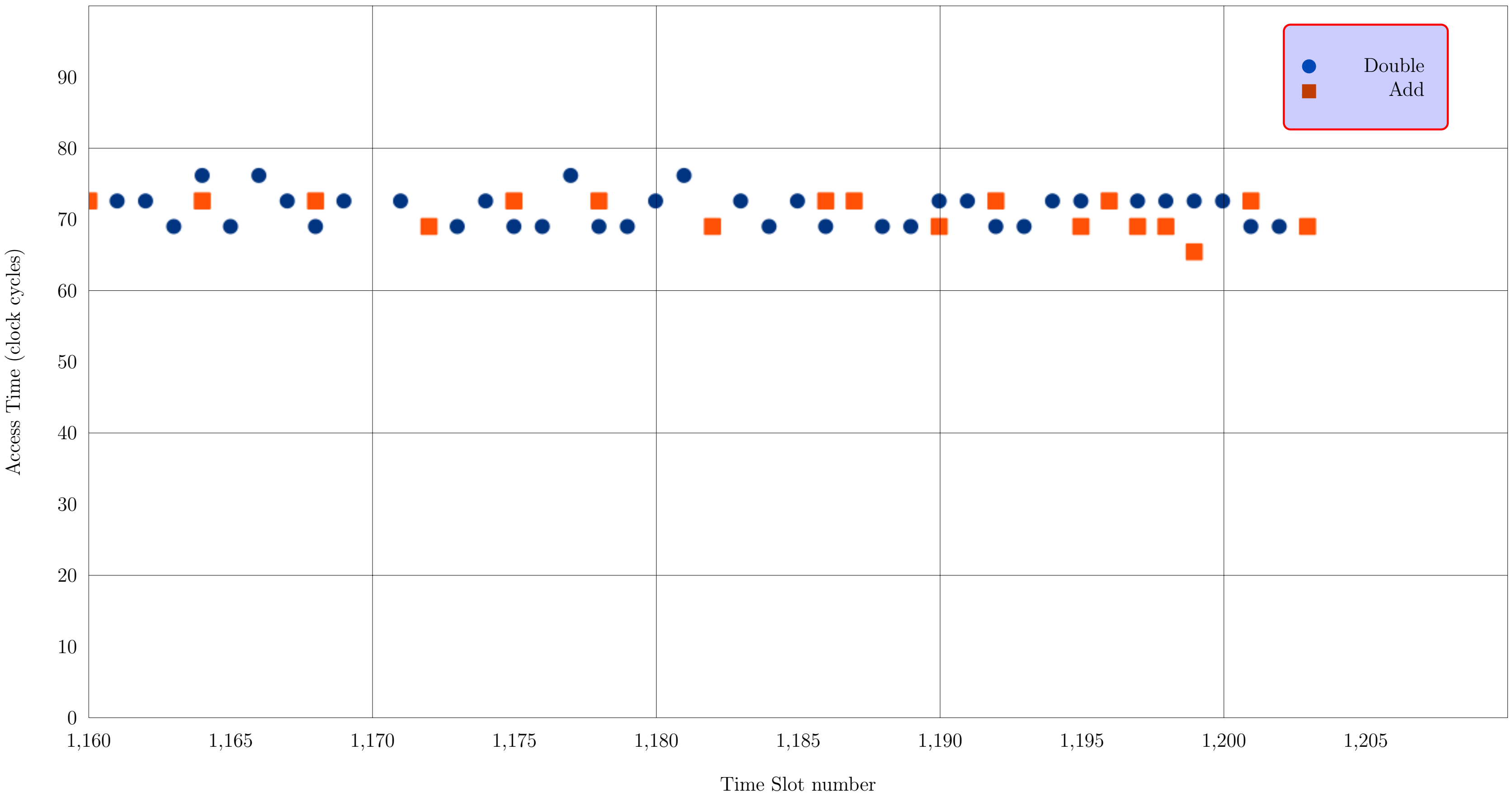



Figure 4.3: Close-up view of the captured access times for the last iterations of the outer `for` loop.
The outer `for` loop corresponds to the interleaved scalar multiplication as presented in Listing 3.2.

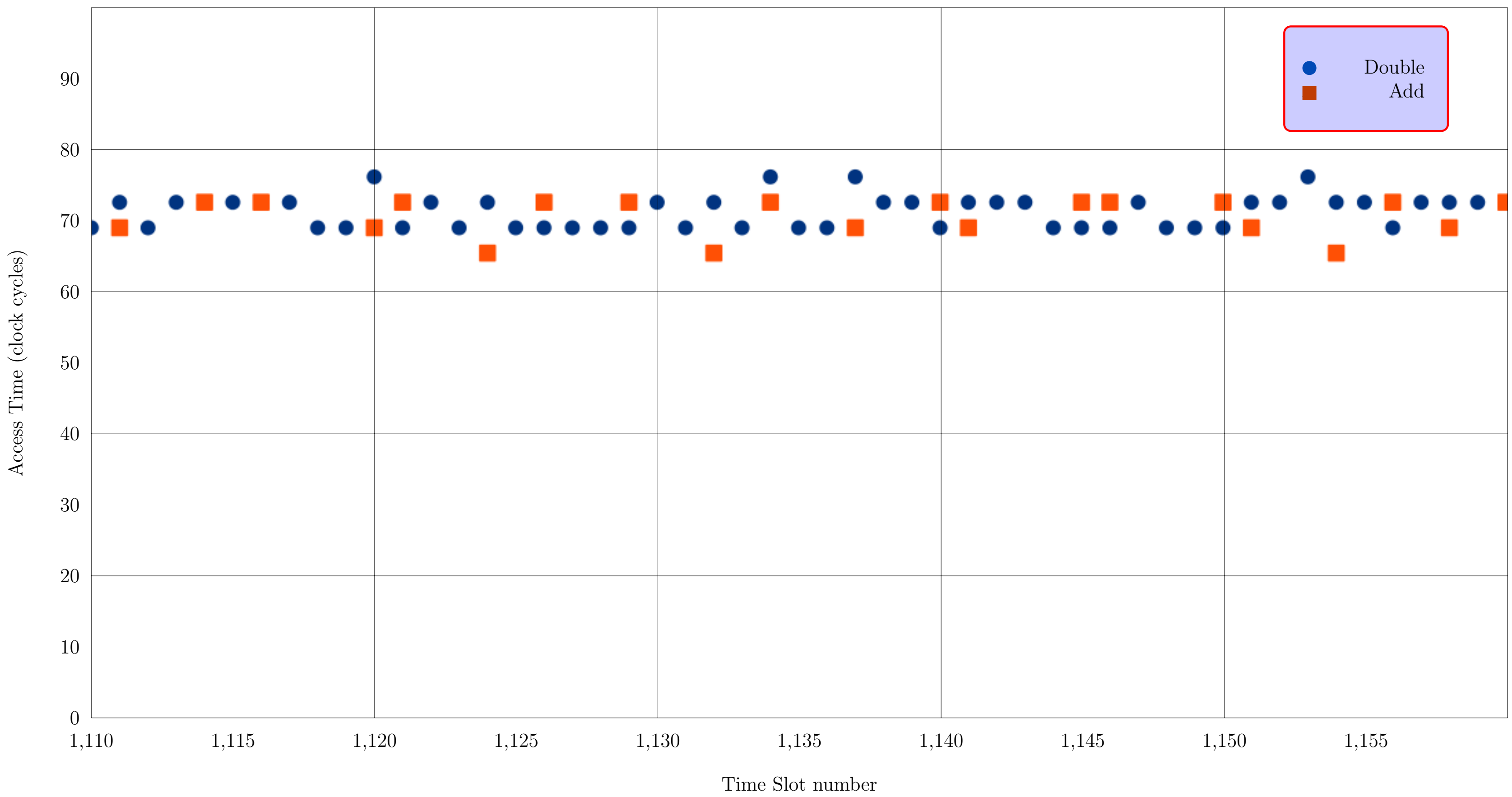



Figure 4.4: Close-up view of the captured access times for the intermediate iterations of the outer `for` loop.

The outer `for` loop corresponds to the interleaved scalar multiplication as presented in Listing 3.2.

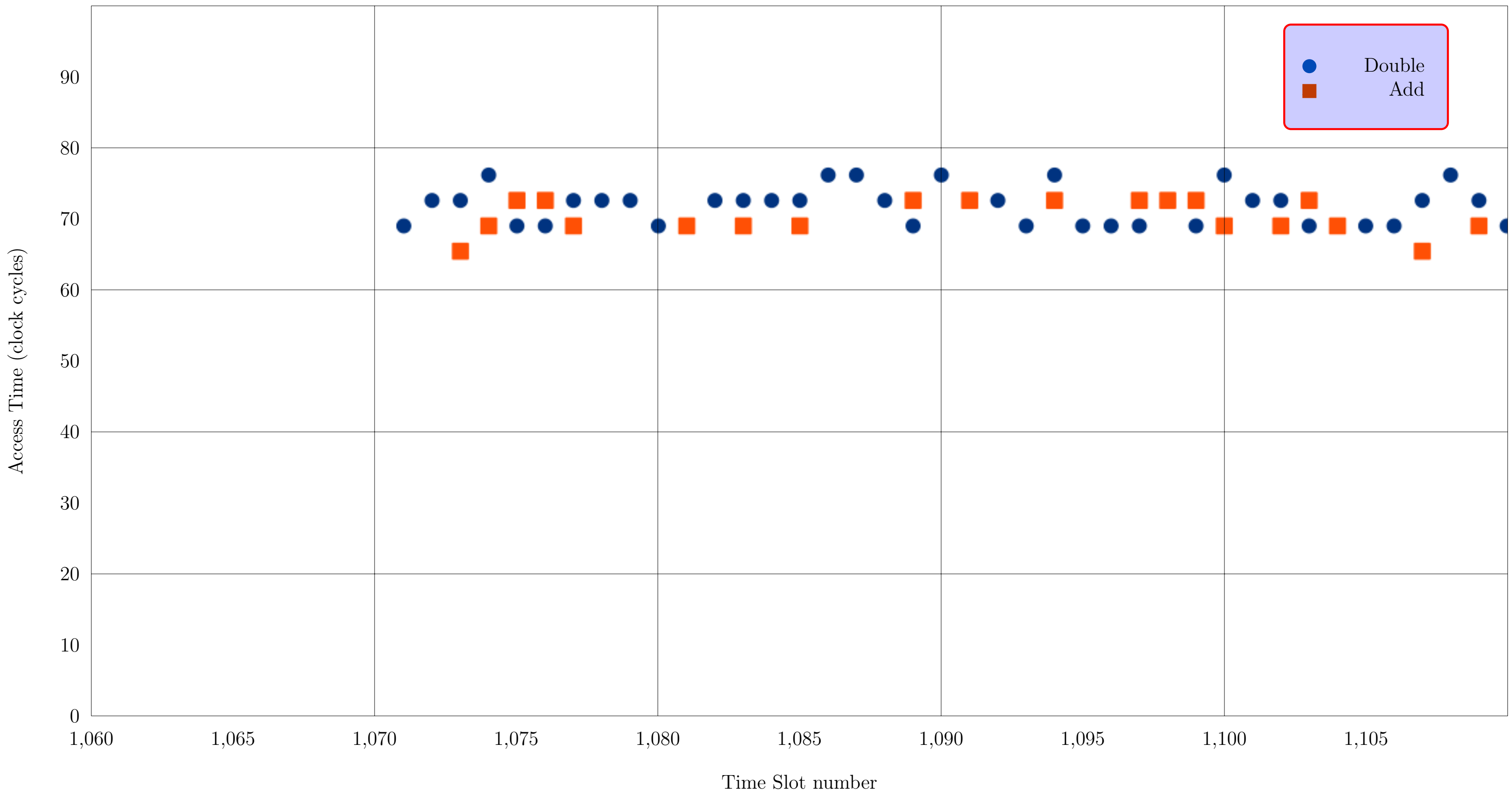



Figure 4.5: Close-up view of the captured access times for the initial iterations of the outer `for` loop.

The outer `for` loop corresponds to the interleaved scalar multiplication as presented in Listing 3.2.

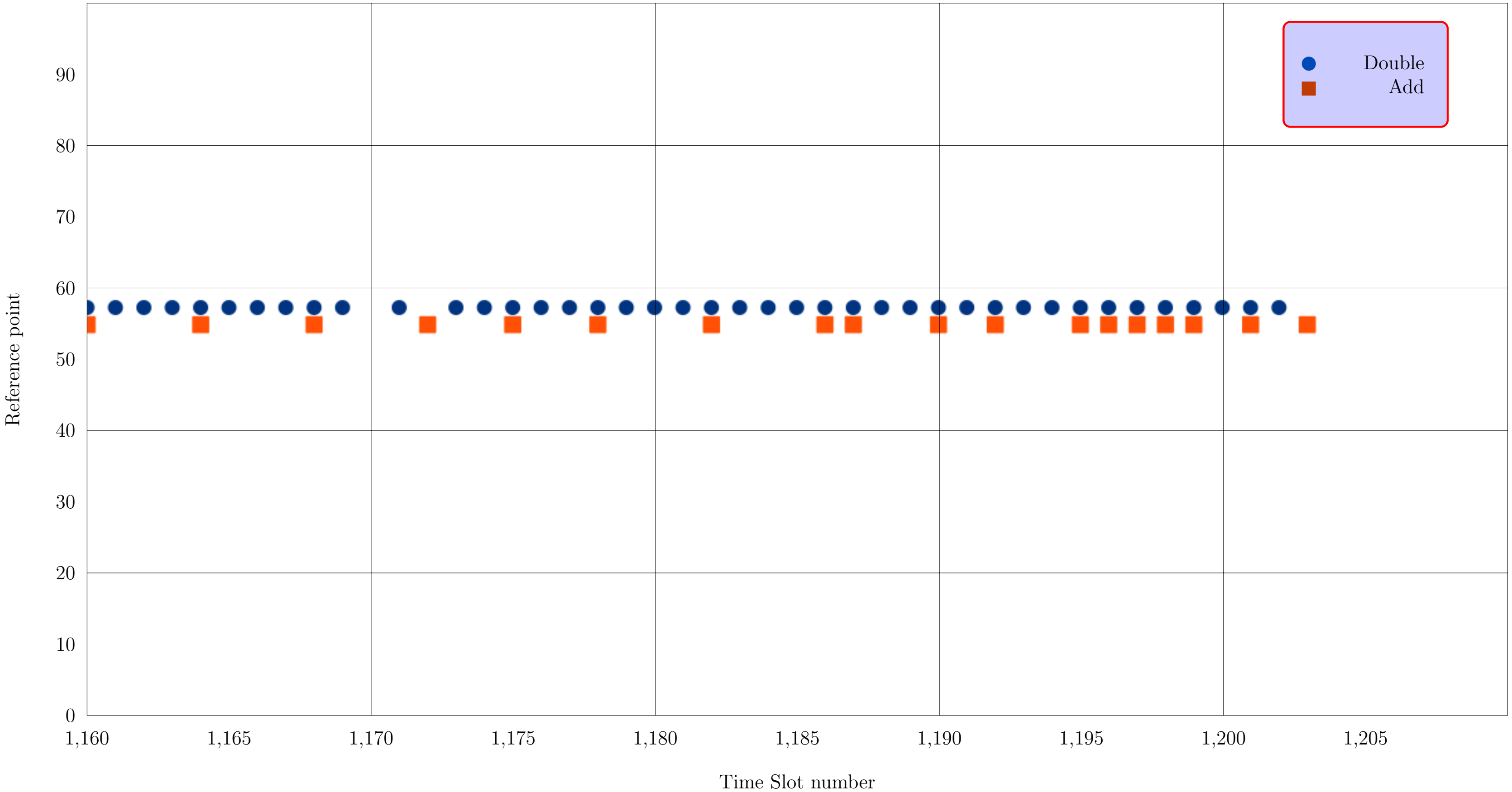



Figure 4.6: The processed access times that correspond to last iterations of the outer `for` loop.

The values on the $y$ axis are chosen to aid the interpretation of results. They are of little significance in the presented plot.

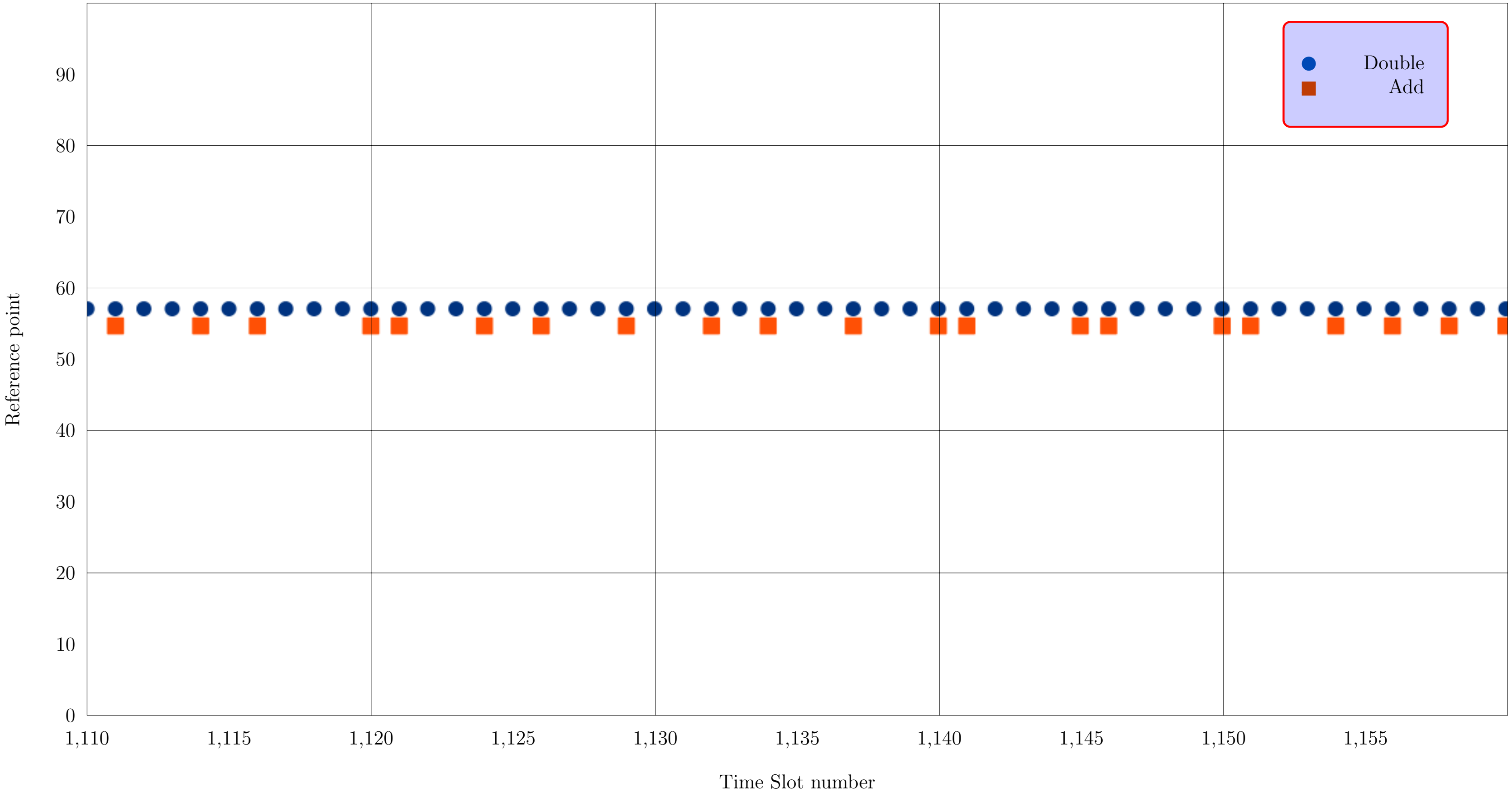



Figure 4.7: The processed access times that correspond to intermediate iterations of the outer `for` loop.

The values on the $y$ axis are chosen to aid the interpretation of results. They are of little significance in the presented plot.

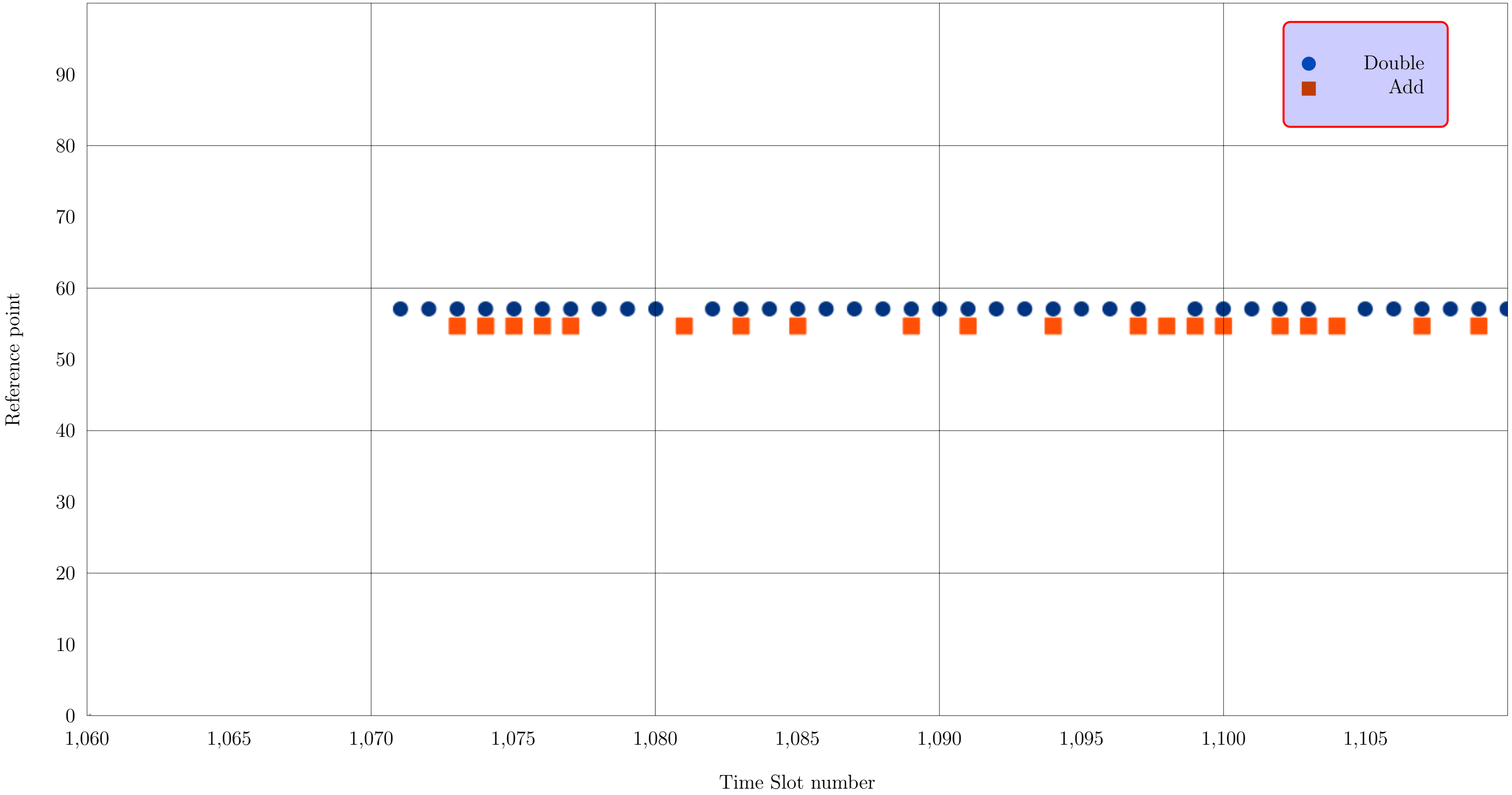
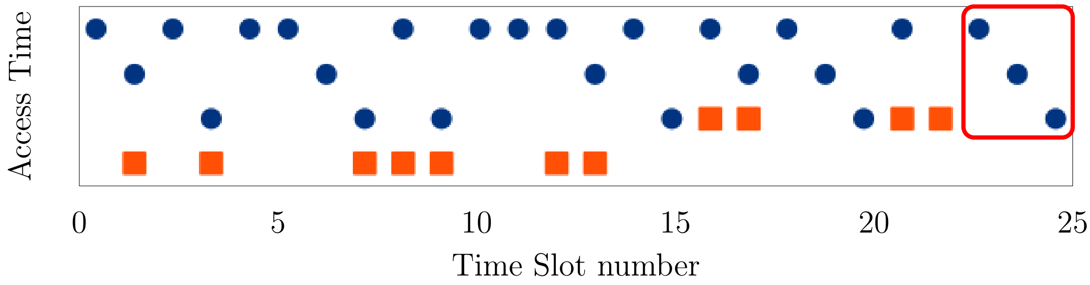
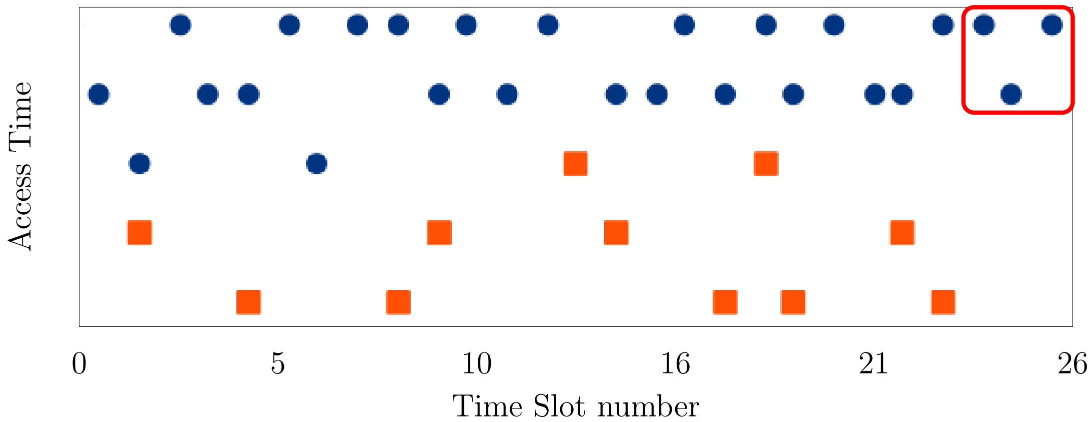



Figure 4.8: The processed access times that correspond to initial iterations of the outer `for` loop.

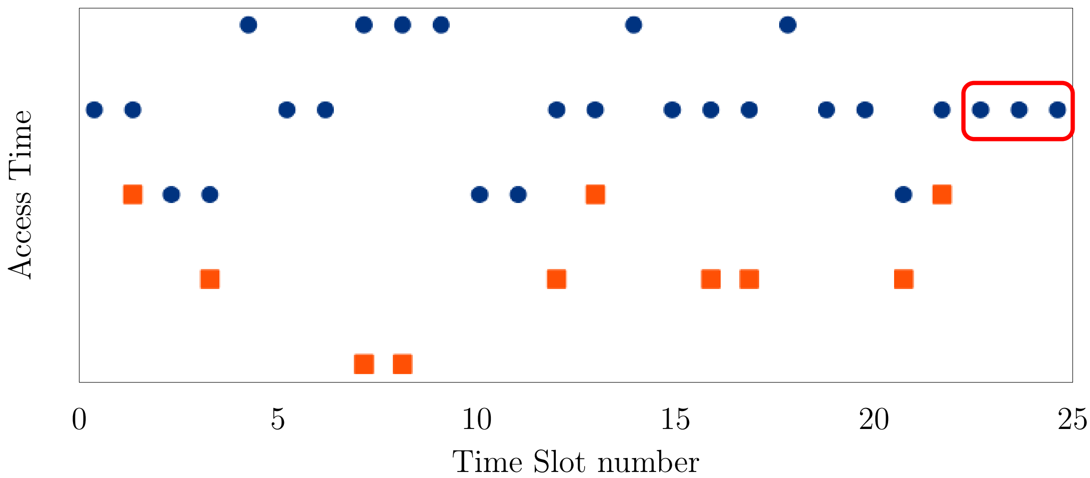
The values on the $y$ axis are chosen to aid the interpretation of results. They are of little significance in the presented plot.




(a) Information captured from an instance of ECDSA signature generation.




(b) Information captured from an instance of ECDSA signature generation.




(c) Information captured from an instance of ECDSA signature generation.

Figure 4.9: Three successful trials of the binomial experiment.

The 3 trailing zero bits of scalars $k_1$ and $k_2$ can be identified from the plots. The captured access times are less than the threshold value. The actual values of access times are of little significance.

# Chapter 5

# Analysis

The interpretation and understanding of the results of an experiment can be as important as the obtained results [24]. Hence, we perform meticulous interpretation of the results and explain their significance in the forthcoming sections.

In the ECDSA cryptosystem, the message $h$, the signature $s$, the secret key $\alpha$ and the corresponding nonce $k$ are related as shown in the Equation 5.1.

$$\boxed{\alpha = (s \cdot k - h) \cdot r^{-1}} \tag{5.1}$$

We reasonably assume that a determined attacker has gained access to the values of $h$, $s$. Thus, disclosure of the random nonce $k$ leads to the recovery of the secret key $\alpha$ [27].

**Property of ECDSA cryptosystem:** For a reasonable number of ECDSA signatures, leakage of a small fraction of the corresponding random nonce $k$ can lead to the recovery of the underlying secret key by application of a lattice based attack [27] [2].

## 5.1 Related Work

Based on the abovementioned property of ECDSA cryptosystem, several works [27] [2] [23] [36] have been published that carry out attacks to recover the underlying secret key from the obtained nonce leaks.

### 5.1.1 Recovery of 160 bit secret key

Nguyen and Shparlinski [27] present a provable polynomial time attack against ECDSA when the nonces are partially known. For a certain (polynomially bounded) number of signatures, if few LSBs of the corresponding nonces are known, then the signer's secret key can be recovered in polynomial time. According to their estimations, when 7 consecutive zero bits of the nonce were detected, $23 \times 2^7 = 2944$

signatures should be examined in order to recover a 160 bit secret key [2]. To recover the secret key, they would use a lattice of dimension 23. In order to carry out their analysis, they required 23 signatures for which 7 consecutive zero bits of the random nonce were leaked through the attack. Obtaining 23 such signatures in their experimental setup requires examining 2944 signatures. Liu and Nguyen [37] developed an algorithm which allowed the recovery of 160-bit secret keys with only 2 bit nonce leakage.

### 5.1.2 Recovery of 256 bit secret key

Benger et al. [2] obtain a significant portion of the nonce with a high probability across several signatures. But for analysis, they could use only a few consecutive bits of the nonce for each signature. The authors apply an improved lattice analysis which requires smaller number of signatures for recovering the secret key. They show that a 256 bit secret key can be recovered with relatively high probability by spying on as little as 200 signatures. In their modified lattice attack, they not only utilize the least significant zero bits obtained from the FLUSH+RELOAD attack, but also notice that if their attack identifies that $LSB_b(k) = 0$, then the $(b+1)^{th}$ bit of $k$ is 1. Also, while considering $LSB_b(k) = 0$ across the observed signatures, they allow the value of $b$ to be a variable. Thus, they utilize all the captured information about the least significant zero bits, in contrast to the approach in [27] where a fixed value of $b$ was used for the analysis.

### 5.1.3 Recovery of 384 bit secret key

Mulder et al. [36] describe the recovery of secret key based on a nonce leak in 384-bit ECDSA. In their work, the authors extracted the low-order bits of each nonce using a template-based power analysis attack against the modular inversion of the nonce. The authors recovered the secret key using a 5-bit nonce leak from 4000 signatures.

In general, in order to recover the secret key from nonce leaks, the obtained leakage is mapped to a Hidden Number Problem (HNP) and is solved using lattice methods [36]. Regardless of the number of signatures, current lattice methods require minimum number of bits of the nonce to leak [36]. Hence, an approach which employs lattice methods may not always be possible due to insufficient amount of nonce leakage for each signature.

Given enough signatures, the Bleichenbacher's approach [38] to the Hidden Number Problem can be applied to recover the secret key for small, even fractional, bit leaks [36].

## 5.2 Analysis of side-channel information

In order to be able to carry out an attack based on lattice methods or the Bleichenbacher's attack, we aim at recovering the least significant bits of the random nonce $k$ for each signature generation instance.

### 5.2.1 Analysis for Method 1

According to Algorithm 2.3, equation 5.2 and equation 5.3 are used to decompose the scalar multiplier $k$ into $k_1$ and $k_2$.

$$k_1 = k - (c_1 a_1 + c_2 a_2) \tag{5.2}$$

$$k_2 = c_1 b_1 - c_2 b_2 \tag{5.3}$$

We undertake a number-theoretic approach and analyze the parity of the involved integers. In an endeavor to obtain the least significant bit of $k$, we explored various possibilities that may reveal the parity of $k$. We focused on equations 5.2 and 5.3 to check if there is a possibility to derive the parity of $k$ in order to obtain the last bit of $k$.

From the attack, we identify instances of ECDSA signature generation process when $k_1$ and $k_2$ have least significant zero bits. Hence, the identified instances have $k_1$ and $k_2$ as even numbers. It follows from basic number theory that when any of the following conditions is satisfied, then $k$ is an even number:

- $a_2$ is even, $b_1$ is odd, $b_2$ is even.
- $a_1$ is even, $b_1$ is even, $b_2$ is odd.
- $a_1, a_2, b_1, b_2$ are even.

The values used in our signature generation process are mentioned in Appendix C. It can be seen from Table C.1 that $a_1$, $a_2$, $b_1$ are odd numbers while $b_2$ is an even number. These values correspond to the value of $l = 70$ in the Algorithm 2.3.

We explore other possible values of $a_1$, $a_2$, $b_1$ and $b_2$ that might reveal the parity of $k$ from the obtained side-channel information. It is important to note that the values $a_1$, $a_2$, $b_1$ and $b_2$ are derived from the Extended Euclidean Algorithm applied on $(n, \lambda)$. We observe the output of the Extended Euclidean Algorithm with $(n, \lambda)$ as inputs. In the output sequences, no two consecutive numbers $r_l$ and $r_{l+1}$ are both even.

Therefore, for any chosen $a_1$, $a_2$, $b_1$ and $b_2$ (in accordance with Algorithm 2.3), the obtained side-channel information does not reveal *at least* the parity of $k$. Hence, we cannot perform a conducive analysis from the obtained side-channel information, when method 1 is used for GLV decomposition.

### 5.2.2 Analysis for Method 2

An implementation of Extended Euclidean Algorithm is indispensable for analyzing Algorithm 2.4. We employed an available implementation [39] of the Extended Euclidean Algorithm for our analysis. The input values $(n, \lambda)$ to the Extended Euclidean Algorithm are presented in Appendix C. For a given scalar multiplier $k$, a desired ratio of balance between the size of the decomposed scalars can be achieved through Algorithm 2.4. Brumley and Nyberg [18] mention that there is no particular advantage in choosing an unbalanced representation. Equation 5.4 gives the relation between the scalar multiplier $k$ and the decomposed scalars $k_1$, $k_2$. Since we can identify the signature generation instances with least significant zero bits for $k_1$ and $k_2$, we need the least significant bits of $k_2\lambda \pmod{n}$ to obtain the least significant bit/s of the nonce $k$.

$$\boxed{k = k_1 + k_2\lambda \pmod{n}} \tag{5.4}$$

We discuss the possibility of obtaining information about the least significant bits of the nonce $k$ for balanced and unbalanced representations.

- **Balanced Representation:** Let us assume that a balanced decomposition of the multiplier is carried out by this method. To obtain a balanced decomposition from the Algorithm 2.4, the value of $m \approx 70$. From the equation 5.4, it is not possible to derive the least significant bits of $k$ from the least significant bits of $k_1$ and $k_2$ though the values of $\lambda$ and $n$ are known to us. From the situation, we can infer that there is no possibility of converting the obtained side-channel leakage into leakage of the nonce $k$.

- **Unbalanced Representation:** Since this method allows an unbalanced representation of the scalar multiplier, we explore other interesting values of $m$ for the chosen $(n, \lambda)$. One such interesting choice of $m$ is 56, since the decimal value of $r_{57}$ ends with zeroes.

  We have $r_{57} \pmod{100} = 0$ and $q_{57} = 1$. In Algorithm 2.4, if the value $m$ is chosen as 56, then $0 \leq |k_2| < |t_{56}| + |t_{57}|$. It follows from Lemma 2.1 that there exist integers $a$ and $b$ such that equation 5.5 holds.

$$\boxed{k_2\lambda \pmod{n} = ar_{56} + br_{57}} \tag{5.5}$$

  Therefore, few least significant bits of $k_2\lambda \pmod{n}$ can be retrieved if we have knowledge about the least significant bits of $ar_{56}$. But we could not gain information about the value of $ar_{56}$ based on the decimal value of $r_{56}$.

$$r_{56} = 3132293713335127508664552971547897706870638407$$

Hence, the use of method 2 for GLV decomposition does not allow the attacker to gain sensitive information about the nonce $k$ from the leaked side-channel information about $k_1$ and $k_2$.

### 5.2.3 Analysis for Method 3

Gallant, Lambert and Vanstone [7] reflect on the security considerations of GLV curves and mention the following statement about attacks on cryptographic schemes.

"... *in any instance of a cryptographic scheme, there is always the chance that an attack will be forthcoming that applies to the special instance and significantly weakens the security.*"

In the following discussion, we consider the example presented in Section 2.4.3. We assume that the obtained side-channel information consists of 3-bit nonce leaks. To find a balanced length-two representation of $k$, we write

$$\boxed{k = 2^{195}k_2' + k_1', \text{ where } k_1' < 2^{195}} \tag{5.6}$$

Since $k_1' < 2^{195}$, the maximum number of bits that the binary representation of $k_1'$ can contain is 195. From equation 5.6, we can derive the least significant bits of $k$, given the least significant bits of $k_1'$.

$$LSB_b(k) = LSB_b(k_1'), \text{ for } b \leq 195 \tag{5.7}$$

We explored the possibility to retrieve $LSB_b(k)$ from the available side-channel information about $LSB_b(k_1)$ and $LSB_b(k_2)$.

$$\begin{aligned} kP &= (2^{195}k_2' + k_1')P && (5.8a) \\ &= ((3\lambda + 2)k_2' + k_1')P && (5.8b) \\ &= \underbrace{(2k_2' + k_1')}_{k_1} P + \underbrace{(3k_2')}_{k_2} \lambda P && (5.8c) \end{aligned}$$

From equation 5.8c,

$$k_1' = 2k_2' - k_1$$
$$LSB_b(k_1') = LSB_b(2k_2') - LSB_b(k_1)$$

From equation 5.7,

$$LSB_b(k) = LSB_b(2k_2') - LSB_b(k_1) \tag{5.9}$$

From the obtained side-channel information, we have knowledge of signature generation instances for which $LSB_3(k_1) = 0$.
From equation 5.9,

$$LSB_3(k) = LSB_3(2k'_2) \tag{5.10}$$

From equation 5.8c,

$$3k'_2 = k_2$$

$$LSB_b(3k'_2) = LSB_b(k_2)$$

From the available side-channel information, we know that the decomposed scalar $k_2$ has 3 least significant zero bits.

$$LSB_3(k_2) = 0 \implies LSB_3(3k'_2) = 0 \implies LSB_3(2k'_2) = 0$$

From equation 5.10,

$$\boxed{LSB_3(k) = 0}$$

Through the FLUSH+RELOAD attack, we identify ECDSA signatures corresponding to signature generation instances satisfying the property, $LSB_b(k_1) = LSB_b(k_2) = 0$. Thus, when this method is employed for GLV decomposition, there exists a possibility to derive a significant amount of nonce leak from the available side-channel information (i.e., from the least significant zero bits of $k_1$ and $k_2$).

In Section 5.1, we discuss several published attacks that recover the underlying secret key from the obtained nonce leaks. In general, the nonce leaks are mapped to a HNP and solved by lattice methods. When a side-channel attack can only collect reliable information about very few bits, Bleichenbacher's approach can be followed as an alternative to lattice techniques [36].

After spying on a reasonable number of signature generation processes, the spy process gains enough side-channel information about the secret key to perform a proper analysis.

## 5.3 Risk of information leakage

From the discussion in section 5.2, it is evident that the obtained side-channel information does not always lead to nonce leakage. In order to quantify the risk, we assume that the practitioner uses method 3 for GLV decomposition. We also assume that the obtained side-channel information can be converted into nonce leakage.

Based on the discussion in section 3.7.1, we present a mathematical estimate of the accuracy with which the FLUSH+RELOAD attack captures side-channel information for a randomly chosen nonce. Under our assumptions, this estimate quantifies the risk of nonce leakage from an ECDSA signature generation process.

- The probability that a random nonce is susceptible to our attack, $P(nonce_{susp}) = 0.216$.
- The probability that the Flush+Reload attack accurately captures the exploitable information, $P(success) = 0.80033$.

***Note:*** Section 3.7.1 defines *'success'* as being able to capture exactly $b$ trailing zeroes whenever $LSB_b(k_1k_2) = 0$. In practice, when $LSB_b(k_1k_2) = 0$, capturing any number of trailing zeroes less than $b$ also provides useful side-channel information.

Thus, the probability of capturing nonce leakage from a random ECDSA signature generation by the Flush+Reload attack is 0.173.

# Chapter 6

# Discussion

***R.Q.1: How would the* FLUSH+RELOAD *side channel attack be applicable to GLV method for point multiplication?***

We considered the implementation of GLV method that decomposes a scalar multiplier into two scalars. We performed the FLUSH+RELOAD side channel attack on instances of ECDSA signature generation process and successfully retrieved sensitive information about the decomposed scalars. In certain cases, the information leakage of the decomposed scalars $k_1$ and $k_2$ can lead to leakage of the scalar multiplier $k$. If GLV decomposition is carried out according to method 1 (Section 2.4.1) or method 2 (Section 2.4.2), then we cannot obtain information about the nonce $k$ from the information leakage of $k_1$ and $k_2$.

*If the GLV decomposition is carried out according to method 3 (Section 2.4.3), then there exist possibilities of obtaining information about least significant bits of the random nonce $k$ provided we have information about the least significant bits of $k_1$ and $k_2$.*

This information about the least significant bits of the random nonce $k$ in the identified ECDSA signature generation instances can lead to the recovery of the underlying secret key. The nonce leakage is mapped to a Hidden Number Problem which is then solved by the application of lattice methods or by Bleichenbacher's solution. Thus, there exist possibilities to apply the FLUSH+RELOAD attack to the GLV method for point multiplication under the following conditions:

- GLV decomposition is performed by method 3 (Section 2.4.3).
- The scalar encoding scheme of the decomposed scalars retains the least significant zero bits (Section 2.5).
- Point multiplication is performed by the interleaved scalar multiplication (Algorithm 2.2).

***R.Q.2: How does the use of GLV method in ECDSA signature generation affect the amount of information obtained from the* FLUSH+RELOAD *attack?***

*The GLV method reduces the amount of crucial information obtained by applying the* FLUSH+RELOAD *attack; the leaked information about $k_1$ and $k_2$ cannot always be converted into information about the nonce $k$. The efficiency of solving the hidden number problem depends upon the amount of information known about the nonce $k$.*

In the following discussion, we assume that our choice of elliptic curve parameters allows us to employ any of the methods for GLV decomposition (Section 2.4). It is reasonable to assume that all the methods produce identical decompositions (i.e., identical $\{k_1, k_2\}$) for an identical input $k$. Our attack retrieves side-channel information from the interleaved multiplication of the decomposed scalars. Thus, the amount of side-channel information obtained from the application of FLUSH+RELOAD attack does not vary by adopting an alternate method for GLV decomposition. But, only a particular choice of GLV decomposition unfolds a possibility of obtaining nonce leakage from the captured side-channel information.

***R.Q.3: How does the encoding of the scalar used in the GLV method defend the method against the* FLUSH+RELOAD *attack?***

Due to limitations in time, we could only find a partial answer to this research question. The decomposed scalars $k_1$, $k_2$ were encoded in the modified $w$NAF form. The trailing zeroes of a scalar are retained in its modified $w$NAF representation. This property of the modified $w$NAF encoding allowed us to identify the least significant zero bits of $k_1$ and $k_2$.

*Therefore, the encoding of the scalar critically affects the applicability of the* FLUSH+RELOAD *attack. A different encoding scheme for the decomposed scalars; an encoding scheme that does not retain trailing zeroes of the input number, can significantly strengthen the defense of the GLV method against the* FLUSH+RELOAD *attack. Such a scalar encoding scheme would render our attack ineffective, even against signature generation instances that employ method 3 (Section 2.4.3) for GLV decomposition.*

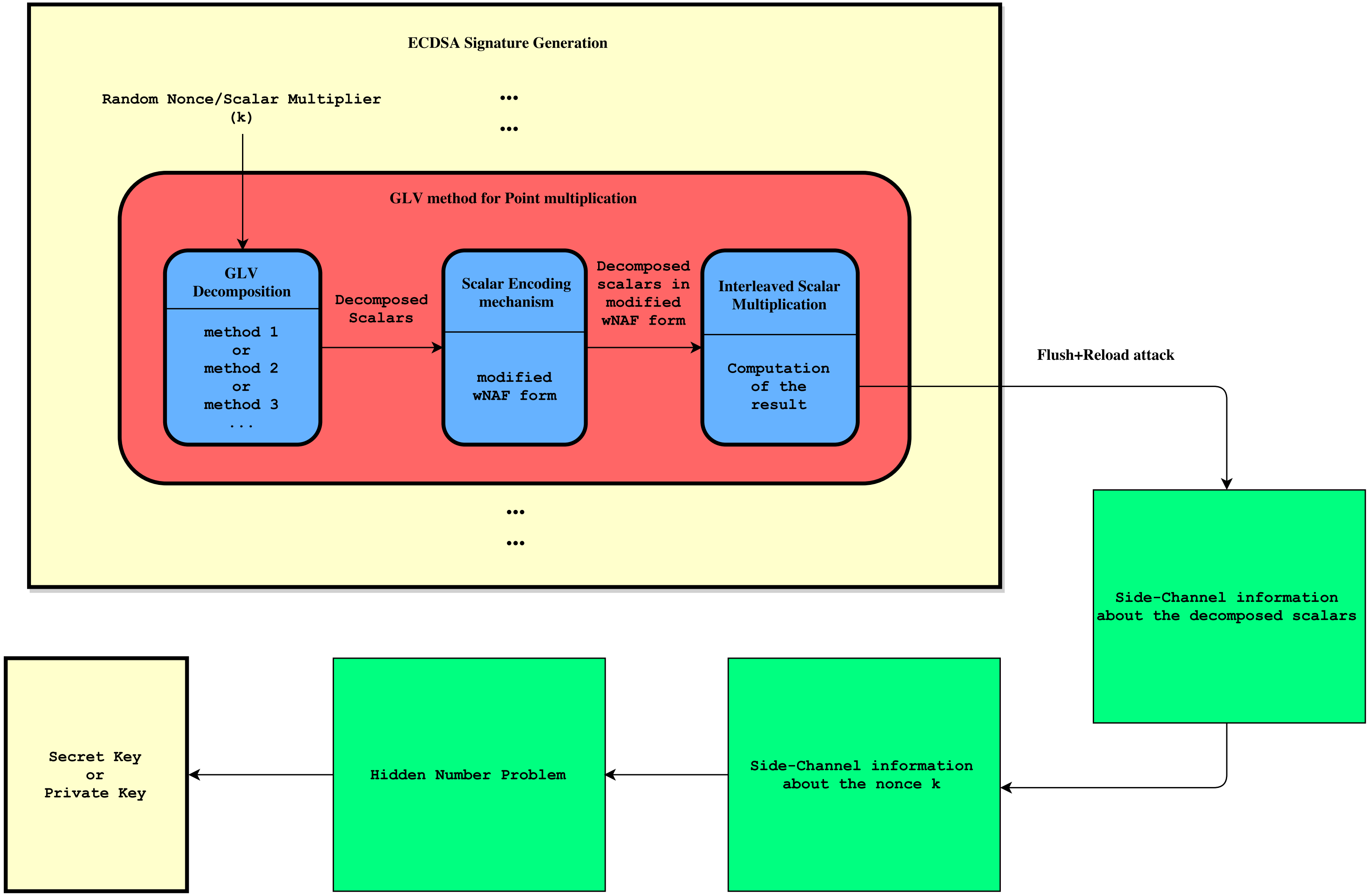



Figure 6.1: Overall view of the study

Our work does not discuss the mapping of nonce leakage to the Hidden Number Problem.

# Chapter 7

# Conclusions and Future Work

Through our work, we demonstrated the applicability of the FLUSH+RELOAD attack on ECDSA signature generation process when GLV method is used for point multiplication. We carried out the attack against the interleaved scalar multiplication step in the GLV method.

We captured side-channel information corresponding to the modified $w$NAF form of the decomposed scalars. Since the modified $w$NAF form of a number preserves its trailing zero bits, the captured information is equivalent to the side-channel information about the decomposed scalars. There exist possibilities of reconstructing the secret key from the obtained side-channel information only when method 3 (Section 2.4.3) is used for GLV decomposition. This involves obtaining nonce leakage from the captured side-channel information and mapping the nonce leakage to a Hidden Number Problem, which is then solved by lattice techniques or by Bleichenbacher's solution. The side-channel information is obtained even when alternate methods (method 1, method 2) are used for GLV decomposition. But in these cases, this information cannot be converted to nonce leakage.

*Based on our work, we conclude that the GLV method does protect against* FLUSH+RELOAD *attack. Based on the results and the presented analyses, it is recommended to avoid method 3 (Section 2.4.3) in ECDSA implementations. When method 3 is used for GLV decomposition, the captured side-channel information can possibly lead to the recovery of the secret key. Since the recovery of side-channel information exploits a property of the modified $w$NAF encoding scheme, we recommend practitioners to adopt better encoding schemes that are robust to such attacks.*

Through our demonstration and critical analyses, we infer that various specific aspects of the GLV method can cause the applicability of the FLUSH+RELOAD side channel attack to succeed. A practitioner should be aware of such aspects of the GLV method that increase its vulnerability towards the FLUSH+RELOAD attack and similar side-channel attacks.

## 7.1 Future Work

An interesting research topic is to check the applicability of the Flush+Reload attack on the decomposition phase of the GLV method for point multiplication. It would also be interesting to perform a thorough review of scalar encoding mechanisms and their effect on the defensive capabilities of GLV method against the Flush+Reload attack.

# Bibliography


[1] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003. ISBN: 038795273X.

[2] Naomi Benger et al. "“Ooh Aah... Just a Little Bit” : A Small Amount of Side Channel Can Go a Long Way". English. In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 75–92. ISBN: 978-3-662-44708-6. DOI: `10.1007/978-3-662-44709-3_5`. URL: `http://dx.doi.org/10.1007/978-3-662-44709-3_5`.

[3] JoppeW. Bos et al. "Elliptic Curve Cryptography in Practice". English. In: *Financial Cryptography and Data Security*. Ed. by Nicolas Christin and Reihaneh Safavi-Naini. Vol. 8437. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 157–175. ISBN: 978-3-662-45471-8. DOI: `10.1007/978-3-662-45472-5_11`. URL: `http://dx.doi.org/10.1007/978-3-662-45472-5_11`.

[4] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: `http://dl.acm.org/citation.cfm?id=2671225.2671271`.

[5] John Kelsey et al. "Side channel cryptanalysis of product ciphers". English. In: *Computer Security — ESORICS 98*. Ed. by Jean-Jacques Quisquater et al. Vol. 1485. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 97–110. ISBN: 978-3-540-65004-1. DOI: `10.1007/BFb0055858`. URL: `http://dx.doi.org/10.1007/BFb0055858`.

[6] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". English. In: *International Journal of Information Security* 1.1 (2001), pp. 36–63. ISSN: 1615-5262. DOI: `10.1007/s102070100002`. URL: `http://dx.doi.org/10.1007/s102070100002`.

[7] RobertP. Gallant, RobertJ. Lambert, and ScottA. Vanstone. "Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms". English. In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 190–200. ISBN: 978-3-540-42456-7. DOI: 10.1007/3-540-44647-8_11. URL: http://dx.doi.org/10.1007/3-540-44647-8_11.

[8] Bodo Möller. "Securing Elliptic Curve Point Multiplication against Side-Channel Attacks". English. In: *Information Security*. Ed. by GeorgeI. Davida and Yair Frankel. Vol. 2200. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 324–334. ISBN: 978-3-540-42662-2. DOI: 10.1007/3-540-45439-X_22. URL: http://dx.doi.org/10.1007/3-540-45439-X_22.

[9] BillyBob Brumley. "Faster Software for Fast Endomorphisms". English. In: *Constructive Side-Channel Analysis and Secure Design*. Ed. by Stefan Mangard and Axel Y. Poschmann. Vol. 9064. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 127–140. ISBN: 978-3-319-21475-7. DOI: 10.1007/978-3-319-21476-4_9. URL: http://dx.doi.org/10.1007/978-3-319-21476-4_9.

[10] Onur Acıiçmez, BillyBob Brumley, and Philipp Grabher. "New Results on Instruction Cache Attacks". English. In: *Cryptographic Hardware and Embedded Systems, CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 110–124. ISBN: 978-3-642-15030-2. DOI: 10.1007/978-3-642-15031-9_8. URL: http://dx.doi.org/10.1007/978-3-642-15031-9_8.

[11] *The GNU Privacy Guard*. https://www.gnupg.org/. Accessed: 2015-06-05.

[12] *OpenSSL: The Open Source toolkit for SSL/TLS*. https://www.openssl.org/. Accessed: 2015-06-05.

[13] Dan Page. "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel." In: *IACR Cryptology ePrint Archive* 2002 (2002), p. 169.

[14] Eran Tromer, DagArne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". English. In: *Journal of Cryptology* 23.1 (2010), pp. 37–71. ISSN: 0933-2790. DOI: 10.1007/s00145-009-9049-y. URL: http://dx.doi.org/10.1007/s00145-009-9049-y.

[15] Henri Cohen. *A Course in Computational Algebraic Number Theory*. New York, NY, USA: Springer-Verlag New York, Inc., 1993. ISBN: 0-387-55640-0.

[16] Young-Ho Park et al. "An Alternate Decomposition of an Integer for Faster Point Multiplication on Certain Elliptic Curves". English. In: *Public Key Cryptography*. Ed. by David Naccache and Pascal Paillier. Vol. 2274. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 323–334. ISBN: 978-3-540-43168-8. DOI: 10.1007/3-540-45664-3_23. URL: http://dx.doi.org/10.1007/3-540-45664-3_23.

[17] Francesco Sica, Mathieu Ciet, and Jean-Jacques Quisquater. "Analysis of the Gallant-Lambert-Vanstone Method Based on Efficient Endomorphisms: Elliptic and Hyperelliptic Curves". English. In: *Selected Areas in Cryptography*. Ed. by Kaisa Nyberg and Howard Heys. Vol. 2595. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 21–36. ISBN: 978-3-540-00622-0. DOI: 10.1007/3-540-36492-7_3. URL: http://dx.doi.org/10.1007/3-540-36492-7_3.

[18] BillyBob Brumley and Kaisa Nyberg. "On Modular Decomposition of Integers". English. In: *Progress in Cryptology – AFRICACRYPT 2009*. Ed. by Bart Preneel. Vol. 5580. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 386–402. ISBN: 978-3-642-02383-5. DOI: 10.1007/978-3-642-02384-2_24. URL: http://dx.doi.org/10.1007/978-3-642-02384-2_24.

[19] L. Babai. "On Lovász' lattice reduction and the nearest lattice point problem". English. In: *Combinatorica* 6.1 (1986), pp. 1–13. ISSN: 0209-9683. DOI: 10.1007/BF02579403. URL: http://dx.doi.org/10.1007/BF02579403.

[20] J. A. Solinas E. Brown B. T. Myers. *Elliptic curves with compact parameters*. English. Combinatorics and Optimization Research Report CORR 2001-68. University of Waterloo, 2001. URL: cacr.uwaterloo.ca/techreports/2001/corr2001-68.ps.

[21] Henri Cohen et al. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition*. 2nd. Chapman & Hall/CRC, 2012. ISBN: 1439840008, 9781439840009.

[22] BillyBob Brumley and Nicola Tuveri. "Remote Timing Attacks Are Still Practical". English. In: *Computer Security – ESORICS 2011*. Ed. by Vijay Atluri and Claudia Diaz. Vol. 6879. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 355–371. ISBN: 978-3-642-23821-5. DOI: 10.1007/978-3-642-23822-2_20. URL: http://dx.doi.org/10.1007/978-3-642-23822-2_20.

[23] Joop van de Pol, NigelP. Smart, and Yuval Yarom. "Just a Little Bit More". English. In: *Topics in Cryptology — CT-RSA 2015*. Ed. by Kaisa Nyberg. Vol. 9048. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 3–21. ISBN: 978-3-319-16714-5. DOI: 10.1007/978-3-319-16715-2_1. URL: http://dx.doi.org/10.1007/978-3-319-16715-2_1.

[24] Justin Zobel. *Writing for Computer Science*. SpringerVerlag, 2014. ISBN: 978-1-4471-6639-9.

[25] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual, Order Number: 248966-028*. 2013. URL: `https://spaces.usu.edu/` (visited on 08/06/2015).

[26] *Patch: Faster GLV elliptic curves in OpenSSL*. `http://rt.openssl.org/Ticket/Display.html?id=3667&user=guest&pass=guest`. Accessed: 2015-06-05.

[27] PhongQ. Nguyen and IgorE. Shparlinski. “The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces”. English. In: *Designs, Codes and Cryptography* 30.2 (2003), pp. 201–217. ISSN: 0925-1022. DOI: `10.1023/A:1025436905711`. URL: `http://dx.doi.org/10.1023/A:1025436905711`.

[28] *Implementation of the FLUSH+RELOAD side channel attack*. `https://github.com/DanGe42/flush-reload`. Accessed: 2015-06-05.

[29] Yuval Yarom and Naomi Benger. “Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack”. In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 140. URL: `http://eprint.iacr.org/2014/140`.

[30] Augustus K. Uht, Vijay Sindagi, and Kelley Hall. “Disjoint Eager Execution: An Optimal Form of Speculative Execution”. In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. MICRO 28. Ann Arbor, Michigan, USA: IEEE Computer Society Press, 1995, pp. 313–325. ISBN: 0-8186-7349-4. URL: `http://dl.acm.org/citation.cfm?id=225160.225208`.

[31] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual, Order Number: 248966-026*. 2012. URL: `https://` (visited on 08/06/2015).

[32] Ronald E Walpole et al. *Probability and statistics for engineers and scientists*. Pearson, 2012.

[33] Everitt. *The Cambridge dictionary of statistics*. 2002.

[34] GottfriedE. Noether. “The Analysis of Binomial Experiments”. English. In: *Introduction to Statistics*. Springer Texts in Statistics. Springer New York, 1991, pp. 349–370. ISBN: 978-1-4612-6955-7. DOI: `10.1007/978-1-4612-0943-0_20`. URL: `http://dx.doi.org/10.1007/978-1-4612-0943-0_20`.

[35] Teodoro Cipresso and Mark Stamp. “Software Reverse Engineering”. English. In: *Handbook of Information and Communication Security*. Ed. by Peter Stavroulakis and Mark Stamp. Springer Berlin Heidelberg, 2010, pp. 659–696. ISBN: 978-3-642-04116-7. DOI: `10.1007/978-3-642-04117-4_31`. URL: `http://dx.doi.org/10.1007/978-3-642-04117-4_31`.

[36] Elke De Mulder et al. "Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version". English. In: *Journal of Cryptographic Engineering* 4.1 (2014), pp. 33–45. ISSN: 2190-8508. DOI: 10.1007/s13389-014-0072-z. URL: http://dx.doi.org/10.1007/s13389-014-0072-z.

[37] Mingjie Liu and PhongQ. Nguyen. "Solving BDD by Enumeration: An Update". English. In: *Topics in Cryptology – CT-RSA 2013*. Ed. by Ed Dawson. Vol. 7779. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 293–309. ISBN: 978-3-642-36094-7. DOI: 10.1007/978-3-642-36095-4_19. URL: http://dx.doi.org/10.1007/978-3-642-36095-4_19.

[38] Daniel. Bleichenbacher. *On the generation of one-time keys in DL signature schemes*. Presentation at IEEE P1363 working group meeting. 2000.

[39] Intel Corporation. *An Implementation of Extended Euclidean Algorithm.* URL: http://www.numbertheory.org/php/euclid.html#euclid (visited on 08/06/2015).

# Appendices

# Appendix A

# List of Abbreviations

| | |
|---|---|
| **ECC** | Elliptic Curve Cryptography, p. i |
| **ECDH** | Elliptic Curve Diffie-Hellman, p. 3 |
| **ECDSA** | Elliptic Curve Digital Signature Algorithm, p. i |
| **GLV** | Gallant-Lambert-Vanstone, p. i |
| **GnuPG** | GNU Privacy Guard, p. 4 |
| **HNP** | Hidden Number Problem, p. 51 |
| **L1** | Level 1, p. 3 |
| **L2** | Level 2, p. 4 |
| **L3** | Level 3, p. 18 |
| **LLC** | Last Level Cache, p. 4 |
| **RAM** | Random Access Memory, p. 3 |
| **RSA** | Rivest-Shamir-Adleman, p. 4 |
| **SSH** | Secure Shell, p. 1 |
| **SSL** | Secure Sockets Layer, p. 21 |
| **TLS** | Transport Layer Security, p. 1 |
| $w$**NAF** | width-$w$ Non-Adjacent Form, p. 15 |

# Appendix B

# Nomenclature

| | |
|---|---|
| $\boldsymbol{k}$ | Random nonce or Scalar multiplier or Ephemeral key, p. 10 |
| $\boldsymbol{k_1}$ | Decomposed scalar, p. 10 |
| $\boldsymbol{k_2}$ | Decomposed scalar, p. 10 |
| $\boldsymbol{\lambda}$ | Integer representation of the endomorphism, p. 9 |
| $\boldsymbol{LSB_b(k_1k_2) = 0}$ | Denotes that $LSB_b(k_1) = 0, LSB_b(k_2) = 0$ and at most one of $\{LSB_{b+1}(k_1), LSB_{b+1}(k_2)\}$ is equal to 0, p. 31 |
| $\boldsymbol{LSB_b(n)}$ | Denotes the $b$ least significant bits of the binary representation of $n$, p. 31 |
| $\boldsymbol{LSB_b(n) = 0}$ | Denotes that the binary representation of number $n$ has $b$ least significant zero bits, p. 31 |
| $\boldsymbol{\alpha}$ | Private key or Secret key, p. 21 |
| $\boldsymbol{secp256k1}$ | The ECDSA curve used in Bitcoin system, p. i |

# Appendix C

# Parameters for secp256k1 curve

| Parameter | Length | Hexadecimal Value | Explanation |
|---|---|---|---|
| **Value of p ($p$)** Integer | 256 bits | `FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE FFFF FC2F` | Elliptic curve is defined over $\mathbb{F}_p$. |
| **Generator ($G$)** Point on the curve | 256 bits | `79BE 667E F9DC BBAC 55A0 6295 CE87 0B07 029B FCDB 2DCE 28D9 59F2 815B 16F8 1798` | `x-coordinate` of the base point $G$. |
| **Generator ($G$)** Point on the curve | 256 bits | `483A DA77 26A3 C465 5DA4 FBFC 0E11 08A8 FD17 B448 A685 5419 9C47 D08F FB10 D4B8` | `y-coordinate` of the base point $G$. |
| **Order ($n$)** Integer | 256 bits | `FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4141` | Order of the base point $G$. |
| **Beta ($\beta$)** An element of order 3 in Fp | 256 bits | `8516 95D4 9A83 F8EF 919B B861 53CB CB16 630F B68A ED0A 766A 3EC6 93D6 8E6A FA40` | It can be verified that $\beta^3 \bmod p = 1$. The following relation between $\beta$ and $\lambda$ can also be verified: $\lambda P = (\beta x, y)$, where $P$ is any point on the curve, denoted by $(x, y)$. |

| | | | |
|---|---|---|---|
| **Lambda ($\boldsymbol{\lambda}$)** Integer Representation of the endomorphism | 256 bits | `AC9C 52B3 3FA3 CF1F 5AD9 E3FD 77ED 9BA4 A880 B9FC 8EC7 39C2 E0CF C810 B512 83CE` | It can be verified that $\lambda$ is a root of the equation $\lambda^2 + \lambda + 1 = 0 \pmod{n}$, the other two roots of the equation being 1, and $n - \lambda - 1$. |
| **Constant ($\boldsymbol{a_1}$)** Description | 128 bits | `E443 7ED6 010E 8828 6F54 7FA9 0ABF E4C3` | Refer Algorithm 2.3 |
| **Constant ($\boldsymbol{a_2 = b_1}$)** Description | 128 bits | `3086 D221 A7D4 6BCD E86C 90E4 9284 EB15` | Refer Algorithm 2.3 |
| **Constant ($\boldsymbol{b_2}$)** Description | 128 bits | `0001 14CA 50F7 A8E2 F3F6 57C1 108D 9D44 CFD8` | Refer Algorithm 2.3 |

Table C.1: Parameters of secp256k1 curve

# Appendix D

# Spy code

Listing D: Code snippet from the spy program

```
int probe_time(char *adrs) {
    volatile unsigned long time;

    asm __volatile__(
        "    mfence             \n"
        "    lfence             \n"
        "    rdtsc              \n"
        "    lfence             \n"
        "    movl %%eax, %%esi  \n"
        "    movl (%1), %%eax   \n"
        "    lfence             \n"
        "    rdtsc              \n"
        "    subl %%esi, %%eax  \n"
        "    clflush 0(%1)      \n"
        :    "=a" (time)
        :    "c" (adrs)
        :    "%esi", "%edx"
    );
    return time;
}
```

# Appendix E

# Plotting script

Listing E: `gnuplot` script to generate the output plots

```
# Threshold value = 120 clock cycles.

#---------------------------------------------------------

reset
set term canvas dashed rounded size 5000,800 enhanced fsize
 ↪ 20 lw 1.6 fontscale 1 standalone mousing jsdir "./js"

# define axis, remove border on top and right and set color
 ↪ to gray
set style line 11 lc rgb '#808080' lt 1
set border 3 back ls 11

# define grid
set style line 12 lc rgb '#808080' lt 0 lw 1
set grid back ls 12

#-----------------------------------------------------------

set title "FLUSH+RELOAD Attack"
set xlabel "Slot number"
set ylabel "Access Time (number of clock cycles)"
set key top right
set key font ",13"
set xtics 2
set grid ytics lt 0 lw 1 lc rgb "#bbbbbb"
set grid xtics lt 0 lw 1 lc rgb "#bbbbbb"

set yrange [0:250]
set pointsize 2
```

```
# Round function
# round(z) = z - floor(z) < 0.5 ? floor(z) : ceil(z)

# If the input to the function is any number less than 100,
↪  then the output is 50, else the output is 150
f(z) = z < 100 ? 50 : 150

#------------------------------------------------------------

Dbl = 50
Add = 40

threshold = 120
dblcount = 0
addcount = 0

#-------------------------------------------------------------

set output 'Initial Plot.html'
# Plot the contents of the file
plot 'out.txt' using ($2 == 0 ? $1 : 1/0): 3 title "Double"
↪  with points pointtype 7 lc rgb '#003380', \
'out.txt' using ($2 == 1 ? $1 : 1/0): 3 title "Add" with
↪  points pointtype 5 lc rgb '#FF5005'

#-------------------------------------------------------------

set output 'Threshold Plot.html'
# Plot the contents of the file
plot 'out.txt' using ($3 < threshold && $2 == 0 ? $1 : 1/0):
↪  3 title "Double" with points pointtype 7 lc rgb '#003380',
↪  \
'out.txt' using ($3 < threshold && $2 == 1 ? $1 : 1/0): 3
↪  title "Add" with points pointtype 5 lc rgb '#FF5005'

#-------------------------------------------------------------

set output 'Processed Plot.html'
# Plot the contents of the file
```

```
plot 'out.txt' using ($3 < threshold && $2 == 0 ? (dblcount =
↪  dblcount + 1, $1) : 1/0): (Dbl) title "Double" with
↪  points pointtype 7 lc rgb '#003380', \
'out.txt' using ($3 < threshold && $2 == 1 ? (addcount =
↪  addcount + 1, $1) : 1/0): (Add) title "Add" with points
↪  pointtype 5 lc rgb '#FF5005'

#--------------------------------------------------------------

print dblcount, addcount

# Press any key to continue
pause -1
```