Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage

Youngjoo Shin
Kwangwoon University
Seoul, Republic of Korea
yjshin@kw.ac.kr

Hyung Chan Kim
The Affiliated Institute of ETRI
Daejeon, Republic of Korea
kimhc@nsr.re.kr

Dokeun Kwon
The Affiliated Institute of ETRI
Daejeon, Republic of Korea
kwondk@nsr.re.kr

Ji Hoon Jeong
The Affiliated Institute of ETRI
Daejeon, Republic of Korea
binoopang@nsr.re.kr

Junbeom Hur
Korea University
Seoul, Republic of Korea
jbhur@korea.ac.kr

ABSTRACT

Data prefetching is a hardware-based optimization mechanism used in most of the modern microprocessors. It fetches data to the cache before it is needed. In this paper, we present a novel microarchitectural attack that exploits the prefetching mechanism. Our attack targets Instruction pointer (IP)-based stride prefetching in Intel processors. Stride prefetcher detects memory access patterns with a regular stride, which are likely to be found in lookup table-based cryptographic implementations. By monitoring the prefetching activities near the lookup table, attackers can extract sensitive information such as secret keys from victim applications. This kind of leakage from prefetching has never been considered in the design of constant time algorithm to prevent side-channel attacks. We show the potential of the proposed attack by applying it against the Elliptic Curve Diffie-Hellman (ECDH) algorithm built upon the latest version of OpenSSL library. To the best of our knowledge, this is the first microarchitectural side-channel attack exploiting the hardware prefetching of modern microprocessors.

CCS CONCEPTS

• **Security and privacy → Cryptanalysis and other attacks**;

KEYWORDS

Hardware prefetching, microarchitectural side-channel attacks, OpenSSL, ECDH algorithm

ACM Reference Format:
Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243734.3243736

1 INTRODUCTION

Modern microprocessors utilize a variety of microarchitectural elements to achieve hardware-level optimization. While attractive from a performance perspective, pursuing optimization has led to several microarchitectural attacks [19]. The CPU cache is a well-known hardware component that has been exploited by cache side-channel attacks [38, 48, 57]. In the attacks, the cache contention between cores results in an unintentional side-channel through which sensitive information may be leaked to attackers from other cores. Branch prediction unit (BPU) is another hardware component that modern processors rely on to enable speculative execution. The Recently discovered microarchitectural attacks, dubbed as Meltdown [37] and Spectre [35], exploit the BPU and speculative execution to break the fundamental isolation between different security domains in software systems.

In this paper, we present a new microarchitectural attack that exploits *hardware-based data prefetcher*, which is a kind of processor optimization element in most commodity CPUs. The hardware prefetcher is located in the CPU cache hierarchies. It attempts to predict the memory access patterns and fetches the anticipated lines to the data cache before they are actually accessed, which otherwise would result in a cache miss.

Contemporary Intel x86 cores are equipped with four kinds of hardware-based data prefetchers, each of which features a specific prefetching algorithm [27]. We discovered that among those prefetchers, the *Instruction Pointer (IP)-based stride prefetcher* affects cache usage during the execution of constant-time cryptographic algorithms. As its name implies, this prefetcher detects memory access patterns with regular stride on load instructions in a loop structure. Such kinds of access patterns are often found in lookup table-based cryptographic implementations. If a lookup operation accidentally produces a sequence of strided memory accesses, the prefetcher is then triggered to fetch the memory lines adjacent to the lookup table, which in turn leaves unique footprints on the cache. Our attack utilizes those footprints near the lookup table as a side-channel to infer secret keys of the cryptographic algorithm from other cores. Such kind of information leakage from prefetching has never been considered in the design of constant-time algorithms.

We examine the potential of the proposed attack by applying it against OpenSSL 1.1.0g [47], the latest version of the cryptographic library at the time of writing this paper[1]. In order to support elliptic curve cryptography [34], the library includes a scalar point multiplication algorithm, which is implemented in the form of *branchless Montgomery ladder.* It is a state-of-the-art constant-time algorithm that is known to have no secret-dependent cache traces and is thereby resistant against cache side-channel attacks [32, 46]. However, in terms of the implementation of squaring over binary field GF(2^m), which is a field arithmetic internally used in the Montgomery ladder algorithm, OpenSSL library implements it using a lookup table to improve computational efficiency. When the scalar point multiplications are executed, the prefetching activities affect the cache lines adjacent to the squaring lookup table. By carefully monitoring those activities, we are able to recover unknown bits of the scalar.

By exploiting this microarchitectural side-channel, we could successfully extract a private key of the Elliptic Curve Diffie-Hellman (ECDH) algorithm. In OpenSSL 1.1.0g with sect571r1 curve (NIST Binary-Curve B-571 [45]), it took about 19 CPU hours for an attacker from a different core to complete the private key recovery process. It is important to note that among all the attack procedures, the online phase takes only about 3 min to complete, which makes our attack practical in real world applications.

Affected processors. We investigated all Intel processors available in our experiments to determine which models are affected by our attack. As a result, we found that Core i5-3570 (Ivy Bridge), i5-4690 (Haswell), Xeon E5-2620v4, and E5-2630v4 (Broadwell) are vulnerable to the attack. It is worth noting that these microarchitectures are the most widely used ones in commercial cloud infrastructure such as AWS EC2 [3].

1.1 Our Contribution

In this paper, we present our novel discovery that hardware-based data prefetching in modern processors can be exploited as a source of information leakage. We demonstrate that the IP-based stride prefetching enables microarchitectural side-channel attacks on constant-time cryptographic algorithms that are supposed to be immune to such attacks. Specifically, the main contributions of this paper are threefold as follows:

Novel method for searching side-channel. We propose a novel method for searching a new side-channel in cryptographic applications. It allows us to find a hidden source of side-channel in applications that cannot be revealed through the program analysis-based approaches [12, 13, 55]. Our methodology differs from those approaches in how the vulnerable locations are found. Specifically, we take into consideration real cache traces on all the memory lines belonging to the executable binary. By exhaustively inspecting them, we learn which lines have secret-dependent cache traces. The discovery of information leakage from the Intel stride prefetcher resulted from the proposed method. This method is generic and can be done automatically; therefore, it can be easily adopted to find the side-channel vulnerabilities of any kind of cryptographic library. We give the detailed procedure in Section 3.

An in-depth analysis on the Intel stride prefetcher. We disclose details of the IP-based stride prefetcher in Intel core, which has never been explored in the literature. Due to the lack of sufficient information on the prefetcher, we conduct an in-depth analysis through diverse experiments. Specifically, we analyze the observed prefetching activities by comparing it with the ground truth of memory access patterns from the actual execution trace. Through this approach, we extend the knowledge on prefetching, *i.e.,* under what conditions prefetching is triggered and what memory line is fetched into the cache. The analysis results on the Intel stride prefetcher is given in Section 4. We demonstrate the prefetching behavior with an example of the scalar point multiplication over GF(2^m) in the OpenSSL library, which is given in Section 5.

Attacking ECDH in the latest version of OpenSSL. Based on the result of prefetching analysis (in Section 4) and further observation of the scalar point multiplication (in Section 5), we implement the first attack on ECDH algorithm of OpenSSL library by exploiting the hardware-based data prefetcher in the Intel microarchitecture. Our attack defeats the state-of-the-art countermeasure implemented in the latest version of OpenSSL library for cache side-channel attacks, and successfully recovers the private keys of ECDH implementation built upon the library. The detailed algorithm of our attack and experimental results are given in Section 6.

1.2 Related Work

Microarchitectural attacks. Due to its vulnerability to side-channel attacks with high resolution and low noise, a variety of side-channel attacks exploiting the CPU cache have been proposed. The proposed attacks can be classified into two major techniques, namely Flush+Reload [4, 23, 24, 26, 36, 57, 58] and Prime+Probe [7, 25, 29, 38, 42, 48], according to the granularity levels of the attack. Several variants use specialized hardware features such as Intel TSX and SGX to enhance the attack performance [11] or hide from being detected [52].

The branch prediction unit is another microarchitectural feature that can be exploited to construct side-channels. Several attacks have been proposed to break the system-level [35, 37, 41] and hardware-level [14] protection by using the BPU.

As one of the other microarchitectural features, prefetching mechanism is also getting considerable attention. Gruss *et al.,* [22] presented Prefetch Side-Channel Attacks. Their work exploits the weakness in software-based prefetching to defeat the kernel protection (*e.g.,* KASLR), which is quite different from our attack with regard to the way of exploitation and the attacker's goal. Bhattacharya *et al.,* [5, 6] examined possible information leakage from the hardware prefetcher, but the examination was conducted on a simulated CPU and no concrete attacks were presented as well. Prefetching was also utilized for protection from cache side-channel attacks [15, 17].

Searching side-channel. Identifying the side-channel vulnerabilities in a program is crucially important for designing secure systems. For this purpose, many program analysis-based approaches have been proposed. CacheAudit [12] is a formal verification tool for automatic analysis of cache side-channels. It takes a program binary and cache configuration of the target system as inputs, and provides the upper bound of information leakage from the side-channel. The tool is extended in a further work [13] to cover dynamically

[1]We noticed that OpenSSL of the version later than 1.1.0g has been released in 27 March 2018, but it is still vulnerable to our attack.

allocated memory for analysis. CacheD [55] is another tool that leverages symbolic execution and constraint solving to identify potential differences in cache usages.

In the program analysis-based approaches, real cache traces at every memory line in a program binary are not taken into consideration during investigation. Therefore, the aforementioned methods cannot find vulnerable memory locations affected by indirect cache activities from the hardware prefetcher.

Cache Template Attacks [24] utilizes real cache traces rather than relying on the program analysis for exploitation, which seems somewhat similar to the proposed searching method. However, we realize a more fine-grained method to detect the side-channel (Refer to Section 3.1 for a detailed comparison).

Attacks on constant-time cryptographic algorithms. Constant-time algorithms are widely used in various software-based cryptographic implementations to mitigate cache side-channel attacks. This countermeasure is, however, still subject to advanced and sophisticated attacks. Yarom *et al.,* [56] found that a conditional branch in OpenSSL's Montgomery ladder algorithm can be exploited to extract secret information by monitoring the cache usages. Genkin *et al.,* [20] and Kaufmann *et al.,* [33] discovered cache side-channels in the conditional branches of low-level field arithmetic algorithms in Curve25519 implementations. On the other hand, Garcia *et al.,* discovered software defects in OpenSSL's constant-time algorithms of ECDSA [18] and DSA [50], all of which are vulnerable to cache side-channel attacks.

Similar to the attacks mentioned above, our attack targets a constant-time cryptographic algorithm, but differs in that hardware prefetching is exploited. Hence, our attack can be delivered even on robust implementations of constant-time algorithms that have neither conditional branches nor software bugs.

2 PRELIMINARIES

2.1 Cache side-channel

Cache hierarchy. Cache is a special hardware in modern CPU microarchitecture. It is a small piece of storage with high speed, and aims at bridging the gap in the latency between the memory and the processor. In modern multi-core processors, the cache hierarchy has three levels of on-chip cache, namely L1,L2, and L3. L1 cache is the closest to the processor core among all caches, while its size is smaller than the others. L3 cache, which is also referred to as the last level cache (LLC), has the largest size (for instance, Intel Xeon E5-2670 v2 used in Amazon EC2 has LLC of 25MB). While L1 and L2 caches are privately used by a core, LLC is shared among all the cores of the processor.

A cache is divided into *cache sets*, and each set contains multiple *cache lines* of fixed size B (usually B=64 bytes). The cache associativity determines the number of cache lines in a cache set.

Flush+Reload technique. Our attack is constructed upon the Flush+Reload technique [57], which is used to establish the LLC-based side-channel. The Flush+Reload technique exploits memory sharing between two processes $\mathcal{A}$ and $\mathcal{B}$. This technique proceeds in three phases. During *Flush* phase, $\mathcal{A}$ flushes the desired memory lines (shared with $\mathcal{B}$) from the entire cache hierarchy using the `clflush` instruction. Owing to the inclusiveness property of the LLC in Intel processors, the `clflush` instruction will evict the cache line from all cache levels. Then, in *Wait* phase, $\mathcal{A}$ waits for $\mathcal{B}$ to execute operations. Finally, during *Reload* phase, $\mathcal{A}$ reloads previously flushed memory line and measures the access time. If $\mathcal{B}$ has accessed the line during *Wait* phase, it will be reloaded from the cache (*i.e.,* a cache hit occurs), which results in a lower reload time. If not, then the memory line resides in the memory, resulting in higher reload time due to a cache miss.

Table 1: List of hardware prefetchers in Intel processors (since Sandy Bridge)

No.	Hardware prefetcher	Detection technique	Cache Level	Bit # in MSR 0x1a4
1	Streamer	Stream	L2	0
2	Spatial prefetcher	Adjacent-line	L2	1
3	DCU prefetcher	Next-line	L1	2
4	IP-based stride prefetcher	Stride	L1	3

2.2 Hardware-based data prefetching

Cache miss results in access to lower-level caches or main memory, which is a time consuming operation that causes execution delays. Data prefetching is a technique that predicts its usage and fetches data from the main memory to the high level cache prior to its actual access. Modern processors provide data prefetching in the form of either software or hardware. While software-based prefetching is supported by the compiler, hardware-based prefetching uses a dedicated hardware in the cache units. In this study, we consider only hardware-based data prefetching techniques.

Hardware prefetching technique. Since there exist diverse applications with different memory access patterns, modern processors utilize various hardware prefetching techniques conjunctively to cover the range of applications. Next-line prefetching [53] is one of the basic approaches; if an access to line N is detected, then it prefetches the next line N+1 to the cache. Adjacent-line prefetching [30] exploits spatial locality and attempts to fetch an adjacent pair of the accessed line N. Stream prefetching [31, 49] assumes that consecutive memory accesses is a part of the streaming algorithm, and therefore fetches multiple memory lines ahead within a page boundary from the current line N. Stride prefetching [8, 16], on which this paper is mainly focused, tracks individual data load instructions in an attempt to detect whether consequently accessed addresses form a pattern $a, a + S, a + 2S, \ldots$ with memory address a and a constant stride S. Once detected, it then fetches memory lines either forward or backward based on the direction of access sequence. The stride prefetcher maintains a table indexed with load/store PC. Each entry in the table consists of the last address a, stride S, and a count c, where the count is used for measuring the confidence of the observed stride.

Hardware prefetching in Intel core. In Sandy Bridge and successive processors, each core is equipped with four types of hardware prefetchers on the L1 and L2 cache: *Streamer*, *Spatial prefetcher*, *Data Cache Unit (DCU) prefetcher*, and *Instruction pointer (IP)-based stride prefetcher* [27]. Unfortunately, details about the behavior of those

Algorithm 1 Montgomery ladder algorithm

```
Input: x-coordinate $X/Z$ for the point $P(X, Z)$ and a scalar $k > 0$
Output: The affine coordinates of the point $Q = kP$
1: procedure MONTGOMERRY_LADDER($k, X, Z$)
2:     $(k_{l-1}, \ldots, k_1, k_0) \leftarrow k$    ▷ $k_i \in \{0, 1\}$ $(0 \le i \le l - 1)$
3:     $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$   ▷ $b$ is a constant
4:     for $i \leftarrow l - 2$ to 0 do
5:         if $k_i = 1$ then
6:             MADD($X_1, Z_1, X_2, Z_2$), MDOUBLE($X_2, Z_2$)
7:         else
8:             MADD($X_2, Z_2, X_1, Z_1$), MDOUBLE($X_1, Z_1$)
9:         end if
10:    end for
11:    return $Q$ = MXY($X_1, Z_1, X_2, Z_2$)   ▷ Transform to an affine coordinate
12: end procedure
```

Algorithm 2 Branchless Montgomery ladder algorithm

```
Input: x-coordinate $X/Z$ for the point $P(X, Z)$ and a scalar $k > 0$
Output: The affine coordinates of the point $Q = kP$
1: procedure MONTGOMERRY_LADDER($k, P$)
2:     $(k_{l-1}, \ldots, k_1, k_0) \leftarrow k$
3:     $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$
4:     for $i \leftarrow l - 2$ to 0 do
5:         $\beta \leftarrow k_i$
6:         CONST_SWAP($X_1, X_2, \beta$), CONST_SWAP($Z_1, Z_2, \beta$)
7:         MADD($X_2, Z_2, X_1, Z_1$), MDOUBLE($X_1, Z_1$)
8:         CONST_SWAP($X_1, X_2, \beta$), CONST_SWAP($Z_1, Z_2, \beta$)
9:     end for
10:    return $Q$ = MXY($X_1, Z_1, X_2, Z_2$)
11: end procedure
```

prefetchers are not publicly known, except for a brief explanation of each prefetching mechanism in Intel documents [27, 28]. Recently, Intel disclosed useful information about a method to control various hardware prefetchers for some processor models [54]. According to the disclosed material, there is a Model Specific Register (MSR) with address 0x1a4 on every core, and bits 0-3 in the register can be used to either enable or disable each prefetcher. If any of those bits are set on a core, then the corresponding prefetcher on that core is turned off. In most cases, all the bits in the MSR are left to be cleared by the BIOS, which indicates that all prefetchers are enabled in the default setting. Table 1 lists those hardware prefetchers with the detection technique, location and the bit number of the MSR for control.

2.3 Scalar point multiplication in OpenSSL

OpenSSL [47] is a comprehensive library of cryptographic primitives. For elliptic curve cryptography, OpenSSL implements a scalar point multiplication algorithm over the elliptic curves of both prime fields and binary fields. In this section, we present the implementation details of the scalar point multiplication as well as the squaring operation, especially over elliptic curves defined over binary fields in the OpenSSL library.

Algorithm 3 Point doubling operation

```
Input: x-coordinate $X/Z$ for the point $P(X, Z)$
Output: x-coordinate $X/Z$ for the point $2P$
1: procedure MDOUBLE($X, Z$)
2:     $X \leftarrow X^2$
3:     $T_1 \leftarrow Z^2$
4:     $Z \leftarrow X \times T_1$
5:     $X \leftarrow X^2$
6:     $T_1 \leftarrow T_1^2$
7:     $T_1 \leftarrow b \times T_1$   ▷ $b$ is a constant specific to an elliptic curve
8:     $X \leftarrow X + T_1$
9:     return $X, Z$
10: end procedure
```

Algorithm 4 Point addition operation

```
Input: x-coordinates $X_1/Z_1$ and $X_2/Z_2$ of two points $P_1(X_1, Z_1)$ and $P_2(X_2, Z_2)$
Output: x-coordinate $X_1/Z_1$ for the point $P_1 + P_2$
1: procedure MADD($X_1, Z_1, X_2, Z_2$)
2:     $T_1 \leftarrow x$
3:     $X_1 \leftarrow X_1 \times Z_2$
4:     $Z_1 \leftarrow Z_1 \times X_2$
5:     $T_2 \leftarrow X_1 \times Z_1$
6:     $Z_1 \leftarrow Z_1 + X_1$
7:     $Z_1 \leftarrow Z_1^2$
8:     $X_1 \leftarrow Z_1 \times T_1$
9:     $X_1 \leftarrow X_1 + T_2$
10:    return $X_1, Z_1$
11: end procedure
```

2.3.1 Montgomery Ladder algorithm. Scalar point multiplication is one of the primitive operations in elliptic curve cryptography. Given a scalar $k > 0$ and an elliptic curve point P, the operation produces kP by adding P to itself k times. The simplest implementation of scalar point multiplication is not secure since it leaks timing information through which an attacker can learn the bits of the scalar k [9].

In order to protect against timing-based side channel attacks, OpenSSL library adopts the Montgomery ladder algorithm [43] (Algorithm 1) for implementing scalar point multiplication. The algorithm takes the projective coordinate X/Z of the point P and a scalar k as inputs. For each bit of k, it performs a constant number of addition (*i.e.,* MADD in Algorithm 4) and doubling (*i.e.,* MDOUBLE in Algorithm 3) operations, regardless of the value of the bit. Constant-time computation is effective in preventing side-channel attackers from learning timing information.

Branchless Montgomery ladder algorithm. The previous version of OpenSSL's implementation of the Montgomery ladder algorithm basically follows the work of Lopez and Dahab [39]. As shown in Algorithm 1, it takes different execution paths on a conditional branch depending on the value of the scalar bit. Such implementations are known to be vulnerable to access-driven cache side-channel attacks [56]. The latest versions of OpenSSL (including 1.1.0g) mitigate the attack by eliminating any branches in the

Algorithm 5 GF(2^m) squaring operation

```
Input: $X \in$ GF($2^m$)
Output: $X^2 \in$ GF($2^m$)
1: procedure GF2M_SQUARE($X$)
2:     $(X_{w-1}, X_{w-2}, ..., X_0) \leftarrow X$        ▷ $X_0$ is the right most word
3:     for $i \leftarrow w-1$ to 0 do
4:         $(x_{15}, x_{14}, ..., x_0) \leftarrow X_i$        ▷ $x_j$ is a nibble (4-bit) data
5:         $W_{15} \leftarrow$ SQR_tb[$x_{15}$] << 56
6:         $W_{14} \leftarrow$ SQR_tb[$x_{14}$] << 48
7:         $W_{13} \leftarrow$ SQR_tb[$x_{13}$] << 40
8:         $W_{12} \leftarrow$ SQR_tb[$x_{12}$] << 32
9:         $W_{11} \leftarrow$ SQR_tb[$x_{11}$] << 24
10:        $W_{10} \leftarrow$ SQR_tb[$x_{10}$] << 16
11:        $W_9 \leftarrow$ SQR_tb[$x_9$] << 8
12:        $W_8 \leftarrow$ SQR_tb[$x_8$]
13:        $W_7 \leftarrow$ SQR_tb[$x_7$] << 56
14:        $W_6 \leftarrow$ SQR_tb[$x_6$] << 48
15:        $W_5 \leftarrow$ SQR_tb[$x_5$] << 40
16:        $W_4 \leftarrow$ SQR_tb[$x_4$] << 32
17:        $W_3 \leftarrow$ SQR_tb[$x_3$] << 24
18:        $W_2 \leftarrow$ SQR_tb[$x_2$] << 16
19:        $W_1 \leftarrow$ SQR_tb[$x_1$] << 8
20:        $W_0 \leftarrow$ SQR_tb[$x_0$]
21:        $Y_{2i+1} \leftarrow W_{15} \vee W_{14} \vee ... \vee W_8$
22:        $Y_{2i} \leftarrow W_7 \vee W_6 \vee ... \vee W_0$
23:    end for
24:    $Y \leftarrow (Y_{2w-1}, Y_{2w-2}, ..., Y_1, Y_0)$
25:    return $Y$ mod $f$        ▷ $f$ is an irreducible polynomial
26: end procedure
```

implementation. Specifically, rather than using branch instructions, CONST_SWAP operation is implemented to conditionally swap two values before and after the addition and doubling operations. CONST_SWAP(X, Y, β) performs swapping between X and Y when $\beta = 1$. Algorithm 2 shows the branchless version of the Montgomery ladder algorithm.

Listing 1: Squaring lookup table in bn_gf2m.c (BN_ULONG type is 64 bits in length for x86-64)

```
static const BN_ULONG SQR_tb[16] = {
  0, 1, 4, 5, 16, 17, 20, 21,
  64, 65, 68, 69, 80, 81, 84, 85
};
```

2.3.2 Squaring operation in GF(2^m). OpenSSL also implements binary field arithmetic including a squaring operation to support the elliptic curves that are defined over binary fields. The elements of a binary field GF(2^m) are the binary polynomials, whose coefficients are in GF(2) = $\{0, 1\}$, of degree at most m-1:

$$\mathrm{GF}(2^m) = \{a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0 : a_i \in \mathrm{GF}(2)\}.$$

Addition of field elements involves the usual addition of polynomials, while multiplication is performed modulo the reduction polynomial $f(x)$. The squaring operation can be performed much faster by using a lookup table rather than multiplying two arbitrary polynomials [2].

OpenSSL has a lookup table-based implementation of squaring operation in GF(2^m), which is presented in Algorithm 5 (*i.e.,* GF2M_SQUARE algorithm). The lookup table, denoted by SQR_tb, is represented by an array of 64-bit words as shown in Listing 1. Let $(X_{w-1}, X_{w-1}, ..., X_0)$ be a sequence of words in the binary representation of a field element $X \in$ GF(2^m). Given an input X, the GF2M_SQUARE algorithm repeatedly transforms each word X_i ($0 \le i \le w-1$) into two consecutive words Y_{2i} and Y_{2i+1} through the loop. Specifically, in each iteration of the loop the word X_i is considered as a sequence of 16 nibbles $(x_{15}, ..., x_0)$. Each nibble x_j ($0 \le j \le 15$) is converted into its expanded word-wise counterpart W_j through a lookup to SQR_tb and a bit-wise left-shift operation. When accessing SQR_tb, the nibble x_j is used as an index to the table. The intermediate result $Y = (Y_{2w-1}, Y_{2w-2}, ..., Y_1, Y_0)$ is then reduced by a reduction polynomial f, which results in the output of the algorithm.

3 SEARCHING A NEW SIDE-CHANNEL

In this section, we propose a new search method for vulnerable locations in cryptographic libraries, which will be used to deliver our microarchitectural attack. Program analysis-based approaches [10, 11, 51] that analyze source codes or executable binary have been vividly proposed to detect the location. However, this approach only reveals vulnerable points directly related to the logical execution of the program. This makes one overlook certain vulnerable locations affected by indirect cache activities from the hidden processor components such as hardware prefetchers.

However, our approach captures all the vulnerable locations by exhaustive inspection of all memory lines belonging to the targeted library. Specifically, with at least two different inputs (*i.e.,* secret), we sample cache access traces for every line during the execution of the cryptographic operation. Any memory location where the relevant cache status is affected by the input will show significant difference in their cache traces. The difference in the traces is a strong indication of a cache side-channel.

In this section, we present our side-channel seeking method in detail with the case of the scalar point multiplication in the OpenSSL library. We then present our findings on the effects of hardware prefetching from the experimental results. Note that our side-channel seeking method is constructed as a generic and automated tool; therefore, it can be applied to any kind of cryptographic library besides OpenSSL and automate the finding procedures.

3.1 Method

The proposed method for searching a new side-channel proceeds through the following phases.

Phase 1: Collecting relevant memory lines. In this phase, we first build an address list of all the memory lines that belong to the main segments (*i.e.,* `.text` and `.data`) of the targeted cryptographic library. In Linux, a library is constructed according to Executable and Linkable Format (ELF), a common standard file format of executable binaries. Information about the runtime memory layout, such as the base addresses of segments and their sizes, is included in a program header of the ELF file. Using this information, the address list can be constructed with ease by iteratively increasing the base address of the text and data segments by line size (*e.g.,* 64 bytes) up to the size of each segment.

After that, the list is filtered so that it contains only *relevant memory lines*, which refers to a set of lines that have cache activity during the execution of the cryptographic operation of interest. We use the Flush+Reload technique (Section 2.1) to identify the relevant memory lines. Specifically, every memory line in the list is flushed from a cache before the execution of the cryptographic operation. When the execution is completed, we check whether each line has been loaded onto the cache. If so, it is marked as a relevant memory line and put into a result set.

In our experiment for OpenSSL 1.1.0g, we choose the EC_POINT_mul function from the library as the target operation. This function takes a scalar k and an elliptic curve point P as input, and then computes kP, for which a function of the Montgomery ladder multiplication is internally invoked.

We used the sect571r1 elliptic curve for performing the scalar point multiplication (*i.e.,* EC_POINT_mul function) in the experiment. As a result, we collected 261 relevant memory lines in total, among which 218 lines belong to the text segment and 43 lines belong to the data segment.

Phase 2: Tracing cache activities. For every relevant memory line collected in the previous phase, we now get the runtime traces of cache activities while the targeted function is running. Let us denote the set of relevant memory lines by $L = \{l_1, ..., l_\lambda\}$, where λ is the number of the memory lines and each $l_i \in L$ is an address of the line. Each status of the cache line (*i.e.,* a cache miss or hit) corresponding to l_i is repeatedly measured by using the Flush+Reload technique at every fixed slot interval during the execution of the function.

Since our final goal is to find out which relevant memory lines have secret-dependent cache traces, we need at least two different cache traces, each of which is obtained with a different input (*i.e.,* a scalar k). In fact, only two randomly chosen inputs k_1 and k_2 are sufficient to identify the secret-dependent traces. Let us denote a runtime cache trace for line l_i with an input k_j by a vector $\mathrm{S_{i,j}} = (s_1, ..., s_\tau)$, where τ is the length of the trace (τ is determined by both the elapsed time of the execution of the targeted function and a slot length of the Flush+Reload technique). Each s_t in $\mathrm{S_{i,j}}$ indicates the cache status at time t. Since there exists non-negligible cache noise in practice due to a diversity of unexpected system activities, the trace should be obtained by averaging from sufficient number of measurements to reduce the noise. Thus, to be more precise, s_t means a cache hit ratio, which is the count of the observed cache hits divided by the total number of measurements, at time t.

When conducting this phase, we use a scalar k of 60 bits in length (*i.e.,* $|k| = 60$). The bit length of our choice is enough to produce good secret-dependent traces while incurring less computational burden in the next phase. With the selected curve, the EC_POINT_mul function on an arbitrary scalar of 60 bits in length takes around 490,000 cycles on average in the experimental environment with Intel Xeon E5-2620v4 processor. Based on the measured elapsed time, we pick a slot length of 7,000 cycles, which gives enough temporal resolution for probing individual operations per bit of the scalar. The length of the slot in turn determines the length of the single trace to be $\tau = 70$.

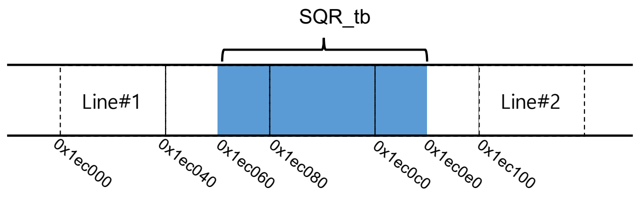


Figure 1: Memory layout of SQR_tb in OpenSSL 1.1.0g

In order to proceed to the next phase, we need to collect a sufficient number of samples for each trace $\mathrm{S_{i,j}}$. By repeating the collection of the trace multiple times (say N), we can produce multiple samples for a trace $\mathrm{S_{i,j}}$, denoted by $\mathrm{S_{i,j}^{(1)}}, ..., \mathrm{S_{i,j}^{(N)}}$.

Phase 3: Finding secret-dependent traces. In the final phase, we try to find out which relevant memory lines in L have runtime cache traces sensitive to an input. The basic idea underlying this phase stems from the reasoning that if a cache trace at line l_i has dependency on an input, then the samples obtained from one input k_1, *i.e.,* $\mathrm{S_{i,1}^{(1)}}, ..., \mathrm{S_{i,1}^{(N)}}$, would be more similar to each other than those obtained from a different input k_2, *i.e.,* $\mathrm{S_{i,2}^{(1)}}, ..., \mathrm{S_{i,2}^{(N)}}$.

The main approach is to perform cluster analysis on the set of all trace samples for the memory line. If the set can be clustered into two distinct groups, then it is inferred that the corresponding memory line has a secret-dependent cache trace. For this, we use K-means clustering algorithm [1, 40]. This algorithm aims to partition a set of unlabeled vectors into K clusters, in which each vector belongs to one of the clusters with the nearest Euclidean distance. Specifically, for each line, we invoke the K-means algorithm (with a parameter $K = 2$) by providing a set of (unlabeled) vectors $\left\{\mathrm{S_{i,1}^{(1)}}, ..., \mathrm{S_{i,1}^{(N)}}, \mathrm{S_{i,2}^{(1)}}, ..., \mathrm{S_{i,2}^{(N)}}\right\}$ as the input. Given the input, this algorithm attempts to find two cluster centroids that best partition those vectors, and labels each vector based on the partition as one of the names of those clusters, C_1 or C_2. Upon completion, it returns a list of labels as output. We evaluate the clustering result for traces of the line l_i by the score σ_i, which is defined as follows:

$$\sigma_i = \frac{\textit{\# of vectors belong to } S_{i,\alpha} \textit{ among those labeled as } C_\beta}{\textit{\# of vectors labeled as } C_\beta},$$

where $C_{\beta \in \{1,2\}}$ refers to the largest cluster of C_1 and C_2, and $\mathrm{S}_{\mathrm{i},\alpha}$ refers to the kind of trace vectors obtained from an input $k_{\alpha \in \{1,2\}}$, which comprises the majority in C_β. For instance, a line l_i that has a secret-dependent trace would have the highest score $\sigma_i = 1$ in the ideal case, which means that all the vectors labeled as C_β by the K-means algorithm actually belong to the same trace $\mathrm{S}_{\mathrm{i},\alpha}$.

In the experiment regarding the EC_POINT_mul function in OpenSSL 1.1.0g, we conducted the clustering analysis for all the relevant memory lines in L with $N = 100$ trace samples for each scalar input. As a result, we discovered that only 2 lines among 261 relevant memory lines have significant scores of larger than 0.9 on average. This strongly indicates that during the execution of the function, these two memory lines experienced cache activities correlated to the value of scalar k.

Table 2: Memory lines showing secret-dependent traces

	OpenSSL 1.1.0g	OpenSSL 1.0.1e
ec_GF2m_montgomery _point_multiply (in ec2_mult.c)	-	†0xbe440, ‡0xbe4c0
SQR_tb (in bn_gf2m.c)	0x1ec000, 0x1ec100	0x16c680, 0x16c780

†: ec2_mult.c:275-276, ‡: ec2_mult.c:281 (line numbers in source code)

Comparison to the previous work. Our searching method is somewhat similar to the existing trace-based approach, *i.e.,* Cache Template Attacks [24]. That approach utilizes a profiled matrix, where each element represents cache hit ratio for a memory address and an event (*i.e.,* input). The side-channel is detected if there is a variance in the ratio on different inputs. The ratio is defined as either a simple value or a time-dependent function. However, specific attacks in the latter case were not presented in detail. Our work differs from the previous work in that we realize the idea of using the ratio with time-dependency for searching side-channel. In the proposed method, the cache hit ratio is represented as a time series, *i.e.,* a vector $\mathsf{S}_{i,j} = (s_1, s_2, \ldots, s_\tau)$ for a line l_i and an input k_j. In fact, the time series contains more information than a simple value (*i.e.,* scalar). For instance, a line l_i with vectors $\mathsf{S}_{i,1} = (0.5, 0, 0, 0)$ and $\mathsf{S}_{i,2} = (0, 0, 0, 0.5)$ on different inputs k_1 and k_2 clearly indicates side-channel, while corresponding scalars would be indistinguishable. Contrary to the previous work, we also present a detailed method to identify the variance among vectors in general cases by employing K-means algorithm. Therefore, our work specifies a more fine-grained method to detect the side-channel.

3.2 Observations

Table 2 presents the searching results for the scalar point multiplication in the OpenSSL 1.1.0g library, described in the previous section. In order to get some insight into the comparison, we also show the result from OpenSSL 1.0.1e with the same experimental setting. It is the previous version of the targeted OpenSSL library with the vulnerable Montgomery ladder implementation for cache side-channel attacks [56].

In OpenSSL 1.1.0g, two identified memory lines are addressed at 0x1ec000 and 0x1ec100, both of which are close to the location of SQR_tb (Listing 1), a lookup table used for the GF2M_SQUARE operation (Algorithm 5). Fig. 1 depicts the memory layout of SQR_tb along with those two memory lines (denoted by Line#1 and #2 in the figure). SQR_tb is 128 bytes in length (See Listing 1) and placed across three memory lines at addresses ranging from 0x1ec040 to 0x1ec0e0. We can observe that Line#1 and #2 are adjacent to both sides of the lookup table.

On the other hand, a total of four memory lines have been identified to have secret-dependent traces in OpenSSL 1.0.1e. Among them, two memory lines are addressed at 0x16c680 and 0x16c780, both of which are placed adjacently on both sides of SQR_tb similar to the case of OpenSSL 1.1.0g. The other two memory lines are newly identified at 0xbe440 and 0xbe4c0. Binary analysis on the library reveals that a portion of execution codes of the Montgomery ladder implementation (Algorithm 1) is placed at those memory locations. More specifically, those memory lines are located at both sides of the code blocks in a conditional branch (*i.e.,* Lines 6 and 8 in Algorithm 1), which is dependent on the bit of a scalar k. One of the memory lines, at 0xbe440, is taken when the bit is set to 1 while the other, at 0xbe4c0, is taken in the opposite case. This result is the same as the work of Yarom *et al.,* [56] which disclosed a cache-side channel attack on this implementation.

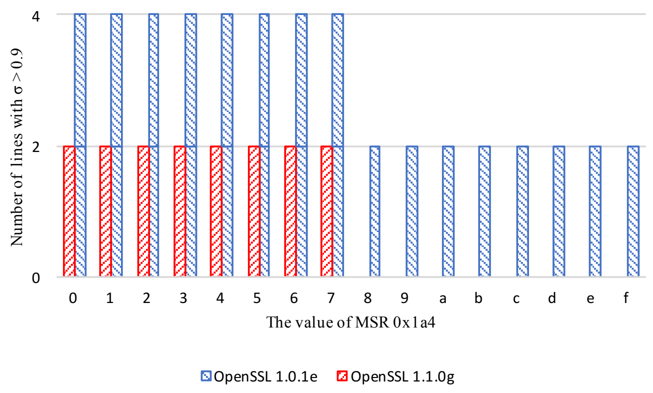


Figure 2: Number of identified lines ($\sigma > 0.9$) according to the value of MSR 0x1a4

Unlike the case of Montgomery ladder implementation, the lines near the SQR_tb table, at 0x1ec000 and 0x1ec100 in the version 1.1.0g and at 0x16c680 and 0x16c780 in the version 1.0.1e, contain neither machine code nor data related to the conduction of the operation. We confirmed this through program analysis on the library by means of reverse-engineering and debugging. This observation leads to a conclusion that cache activities on those lines (*e.g.,* Line#1 and #2 in Fig.1) are actually introduced by the hardware-based data prefetching mechanism.

Validation of our reasoning. As described in Section 2.2, each Intel core is equipped with four kinds of hardware prefetchers with various cache prefetching strategies at different locations. These prefetchers can be controlled independently by four bits of the MSR with address 0x1a4 (see more details in Section 2.2). In order to validate our reasoning about the effect of cache prefetching, we check whether these prefetchers affect the secret-dependent cache activities near the lookup table by utilizing the MSR, and if so, figure out which one actually does.

In the system setup with Intel Xeon E5-2620v4, we conducted the procedure described in Section 3.1 for both versions of the OpenSSL library with each bit of the MSR being manipulated. The experimental results are presented in Fig. 2. The graph in the figure shows the number of lines identified to have secret-dependent cache traces (*i.e.,* the score σ is more than 0.9) according to the value of the MSR 0x1a4. Note that each MSR value represents the combination of status of the hardware prefetchers. For instance, the value of 0xF indicates that all the prefetchers are disabled, and the value of 0x0 indicates that all prefetchers are enabled.

Table 3: Access patterns with a regular stride at the first load instruction of GF2M_SQUARE in the sequence of MADD operations (on the left side) and the input value on the 20-th invocation (on the right side)

Sequence	Access pattern (indices of SQR_tb lookups)	Stride	Direction
20	$0 \rightarrow 4 \rightarrow 5 \rightarrow \underline{8 \rightarrow 9 \rightarrow a} \rightarrow a \rightarrow 2 \rightarrow d$	8	Forward
22	$0 \rightarrow c \rightarrow \underline{9 \rightarrow 8 \rightarrow 7} \rightarrow 8 \rightarrow 2 \rightarrow 5 \rightarrow d$	8	Backward
25	$0 \rightarrow 0 \rightarrow 2 \rightarrow 3 \rightarrow \underline{1 \rightarrow 4 \rightarrow 7} \rightarrow e \rightarrow a$	24	Forward
26	$0 \rightarrow d \rightarrow 1 \rightarrow \underline{9 \rightarrow 7 \rightarrow 5} \rightarrow a \rightarrow a \rightarrow 6$	16	Backward
44	$\underline{0 \rightarrow 4 \rightarrow 8} \rightarrow 8 \rightarrow 5 \rightarrow d \rightarrow a \rightarrow 2 \rightarrow f$	32	Forward
55	$0 \rightarrow 8 \rightarrow e \rightarrow 1 \rightarrow \underline{7 \rightarrow 8 \rightarrow 9} \rightarrow c \rightarrow 6$	8	Forward

Input of GF2M_SQUARE at sequence 20	
X_8:$\underline{0}$1306E5A75514CF1	X_7:$\underline{4}$74F813861F0EE7C
X_6:$\underline{5}$F1EADE8248BFB8B	X_5:$\underline{8}$2459457243E0058
X_4:$\underline{9}$8BC1F8FF8E0E86E	X_3:$\underline{A}$11D81BEC146C7D1
X_2:$\underline{A}$034E951A17B407E	X_1:$\underline{2}$C4D730E4550C751
X_0:$\underline{D}$09768D838984148	

We can clearly observe the difference between the values ranging from 0x0 to 0x7 and the values ranging from 0x8 to 0xF. With MSR values of more than 0x7, no cache activities are observed on the lines near SQR_tb, which results in the decrement of the number of lines by 2 in the graph. This indicates that hardware prefetchers actually have an effect on the cache activities and produce secret-dependent cache traces on the relevant memory lines. Furthermore, we can learn from the result that among the four kinds of prefetchers, *IP-based stride prefetcher*, whose control bit corresponds to Bit #3 in the MSR, solely affects the memory lines near SQR_tb.

4 ANALYSIS ON STRIDE PREFETCHING

In this section, we consider the behavior of the IP-based stride prefetcher in Intel processors, which has been identified in the previous section as a hidden source of observable cache activities. Concretely, we examine its impact on producing cache traces on a couple of memory lines adjacent to the squaring lookup table (SQR_tb) when performing scalar point multiplication.

As described in Section 2.2, the IP-based stride prefetcher basically follows a strategy of detecting the stride on load operations in a loop structure. However, any details of the behavior of the prefetcher, *i.e.,* under what conditions the prefetching is triggered and what memory line is fetched into the cache, are publicly unknown. Hence, we have to infer the behaviors through observations from in-depth experiments.

Experimental analysis. The prefetching activity is observed on memory lines near the SQR_tb lookup table, which is used by the GF2M_SQUARE algorithm (Algorithm 5). The squaring algorithm is one of the primitive operations for both MADD (Line 7 in Algorithm 4) and MDOUBLE (Lines 2,3,5,6 in Algorithm 3) algorithms. Hence, when running the Montgomery ladder multiplication, a series of GF2M_SQUARE operations is necessarily invoked for the computation of each bit of the scalar.

Given an input $X \in \mathrm{GF}(2^m)$, the GF2M_SQUARE algorithm computes the squaring of X by iterative lookups to SQR_tb through a loop. Specifically, for each iteration of the loop, 16 lookups to SQR_tb occur in total, each of which is indexed by four bits of a word X_i (Lines 5 to 20 in Algorithm 5). In machine code, each table lookup is translated into a load instruction to fetch a word from the memory. The memory address of the word is calculated from the base address of SQR_tb and the corresponding index. The number of iterations of the loop is determined by w, the length of X in a word. For instance, with the sect571r1 curve, an input

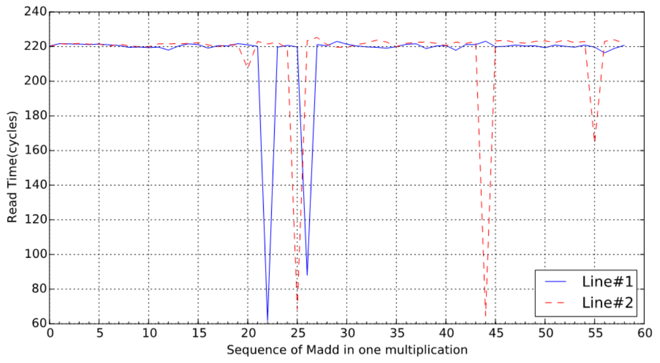


Figure 3: Average memory access time on Line#1 and Line#2 after execution of the first load instruction of GF2M_SQUARE in the sequence of MADD operations (measured on Xeon E5-2620v4)

X of GF2M_SQUARE is 571 bits in length, which results in $w = 9$ iterations of the loop.

We demonstrate how to examine the behavior of the stride prefetcher in the squaring algorithm. Our approach is, in short, to analyze the observed prefetching activities by comparing it to the ground truth of memory access patterns on SQR_tb lookups.

First, we obtain the full sequence of input data of the GF2M_SQUARE algorithm during execution to use as the ground truth. For this, we run the Montgomery ladder multiplication algorithm with a fixed point P_1 in the sect571r1 curve. We choose an arbitrary scalar k of 60 bits in length as another input. While running the algorithm, the values of input X are traced on every invocation of the GF2M_SQUARE algorithm. From this trace, we can get the actual memory access patterns of every load instruction corresponding to SQR_tb lookups. More specifically, the access patterns for 16 load instructions are collected for each invocation of GF2M_SQUARE.

Once the memory access patterns have been collected, we look for patterns among them that have a regular stride with confidence $c = 2$ (*i.e.,* patterns containing at least three consecutive accesses with constant stride). We choose $c = 2$ when searching the patterns since it is the minimal condition to have a regular stride.

Table 3 shows the searching result for the first load instruction (Line 5 in Algorithm 5) of the GF2M_SQUARE operation. The result was obtained when the squaring operation was invoked by

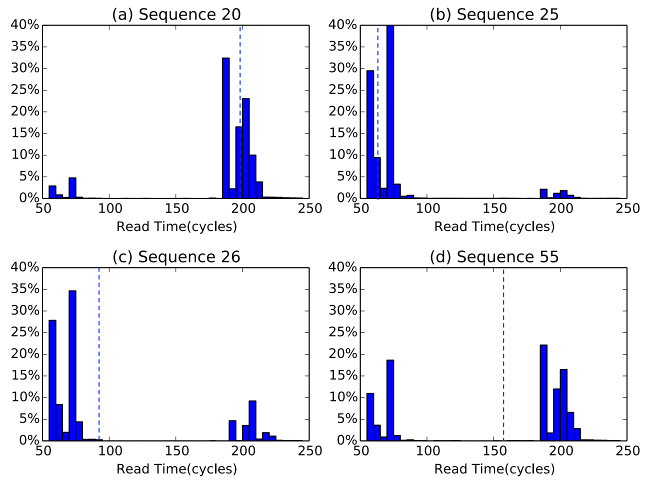


Figure 4: Distributions of access times for sequences that have a regular stride with $c = 2$ (The dashed line represents the mean of the distribution)

Madd (Line 7 in Algorithm 4). Given the length of the scalar k (*i.e.,* 60 bits), we have 59 access patterns in total for the first load instruction. Among them 6 patterns are identified to have a regular stride (underlined in Table 3). For instance, the input of X at the 20-th invocation of GF2m_Square (*i.e.,* Sequence 20) is given on the right side of Table 3. Its value consists of 9 words (X_8 - X_0), and the most significant 4 bits of each word are used as a lookup index to SQR_tb at the first load instruction. The access pattern at sequence 20 has sequential accesses with indices 0x8, 0x9, and 0xa that form a regular stride of 8 bytes. The sequence of indices is incremental; therefore, the stride is in a forward direction. The next access pattern, found at sequence 22, has accesses with indices 0x9,0x8, and 0x7 that form a regular stride in a backward direction. These identified access patterns will serve as the ground truth for analysis on the observed cache activities.

Now, we examine the cache traces observed on two memory lines, Line#1 at 0x1ec000 and Line#2 at 0x1ec100, located at both sides of SQR_tb. In the original implementation of OpenSSL library, the GF2m_Square algorithm has more than a dozen load instructions that affect the cache activities simultaneously on those lines. To infer the prefetching behavior for an individual instruction, we dismantle the observed cache traces. In this case, we focus on the cache activities caused by the first load instruction. For this purpose, the remaining load instructions in the implementation are modified so that they no longer affect the cache activities on those lines while permitting the multiplication to work correctly.

Fig. 3 shows the average memory access time measured on Line#1 and Line#2 at the end of every execution of Madd operation during the multiplication. A time sequence that takes shorter cycles for access than the others indicates that the line is prefetched to the cache at that time. From the graph, we observe a total of 6 sequences that show shorter access times on those lines. For Line#1, there are a couple of low peaks at sequence 22 and 26, both of which have access patterns with regular strides in a backward direction,

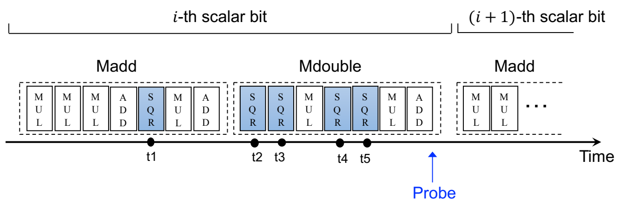


Figure 5: Invocations of GF2m_Square (SQR) in a 1-bit processing sequence of the Montgomery ladder multiplication

as shown in Table 3. Line#1 is located on the left side of SQR_tb, which is at a lower address than the address of the lookup table. It is inferred from the observation that IP-based stride prefetching is triggered when a regular stride with $c = 2$ is detected, and if it is in a backward direction, an adjacent memory line at a lower address is fetched to the cache.

The observation at the other line also conforms to our inference. For Line#2, which is located on the right side of SQR_tb, four low peaks are observed at sequences 20, 25, 44, and 55. All these peaks are matched to sequences in Table 3 that have regular strides with $c = 2$ in a forward direction, which triggers the prefetcher to fetch an adjacent line at an address higher than SQR_tb.

Dynamic behavior of the prefetcher. It is worth noting that sequences 20 and 55 in Fig.3 show relatively high average memory access time than the other sequences with low peaks. Since most parts of the IP-based stride prefetcher in Intel processors have not been disclosed yet, we have a certain amount of uncertainty about the behavior of the prefetcher. However, from observing the distributions of access times on several sequences, which are shown in Fig. 4, we believe that the stride prefetcher does not behave in a deterministic way. For instance, sequences 20 and 55, shown in Fig.4(a) and Fig.4(d), respectively, follow a distribution that is quite different from that of the other sequences, shown in Fig.4(b) and Fig.4(c), in which almost every memory access results in a cache hit. This implies that the decision of prefetching is made based on not only the observed access patterns but also unknown dynamic mechanism (*e.g.,* using a history buffer or a probabilistic model), which should be further investigated in a future work. The analysis on the uncertain behavior would require a more systematic approach. One possible direction is to reverse-engineer the internal workings of the hardware prefetcher by using hardware performance counters via Intel Performance Monitoring Unit [27, 28], which provides lots of useful information on internal states of the processor.

5 CACHE PROFILING ON SCALAR BITS

The analysis shown in the previous section implies that the behavior of the stride prefetcher during execution of the Montgomery ladder multiplication is dependent on the input of a scalar. That is, each 1-bit processing sequence of the loop in the multiplication algorithm shows an unique prefetching behavior according to the value of the bit of the scalar. In this section, we present our findings in detail. Furthermore, we show that cache activities due to prefetching can be used for distinguishing between the executions of branchless Montgomery ladder multiplication with two different scalars.

Table 4: An example of measurements for the first 20 iterations of the Montgomery ladder multiplication (measured on Xeon E5-2620v4)

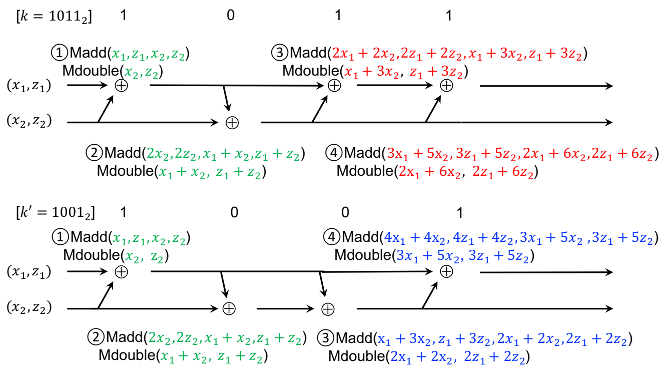
Slot	Cache hit ratio (ρ)		Slot	Cache hit ratio (ρ)	
	Line#1	Line#2		Line#1	Line#2
1	73.58	15.99	11	73.71	25.47
2	20.97	41.17	12	25.98	66.89
3	84.88	20.04	13	58.58	77.82
4	66.95	84.05	14	40.64	41.94
5	79.3	24.06	15	70.38	68.6
6	83.44	83.54	16	25.81	29.28
7	29.81	3.79	17	76.5	67.98
8	20.34	82.66	18	26.38	81.65
9	4.89	81.17	19	62.8	81.03
10	72.58	31.67	20	37.13	76.07

5.1 Measurement on a 1-bit processing sequence

In the branchless Montgomery ladder algorithm (Algorithm 2), the Madd operation takes a list of four arguments (X_{c_i}, Z_{c_i}, $X_{\overline{c_i}}$, $Z_{\overline{c_i}}$), and the Mdouble operation takes two arguments ($X_{\overline{c_i}}$, $Z_{\overline{c_i}}$) at sequence i, where $c_i = k_i + 1$ and $\overline{c_i} = \overline{k_i} + 1$ (k_i is the value of the i-th scalar bit).

While the algorithm is running, the GF2m_Square function is invoked on every sequence of the Madd and Mdouble operations. For the Madd operation, the squaring function is called with an input derived from X_{c_i}, Z_{c_i}, $X_{\overline{c_i}}$ and $Z_{\overline{c_i}}$ (see Algorithm 4). For Mdouble operation, the function is invoked four times with inputs derived from $X_{\overline{c_i}}$ and $Z_{\overline{c_i}}$ (see Algorithm 3). Note that when executing the GF2m_Square function, all the indices used for lookups to SQR_tb are taken from the input of the function. Hence, the behavior of the stride prefetcher at the i-th sequence of the 1-bit processing in the Montgomery ladder multiplication is dependent on an ordered list of inputs ($X_{c_i}, Z_{c_i}, X_{\overline{c_i}}, Z_{\overline{c_i}}$), which is determined by the scalar bit k_i.

We can measure the prefetching activities by probing the cache status resulting from the executions of the GF2m_Square function at every sequence. As illustrated in Fig. 5, each iteration of the Montgomery ladder multiplication makes a total of five invocations of the GF2m_Square function. Due to the limited resolution of the Flush+Reload probing technique, it is difficult to observe all the prefetching activities separately. Instead, the probing is performed once for each sequence, which will result in measurements on the aggregated activities by a series of GF2m_Square invocations. In order to make the measurement results of consecutive sequences distinguishable, the probing is done at the end of the Mdouble execution (*i.e.,* the time after $t5$ in Fig.5). The timing information can be obtained by probing another memory line containing the epilogue stub of the Mdouble function. Owing to the constant-time execution of the Montgomery ladder multiplication, repeated probings at every fixed time slot allow consistent measurements on the cache activity at that time of every iteration.



$[k = 1011_2]$
1
0
1
1
①Madd(x_1,z_1,x_2,z_2)
Mdouble(x_2,z_2)
③Madd($2x_1 + 2x_2$,$2z_1 + 2z_2$,$x_1 + 3x_2$,$z_1 + 3z_2$)
Mdouble($x_1 + 3x_2$, $z_1 + 3z_2$)
(x_1, z_1)
(x_2, z_2)
②Madd($2x_2$,$2z_2$,$x_1 + x_2$,$z_1 + z_2$)
Mdouble($x_1 + x_2$, $z_1 + z_2$)
④Madd($3x_1 + 5x_2$,$3z_1 + 5z_2$,$2x_1 + 6x_2$,$2z_1 + 6z_2$)
Mdouble($2x_1 + 6x_2$, $2z_1 + 6z_2$)
$[k' = 1001_2]$
1
0
0
1
①Madd(x_1,z_1,x_2,z_2)
Mdouble(x_2, z_2)
④Madd($4x_1 + 4x_2$,$4z_1 + 4z_2$,$3x_1 + 5x_2$,$3z_1 + 5z_2$)
Mdouble($3x_1 + 5x_2$, $3z_1 + 5z_2$)
(x_1, z_1)
(x_2, z_2)
②Madd($2x_2$,$2z_2$,$x_1 + x_2$,$z_1 + z_2$)
Mdouble($x_1 + x_2$, $z_1 + z_2$)
③Madd($x_1 + 3x_2$,$z_1 + 3z_2$,$2x_1 + 2x_2$,$2z_1 + 2z_2$)
Mdouble($2x_1 + 2x_2$, $2z_1 + 2z_2$)

Figure 6: Operation sequences in the branchless Montgomery ladder multiplication (Algorithm 2) on two different scalars

Table 4 shows an example of measurements on prefetching activities for the first 20 iterations (*i.e.,* time slots) of the Montgomery ladder multiplication. We use the metric of cache hit ratio (ρ), which is defined as the number of observed cache hits divided by the total number of measurements. Each slot shows distinguishable values of ρ on both cache lines (note that prefetching activities on Line#1 and Line#2 are independent of each other). A low ratio ($\rho < 5$) indicates that there is no prefetching activity on the cache line during that time slot, while a high ratio ($\rho > 70$) is a strong indication that at least one of the invocations during times $t1$ to $t5$ has triggered the stride prefetching. In addition, we observe the other cases with $20 < \rho < 70$ in the remaining measurement results. From the results of an empirical analysis with memory access patterns on them, we conclude that they happened as a result of the dynamic behavior of the stride prefetcher, as noted in the previous section.

5.2 Distinguishability between two scalars

The measurement of the prefetching activity on the i-th iteration of the Montgomery ladder algorithm can be modeled as a map $\mathcal{M}$

$$(\rho_1, \rho_2) \leftarrow \mathcal{M}(\mathbf{a_i}), \quad \mathbf{a_i} = \left(X_{c_i}, Z_{c_i}, X_{\overline{c_i}}, Z_{\overline{c_i}}\right), \tag{1}$$

where ρ_1 and ρ_2 refer to the cache hit ratio on Line#1 and Line#2, respectively. $\mathcal{M}$ is defined such that given an input vector $\mathbf{a_i}$ of sequence i, it outputs a tuple of the values of two cache hit ratios.

By using the map, we can distinguish between two executions of the branchless implementation of Montgomery ladder multiplication on different scalars. We consider an example of multiplications with two 4-bit scalars $k = 1011_2$ and $k' = 1001_2$. Fig. 6 shows the sequences of the Madd and Mdouble operations when running Algorithm 2 with inputs of those scalars. Suppose that the iteration begins with the initial values $(X_1, Z_1) = (x_1, z_1)$ and $(X_2, Z_2) = (x_2, z_2)$. Since k and k' share the same values for the two leftmost bits, they both take the same inputs $\mathbf{a_1} = (x_1, z_1, x_2, z_2)$ and $\mathbf{a_2} = (2x_2, 2z_2, x_1 + x_2, z_1 + z_2)$ in the first two iterations. This leads to the same prefetching activities on them. On the other hand, they differ in the value of the third bit, thereby having the input $\mathbf{a_3} = (2x_1 + 2x_2, 2z_1 + 2z_2, x_1 + 3x_2, z_1 + 3z_2)$ for k and $\mathbf{a_3} = (x_1 + 3x_2, z_1 + 3z_2, 2x_1 + 2x_2, 2z_1 + 2z_2)$ for k', which results in

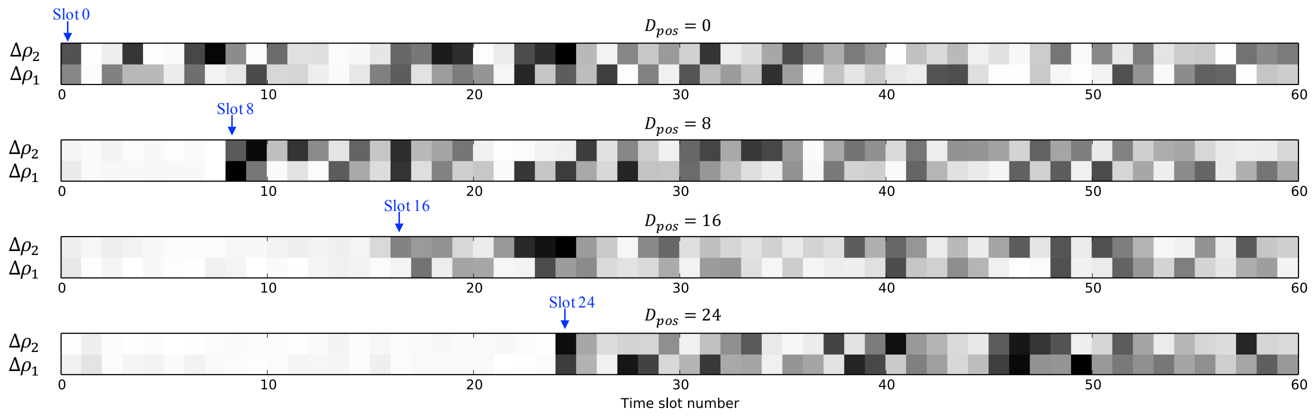


Figure 7: Differences of ρ_1 and ρ_2 between two scalars that share the D_{pos} leftmost bits

different prefetching activities. At the fourth bit, those scalars have the same value. In this case, however, they result in different behaviors since the arguments of the MADD and MDOUBLE functions have been accumulated from the results of previous iterations.

The above example suggests a crucial aspect of the prefetching activities in the branchless Montgomery ladder multiplication. That is, by comparing the sequences of $\mathcal{M}(\mathbf{a}_1), \mathcal{M}(\mathbf{a}_2), \ldots$ between two executions, we can learn the following:

- Whether two scalars k and k' have the same leftmost bits.
- If so, how many bits are actually shared between them.

Another example illustrated in Fig.7 supports these findings. Let us consider two scalars k and k' that share the D_{pos} leftmost bits ($D_{pos} \geq 0$). Fig.7 shows the differences of ρ_1 (as well as ρ_2) between the two scalars at each 1-bit processing sequence of the branchless Montgomery ladder multiplication. Measurements on the $(i+1)$-th scalar bit are shown at a time slot of number i in the graph. Bright color shows that there is almost no difference between the values at that time slot, while significant difference is marked with dark color. It is noticeable that for all the four cases, the D_{pos} leftmost slots show no difference in both ρ_1 and ρ_2, compared to that shown by the remaining slots.

6 EXPLOITATION

6.1 Recovering unknown scalar bits

The value of the scalar used for the scalar point multiplication is treated as a secret in most cryptographic algorithms. The distinguishability given by the map $\mathcal{M}$ can be exploited by an attacker who attempts to reveal unknown bits of the secret scalar used in the branchless implementation of Montgomery ladder multiplication.

In this section, we demonstrate how an attacker can recover unknown scalar bits by utilizing information leakage from the prefetcher.

Method. The basic idea of our method is to try to recover the leftmost bits by bits of the unknown scalar through iterations. Before giving details of the method, we present some notations. Let k be a bit string of the secret scalar to be recovered by our attack and n_k be the length of k in bits. Let σ be a substring of k with length n_σ starting from the leftmost bit. In other words, we denote by σ the recovered substring from k after iterations of our attack. Initially, we set $\sigma = \mathsf{nil}$ and $n_\sigma = 0$. In each iteration of the attack, we try to recover a substring of length n_c ($0 < n_c \leq n_k$) from k. The length n_c may be chosen as an attack parameter.

The attack proceeds through the following steps.

Step 1: Measure prefetching activities (*i.e.,* cache usages on Line#1 and Line#2) yielded during the victim's execution of the scalar point multiplication. As a result, the attacker obtains a list $\mathcal{V} = (\mathcal{M}(\mathbf{a}_1), \ldots, \mathcal{M}(\mathbf{a}_{n_k}))$.

Step 2: Construct a list of candidate scalar k' as follows
- (a) Initialize $\mathcal{D} \longleftarrow \{\}$.
- (b) For each substring $c \in \{0,1\}^{n_c}$, construct a candidate bit string $k' = \sigma \| c \| r$, where r is an arbitrary bit string of length $n_r = n_k - n_\sigma - n_c$, and insert k' into $\mathcal{D}$.

Step 3: For each $k' \in \mathcal{D}$
- (a) Execute the multiplication with k' on the attacker's machine and obtain a list $\mathcal{A} = \left(\mathcal{M}(\mathbf{a}'_1), \ldots, \mathcal{M}(\mathbf{a}'_{n_k})\right)$.
- (b) Compute the differences

$$(\Delta_1, \Delta_2) \longleftarrow \frac{1}{n_c} \sum_{i=n_\sigma+1}^{n_\sigma+n_c} \left|\mathcal{M}(\mathbf{a}_i) - \mathcal{M}(\mathbf{a}'_i)\right|.$$

Step 4: Select $k' (= \sigma \| c \| r)$ in $\mathcal{D}$ such that both Δ_1 and Δ_2 are the least values among the candidates in $\mathcal{D}$, and update σ and n_σ as follows

$$\sigma \longleftarrow \sigma \| c,$$
$$n_\sigma \longleftarrow n_\sigma + n_c.$$

Step 5: Repeat Step 2 until $n_\sigma = n_k$.

Step 6: Output σ as a result of the recovery of the secret scalar.

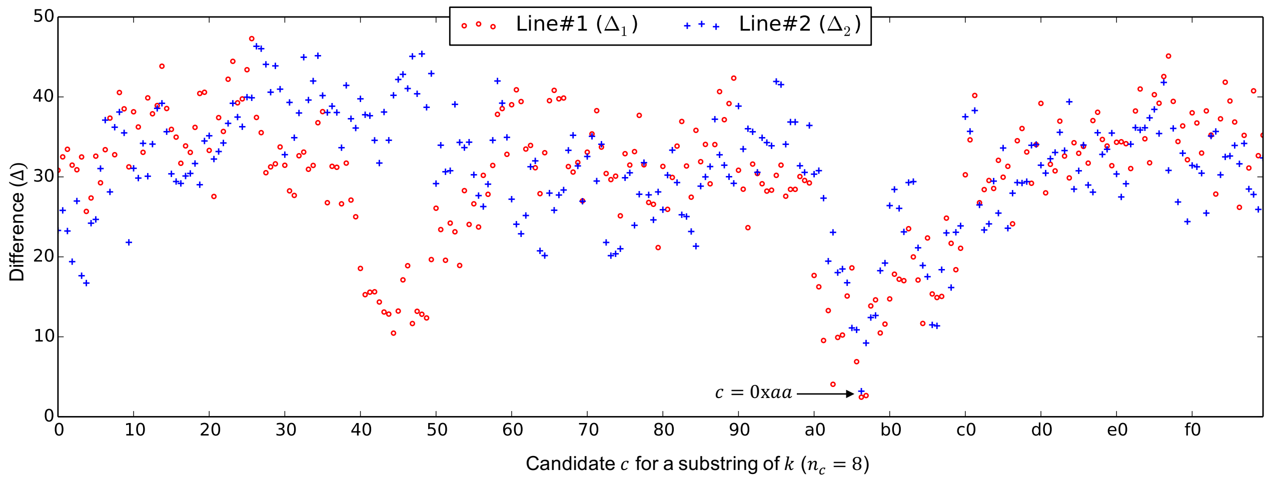


Figure 8: Evaluation of the difference in prefetching activities for all possible 8-bit substrings of k

Error handling. In the ideal case of performing Step 4 in the above method, the attacker will always find out a unique k' in $\mathcal{D}$ that has the least values of both Δ_1 and Δ_2. In most practical cases, however, system noise is inevitably introduced during the measurement of prefetching activities due to a variety of unexpected system behaviors. This may lead to an error in selecting the correct scalar in $\mathcal{D}$. For instance, it would be possible to have two candidates k' and k'' such that k' has the least value of Δ_1 while k'' also has the least value of Δ_2, which makes it infeasible to determine the correct one.

Such errors can be handled by reducing the noise. Specifically, the attacker repeats the process of Step 3 and Step 4 and counts a candidate that has the least value of Δ_1 or Δ_2. After that, by using a simple majority rule, the attacker chooses the candidate that has been counted the most number of times.

6.2 Attacking ECDH

In this section, we deliver an attack against the ECDH key agreement protocol implemented in the OpenSSL library.

ECDH. Elliptic Curve Diffie-Hellman (ECDH) is a variant of the Diffie-Hellman algorithm using Elliptic curve cryptography. In general case ECDH is used for the purpose of key agreement between two entities. Suppose that users $\mathcal{A}$ and $\mathcal{B}$ want to establish a shared secret with each other. $\mathcal{A}$ and $\mathcal{B}$ have private keys $\text{Pri}_A = k_A$ and $\text{Pri}_B = k_B$, and public keys $\text{Pub}_A = k_A P$ and $\text{Pub}_B = k_B P$, respectively, where P is the generator for the curve. By exchanging their public keys with each other, they can compute the shared secret key $s = k_A \text{Pub}_B = k_B \text{Pub}_A = k_A k_B P$.

Experimental setup. We target the OpenSSL that implements the branchless Montgomery ladder multiplication algorithm. As described in Section 1, we conducted experiments on OpenSSL 1.1.0g, which is the latest version of OpenSSL at the time of writing this paper. All the experiments were performed on a server equipped with a Xeon E5-2620v4 Broadwell processor and 16GB of memory, running 64-bit Ubuntu 16.04.4 LTS. Each CPU core has an 8-way 32KB L1 data/instruction cache, an 8-way 256KB L2 unified cache, and a 20-way 20MB unified LLC cache shared by all cores across the CPU.

Since OpenSSL 1.1.0g is not contained as a bundle in Ubuntu 16.04, we built the library by downloading it from the official website of OpenSSL. The library was built with debugging symbols on the executable for the purpose of identifying locations of the Flush+Reload probing. Note that debugging symbols are not loaded during run time, thus they do not affect the feasibility of the attack as well as the performance of the victim application.

Victim application. For the experiment, we built our own victim application that performs the ECDH algorithm. It is implemented in C with OpenSSL 1.1.0g being linked to the executable. For the ECDH key generation, we used the ECDH_compute_key function in the library. This function internally makes a call to the EC_POINT_mul function by passing a private key as a scalar k, through which our targeted function is consequentially invoked. The victim application uses sect571r1 elliptic curve for the multiplication. We generated a pair of public and private key for the victim by using the OpenSSL command line tool. The private key, *i.e.,* a scalar k that we try to recover, is 568 bits in length.

The victim application waits for key generation requests from a spying process. Upon receipt of the request as well as the attacker's public key, it performs ECDH key generation, *i.e.,* the scalar point multiplication, then responds to the attacker with the result. We consider an attack scenario where an attacker's spying process is co-located with the victim process, but on different physical cores. The communication between the spying process and the victim is carried out through socket communications.

Results. We implemented the attack described in the previous section and evaluated it against the victim application. In the experiment, we chose the attack parameter $n_c = 8$. That is, the spying process attempted to recover 8 bits of the scalar k from the leftmost bit for each iteration. We performed the attack through 71 iterations of Steps 2-5 in the above method to recover all the bits of k. In each iteration, prefetching activities were measured for all 256 possible candidates of the scalar. To measure the cache hit ratio on both Line#1 and Line#2, *i.e.,* evaluate $\mathcal{M}(\cdot)$, we captured 1,000 traces for each 1-bit processing sequence of the scalar point multiplication.

Fig.8 shows the experimental result of recovering an 8-bit substring of the scalar, which has a value of 0xaa. After processing Step 3, we found that a candidate c=0xaa has the least values of both Δ_1 and Δ_2 among 256 candidates, which is the correct answer for the unknown scalar bits.

In the case where no measurement error occurred, it took around 15 min to recover 8-bit data from the scalar. While conducting the attack experiments, we experienced a total of two measurement errors over 71 iterations of the recovering process. For the overall process, the attack took around 19 CPU hours to recover all the bits of k. This includes the elapsed time to conduct the process of Step 1, which took only around 3 min to obtain the trace of the victim's prefetching activities.

We emphasize that Step 1 has been processed only once during the entire attack execution. That is, our attack does not require any involvement of the victim application when proceeding with the recovery process (*i.e.,* Steps 2-6). This makes our method a more pragmatic attack on real-world applications.

6.3 Impact on the other algorithms

We demonstrated that our attack is able to successfully recover private keys in ECDH algorithm. However, the presented attack is not only limited to this specific algorithm. Now we discuss the impact of our attack on the other cryptographic algorithms.

In elliptic curve-based cryptography, the scalar point multiplication algorithm is a core component for constructing cryptographic functions. As described in the previous section, the attack we presented targets the scalar point multiplication, thus it can be applied to the other EC-based cryptographic algorithms including both signature (*e.g.,* ECDSA) and encryption (*e.g.,* ECIES).

Many RSA implementations also make good use of a lookup table. The main operation performed during RSA decryption is the modular exponentiation, which is implemented to repeatedly access precomputed multipliers on the table according to the chunks of the exponent. Since the access pattern for multipliers is dependent on the exponent, it is also susceptible to our attack.

AES is yet another instance of lookup-based implementations. Each round operation involves several lookups to *T-tables*, which will result in input-dependent prefetching activities. In fact, such AES implementations are already known to be vulnerable to cache side-channel attacks [24, 29]. However, our attack against AES is a little more challenging compared to those cache-based attacks, since the prefetching is triggered only if the key bytes form a regular stride. Moreover, our attack requires the target implementation to have a loop structure (*i.e.,* not fully unrolled); otherwise, the stride prefetching would not be triggered.

6.4 Mitigation

Disabling prefetcher. The root cause of the attack is the secret-dependent activity patterns from the IP-based stride prefetcher. Possible mitigation may involve disabling this prefetcher on the processor core. The targeted Intel processors provide a method to control individual hardware prefetchers via a Model Specific Register (MSR). The stride prefetcher can be turned off by setting the leftmost bit (Bit #3) in the MSR with address of `0x1a4` [54]. In Linux, some user-level utilities are available such as `msr-tools` [44] to manipulate the register. Achieving protection by suppressing the prefetcher may incur a certain level of performance degradation.

Ensuring constant access pattern. One of the possible software-based mitigations is to eliminate any involved lookup tables in a given implementation. In our targeted function (*i.e.,* GF2M_SQUARE) in the OpenSSL library, the lookup to SQR_tb can be removed by replacing it with on-the-fly calculation. Such a countermeasure has already been applied in the patched version of OpenSSL library[2]. Since the lookup-based technique generally achieves the highest performance optimization, this method results in inevitable performance degradation. The OpenSSL benchmarking tool reports that the patch causes 4-8% performance drop.

Another possible mitigation is to make the memory access pattern of table lookup always constant regardless of the input. For instance, recent implementations of RSA modular exponentiation in OpenSSL adopt a scatter-gather technique [21]. It arranges elements on the lookup table so that all the elements are accessed via the constant pattern. This technique can be generalized to any kind of lookup-based implementations.

Preventing Flush+Reload attack. Our prefetching attack is constructed upon Flush+Reload technique. Hence, any suggested mitigation strategies against Flush+Reload attack will be effective as a countermeasure against our attack. One prerequisite of this attack is that memory in use has to be shared between attacker and victim. Copy-on-access mechanism [59] is a software-based solution that hinders memory pages from being shared across security domains (*e.g.,* processes or VMs). Specifically, it duplicates a shared copy of physical page whenever it is accessed by multiple security domains. Software diversification [10, 51] is another defense mechanism against Flush+Reload attack. This technique dynamically randomizes the executable binary of software so that memory sharing is inherently restricted.

7 CONCLUSION

In this paper, we presented a new microarchitectural attack that exploits hardware-based data prefetching to leak secret data. Our attack targets the IP-based stride prefetcher, which is one of the hardware prefetchers enclosed in recent Intel processors. We demonstrated the potential of the attack by recovering a private key in ECDH algorithm with the latest version of OpenSSL library. The proposed attack is not limited to this specific cryptographic algorithm; any implementation that utilizes a lookup table are subject to the attack, which exploits hardware prefetchers. Therefore, system developers and researchers should consider this attack as a new security threat when designing future systems. As a temporary countermeasure, we recommended several mitigating methods against the prefetching vulnerability on the processor. However, more effective and fundamental countermeasures need to be devised against the attack to minimize any possible performance degradation.

[2]https://github.com/openssl/openssl/commit/b336ce57f2d5cca803a920d2a9e622b588cead3c

RESPONSIBLE DISCLOSURE

We responsibly reported to OpenSSL as well as Intel about our findings of the hardware prefetching vulnerability. In response, OpenSSL addressed the issue and patched the GF2M_SQUARE function (in bn_gf2m.c) by eliminating the table lookup operation.

ACKNOWLEDGEMENT

We are grateful to the anonymous reviewers and our shepherd, Boris Köpf, for their valuable feedbacks. We also would like to thank Matt Caswell for suggesting the mitigation of prefetching vulnerability in OpenSSL library. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No.2017R1C1B5015045 and No. 2016R1A2A2A05005402) and by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW supervised by the IITP (Institute for Information & communications Technology Promotion)(2017-0-00096). This work was also supported in part by NSR grant (2018-057) and by an IITP grant funded by the MSIP (No. 2017-0-00184, Self-Learning Cyber Immune Technology Development).

REFERENCES

[1] Michael R Anderberg. 1973. *Cluster Analysis for Applications* (1st ed.). Elsevier.

[2] Diego F Aranha, Julio López, and Darrel Hankerson. 2010. Efficient software implementation of binary field arithmetic using vector instruction sets. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT 2010)*. Springer, 144–161.

[3] AWS. 2018. Instance types of AWS EC2. https://aws.amazon.com/intel/

[4] G Berk, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. A Faster and More Realistic Flush + Reload Attack on AES. In *International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2015)*. 111–126.

[5] Sarani Bhattacharya and Chester Rebeiro. 2016. A Formal Security Analysis of Even-Odd Sequential Prefetching in Profiled Cache-Timing Attacks. In *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP 2016)*. 1–8.

[6] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. 2012. Hardware prefetchers leak : A revisit of SVF for cache-timing attacks. In *IEEE/ACM 45th International Symposium on Microarchitecture Workshops (MICROW 2012)*. 17–23.

[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 2017)*. 1–12.

[8] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (1995), 609–623.

[9] Jean-Sebastien Coron. 1999. Resistance against differential power analysis for elliptic curve cryptosystems. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999)*. Springer, 292–302.

[10] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Proceedings of the 22th Network and Distributed System Security Symposium (NDSS 2015)*. 8–11.

[11] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. PRIME+ABORT: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *Proceedings of the 26th USENIX Security Symposium*. 51–67.

[12] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proceedings of the 22th USENIX Security Symposium*. 431–446.

[13] Goran Doychev and Boris Köpf. 2017. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 406–421.

[14] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope : A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018)*. 693–707.

[15] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, and Guru Venkataramani. 2018. Prefetch-guard: Leveraging hardware prefetchers to defend against cache timing channels. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST 2018)*. 1–4.

[16] John W C Fu, Janak H Patel, and Bob L Janssens. 1992. Stride directed prefetching in scalar processors. *Proceedings of the 25th annual ACM/IEEE International Symposium on Microarchitecture (MICRO 25)* (1992), 102–110.

[17] Adi Fuchs and Ruby B. Lee. 2015. Disruptive prefetching: Impact on Side-Channel Attacks and Cache Designs. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR 2015)*. 1–12. https://doi.org/10.1145/2757667.2757672

[18] Cesar Pereida García and Billy Bob Brumley. 2017. Constant-Time Callees with Variable-Time Callers. In *Proceedings of the 26th USENIX Security Symposium*. 83–98.

[19] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

[20] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*. 845–858.

[21] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. 2009. Fast and Constant-Time Implementation of Modular Exponentiation. In *Embedded Systems and Communications Security*.

[22] Daniel Gruss, Clementine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch SideChannel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. 368–379.

[23] Daniel Gruss, Clementine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2006)*. 279–299.

[24] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Proceedings of the 24th USENIX Security Symposium*. 897–912.

[25] Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES 2016)*. 368–388.

[26] Taylor Hornby. 2016. Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+RELOAD. In *BlackHat USA 2016*.

[27] Intel. 2018. *Intel 64 and IA-32 Architectures Optimization Reference Manual (April 2018)*.

[28] Intel. 2018. *Intel 64 and IA-32 Architectures Software Developer Manuals (March 2018)*.

[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A shared cache attack that works across cores and defies VM sandboxing - And its application to AES. In *Proceedings of 2015 IEEE Symposium on Security and Privacy*. 591–604.

[30] Teresa L Johnson, Matthew C Merten, and Wen-Mei W Hwu. 1997. Run-time spatial locality detection and optimization. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture (MICRO 30)*. IEEE Computer Society, 57–64.

[31] Norman P Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture (ISCA 1990)*. IEEE, 364–373.

[32] Marc Joye and Sung-Ming Yen. 2002. The Montgomery Powering Ladder. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*. 291–302.

[33] Thierry Kaufmann, Herve Pelletier, Serge Vaudenay, and Karine Villegas. 2016. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015. In *15th International Conference on Cryptology and Network Security (CANS 2016)*. 573–582.

[34] Neal Koblitz. 1987. Elliptic curve cryptosystems. *Math. Comp.* 48, 177 (1987), 203–209.

[35] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203* (2018).

[36] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *Proceedings of the 25th USENIX Security Symposium*. 549–564.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of 2015 IEEE Symposium on Security and Privacy*. 605–622.

[39] Julio Lopez and Ricardo Dahab. 1999. Fast Multiplication on Elliptic Curves over GF(2 to m) without Precomputation. In *CRYPTO 1999*. 316–327.

[40] James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical*

Statistics and Probability, Vol. 1. Oakland, CA, USA, 281–297.

[41] Giorgi Maisuradze and Christian Rossow. 2018. SPECULOSE: Analyzing the Security Implications of Speculative Execution in CPUs. *arXiv preprint arXiv:1801.04084* (2018).

[42] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES 2017)*. 69–90.

[43] Peter L Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* 48, 177 (1987), 243–264.

[44] Msrtool. 2018. The website of Msrtool project. https://www.coreboot.org/Msrtool

[45] National Institute of Standards and Technology. 2013. *FIPS PUB 186-4 Digital Signature Standard (DSS)*.

[46] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. 2000. Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications. In *International Workshop on Public Key Cryptography (PKC 2000)*, Hideki Imai and Yuliang Zheng (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–257.

[47] OpenSSL. 2018. OpenSSL, Cryptography and SSL/TLS Toolkit. http://www.openssl.org

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. In *Cryptographers Track at the RSA Conference (CT-RSA 2006)*. 1–20.

[49] Subbarao Palacharla and Richard E Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer Architecture (ISCA 1994)*. IEEE Computer Society Press, 24–33.

[50] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. 1639–1650.

[51] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of the 24th USENIX Security Symposium*. 431–446.

[52] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clementine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2017)*. 3–24.

[53] Alan Jay Smith. 1982. Cache memories. *ACM Computing Surveys (CSUR)* 14, 3 (1982), 473–530.

[54] Vish Viswanathan. 2014. Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors

[55] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *Proceedings of the 26th USENIX Security Symposium*. 235–252.

[56] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-channel Attack. *IACR Cryptology ePrint Archive, Report 2014/140* (2014).

[57] Yuval Yarom and Katrina Falkner. 2014. Flush + Reload : a High Resolution , Low Noise , L3 Cache Side-Channel Attack. In *Proceedings of the 23th USENIX Security Symposium*. 719–732.

[58] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 SIGSAC ACM conference on Computer and communications security (CCS 2014)*. 990–1003.

[59] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A Software Approach to Defeating Side Channels in Last-Level Caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. 871–882.