8-8-2017

Privacy Leakage via Write-Access Patterns to the Main Memory

Tara M. John
taramerin@gmail.com

Privacy Leakage via Write-Access Patterns to the Main Memory

Tara Merin John

B. Tech. Electronics and Communication Engineering, University of Kerala, India, 2009

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2017

APPROVAL PAGE

Master of Science

Privacy Leakage via Write-Access Patterns to the Main Memory

Presented by

Tara Merin John, B. Tech.,

Major Advisor _____
Marten van Dijk

Associate Advisor _____
Omer Khan

Associate Advisor _____
John Chandy

University of Connecticut

2017

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my primary advisor Prof. Marten van Dijk for his support and guidance throughout my graduate career.

I would also like to thank my co-advisors Prof. John Chany and Prof. Omer Khan for reviewing and providing valuable feedback to improve my work.

Last but not the least, I would like to thank all my colleagues and co-authors, Syed Kamran Haider and Hamza Omar, who have contributed towards my successful completion of Masters thesis work by providing their valuable time and support. Because of their efforts, we were able to achieve a prestigious *Best Poster Award* at IEEE Symposium on Hardware Oriented Security and Trust (*HOST*), 2017.

Contents

List of Figures

ABSTRACT

Data-dependent access patterns of an application to an untrusted storage system are notorious for leaking sensitive information about the user's data. Previous research has shown how an adversary capable of monitoring both read *and* write requests issued to the memory can correlate them with the application to learn its sensitive data. However, information leakage through *only* the write access patterns is less obvious and not well studied in the current literature. This work demonstrates an actual attack on power-side-channel resistant Montgomery's ladder based modular exponentiation algorithm commonly used in public key cryptography. The complete 512-bit secret exponent was inferred in $\sim$ **3.5** minutes by virtue of just the write access patterns of the algorithm to the main memory. In order to learn the victim algorithm's write access patterns under realistic settings, this work exploits a compromised DMA device to take frequent snapshots of the application's address space, and then run a simple differential analysis on these snapshots to find the write access sequence. The attack has been shown on an Intel Core(TM) i7-4790 3.60GHz processor based system. Lastly, there are further discussions about a possible attack on McEliece public-key cryptosystem that also exploits the write-access patterns to learn the secret key.

1 Introduction

Users' data privacy is becoming a major concern in computation outsourcing in the current cloud computing world. Numerous secure processor architectures (e.g., XOM [1, 2], TPM+TXT [3], Aegis [4], Intel-SGX [5] etc.) have been proposed for preserving data confidentiality and integrity during a remote secure computation. The user sends his encrypted data to a secure processor where it is decrypted and computed upon in a tamper-proof environment, and finally the encrypted results of the computation are sent back to the user.

While the secure processors provide sufficient levels of security against *direct* attacks, most of these architectures are still vulnerable to *side-channel* attacks. For instance, XOM and Aegis architectures are vulnerable to control flow leakage via address bus snooping [6, 7, 8]. Similarly, Intel-SGX, being a strong candidate in secure architectures, is vulnerable to side-channel attacks via a compromised OS[9].

Zhuang *et al.* [10] showed that although the data in the main memory of the system can be encrypted, the access patterns to the memory could still leak privacy. An adversary who is able to monitor both read *and* write accesses made by an application can relate this pattern to infer secret information of the application. For example, Islam *et al.* [11] demonstrated that by observing accesses to an encrypted email repository, an adversary can infer as much as 80% of the search queries. This, however, is a very strong adversarial model which, in most cases, requires direct physical access to the memory address bus. In cloud computing, for example, this requires the cloud service itself to be untrusted. The challenging requirements posed by the above mentioned strong adversarial model leads one to think that applications are vulnerable to privacy leakage via memory access patterns only if such a strong adversary exists,

i.e., one capable of monitoring both read *and* write accesses.

In this paper, we counter this notion by demonstrating privacy leakage under a significantly weaker adversarial model. In particular, we show that an adversary capable of monitoring *only* the write access patterns of an application can still learn a significant amount of its sensitive information. Hence, in the model of computation outsourcing to a secure processor discussed earlier, even if the cloud service itself is trusted, a remote adversary is still able to steal private information if the underlying hardware does not protect against leakage from write access patterns.

We present a real attack on the famous Montgomery's ladder technique [12] commonly used in public key cryptography for modular exponentiation. Exponentiation algorithms, in general, are vulnerable to various timing and power side-channel attacks [13, 14, 15]. Montgomery's ladder performs redundant computations as a countermeasure against power side-channel attacks (e.g., simple power analysis [16]). However, by monitoring the order of write accesses made by this algorithm, one can still infer the secret *exponent* bits.

In our weaker adversarial model, since we cannot directly monitor the memory address bus, we learn the pattern of write accesses by taking frequent memory snapshots. For this purpose, we exploit a compromised Direct Memory Access device (DMA[1]) attached to the victim computer system to read the application's address space in the system memory [17, 18, 19]. Clearly, any two memory snapshots only differ in the locations where the data has been modified in the latter snapshot. In other words, comparing the memory snapshots not only reveals the fact that write accesses (if any) have been made to the memory, but it also reveals the exact locations

[1]DMA grants full access of the main memory to certain peripheral buses, e.g. FireWire, Thunderbolt etc.

of the accesses which leads to a precise access pattern of memory writes.

Our experimental setup uses a PCI Express to USB 3.0 adapter attached to the victim system, alongside an open source application called PCILeech [20], as the compromised DMA device. We implement the Montgomery's ladder for exponentiation of a 128 byte message with a 64 byte (512 bits) secret exponent [21]. Through our attack methodology, we are able to infer all 512 secret bits of the exponent in just 3 minutes and 34 seconds on average.

Although our experimental setup utilizes a wired connection to a USB 3.0 port on the victim system for DMA, Stewin *et al.* demonstrated that DMA attacks can also be launched *remotely* by injecting malware to the dedicated hardware devices, such as graphic processors and network interface cards, attached to the host platform [22]. Therefore, our attack methodology allows even remote adversaries to exploit the coarse grained side-channel information obtained by memory snapshots to infer the secret data. Hence, this effort opens up new research avenues to explore efficient countermeasures to prevent privacy leakage under remote secure computation.

2 Background

2.1 Exponentiation Algorithms

Exponentiation algorithms have central importance in cryptography, and are considered to be the back-bone of nearly all the public-key cryptosystems. Although numerous exponentiation algorithms have been devised, algorithms for constrained devices are scarcely restricted to the square-and-multiply algorithms. RSA algorithm, used in e.g. Diffie-Hellman key agreement, is a commonly used exponentiation algorithm which performs computation of the form $y = g^k \mod n$, where the attacker's goal is to find the secret key k. The commonly used square-and-multiply implementation of this algorithm is shown in Algorithm 1. For a given input g and a secret key k, Algorithm 1 performs multiplication and squaring operations on two local variables R_0 and R_1 for each bit of k starting from the most significant bit down to the least significant bit.

Notice that the conditional statement on line 4 of Algorithm 1 executes based on the value of secret bit k_j. Such conditional branches result in two different power and timing spectra of the system for $k_j = 0$ and $k_j = 1$, hence leaking the secret key k over the timing/power side-channels. Similar attacks [21] have leaked 508 out of 512 bits of an RSA key by using branch prediction analysis (BPA). Thus, the attack-prone nature of RSA algorithm (Algorithm 1) poses a need for an alternate secure algorithm.

Algorithm 1 RSA - Left-to-Right Binary Algorithm

Inputs: g, $k = (k_{t-1}, \cdots, k_0)_2$ **Output:** $y = g^k$

Start:

1: $R_0 \leftarrow 1$; $R_1 \leftarrow g$
2: **for** $j = t - 1$ *downto* 0 **do**
3: $\quad R_0 \leftarrow (R_0)^2$
4: $\quad$ **if** $k_j = 1$ **then** $R_0 \leftarrow R_0 R_1$ **end if**
5: **end for**

return R_0

Algorithm 2 Montgomery Power Ladder Algorithm

Inputs: g, $k = (k_{t-1}, \cdots, k_0)_2$ **Output:** $y = g^k$

Start:

1: $R_0 \leftarrow 1$; $R_1 \leftarrow g$
2: **for** $j = t - 1$ *downto* 0 **do**
3: $\quad$ **if** $k_j = 0$ **then** $\quad R_1 \leftarrow R_0 R_1; \quad R_0 \leftarrow (R_0)^2$
4: $\quad$ **else** $\qquad\qquad R_0 \leftarrow R_0 R_1; \quad R_1 \leftarrow (R_1)^2$
5: $\quad$ **end if**
6: **end for**

return R_0

2.2 Montgomery's Power Ladder Algorithm

Montgomery Power Ladder [12] shown in Algorithm 2 performs exponentiation without leaking any information over power side-channel. Regardless of the value of bit k_j, it performs the same number of operations in the same order, hence producing the same power footprint for $k_j = 0$ and $k_j = 1$. Notice, however, that the specific order in which R_0 and R_1 are updated in time depends upon the value of k_j. E.g., for $k_j = 0$, R_1 is written first and then R_0 is updated; whereas for $k_j = 1$ the updates are done in the reverse order. This sequence of write access to R_0 and R_1 reveals to the adversary the exact bit values of k. In this paper, we exploit this vulnerability in a real implementation of Montgomery ladder to learn the secret key k.

3 The Proposed Attack

3.1 Adversarial Model

Consider a computer system that is continuously computing exponentiations of the form $y = g_i^k$ for the given inputs g_i using the same secret exponent k according to Algorithm 2. We call this system the *victim* system. All the data stored in the main memory of this system is encrypted. Let there be a compromised DMA device (e.g., a PCI-to-USB adapter) connected to the victim system through which an attacker system can read the whole main memory of the victim as shown in Figure 3.1. The attacker system, however, is limited in its ability to successively read the victim's memory by the data transfer rate of the underlying DMA interface. The adversary's goal is to find the key k by learning the application's write pattern through frequent snapshots of the victim system's memory. The victim system used in our attack comes with a *write-through* cache configuration enabled by default. As a result, any write operations performed by the application are immediately propagated through the memory hierarchy down to the untrusted DRAM. Furthermore, we assume that the victim application receives all the inputs g_i in a batch and continuously produces the corresponding cipher texts such that the physical memory region allocated to the application during successive encryptions remains the same. In other words, the application is not relocated to a different physical address space by the OS throughout the attack. Such use cases can be found in the applications that require computing signatures of large files.

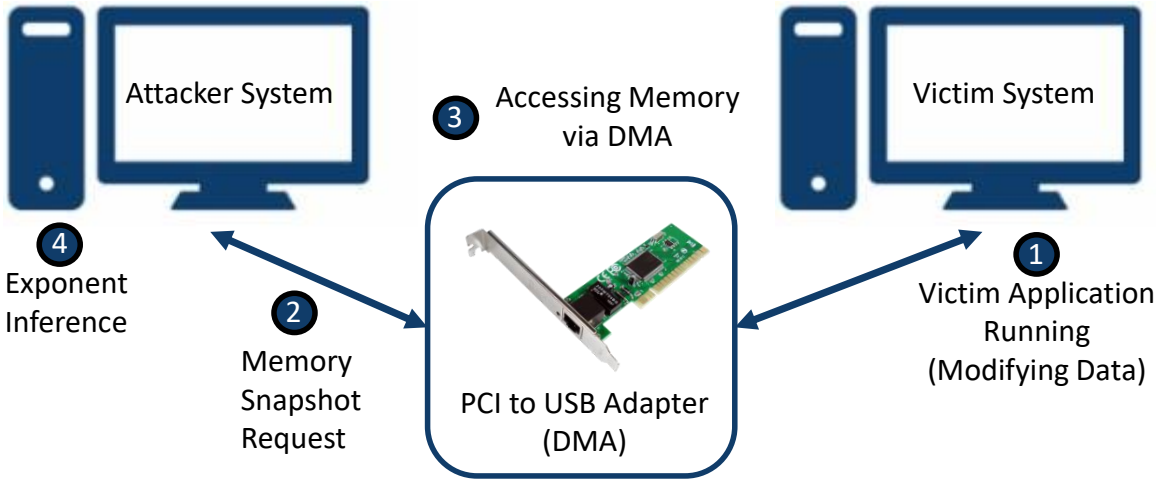

Figure 3.1: Our adversarial model: The attacker system takes snapshots of the victim's DRAM via the PCI adapter to infer the secret key.

3.2 Attack Outline

Given the above mentioned setting, we proceed with our attack methodology as follows: First, a full scan of the victim's memory is performed to identify the physical address space allocated to the victim's application. Since the adversary requires victim application's memory snapshots at a high frequency, it is infeasible for him to always read the full victim memory because of the data transfer rate being the frequency limiting factor. Once the address space is identified, the next step is to identify the two memory regions allocated to each of the local variables R_0 and R_1 (cf. Algorithm 2) within the victim application's address space. This allows any observed change in either of these two regions to be linked with an update to the variables R_0 and R_1 respectively. Finally, the updates in R_0 and R_1 memory regions are observed via frequent snapshots for a period of one complete encryption, and the order of these updates is linked back to Algorithm 2 to learn the key k. We explain these steps in detail in the following subsections.

Algorithm 3 Victim App's Address Space Identification

Inputs: M: Set of memory blocks to scan.
Output: S: Set of application's memory block(s).
Start:

```
1: S = ∅                                          ▷ Initially empty set.
2: for m ∈ M do                                   ▷ Scan each block.
3:     s1 = TakeSnapshot(m)
4:     s2 = TakeSnapshot(m)
5:     if CompareMatch(s1, s2) then
6:         S = S ∪ m
7:     end if
8: end for
9: return S
```

3.3 Step 1: Application's Address Space Identification

Since the application is supposed to be continuously updating its data (e.g., variables R_0, R_1), its address space can be identified by finding the memory regions which are continuously being updated. Algorithm 3 shows this process at an abstract level. The whole of the victim system's memory space is divided into M blocks, each of some reasonable size B (say a few megabytes). Two subsequent snapshots of each block $m \in M$ are compared with each other through CompareMatch procedure. It is a heuristic based process which searches for a specific pattern of updates between the two snapshots which potentially represents the application's footprint. For example, a sequence of two modified consecutive 64 byte cache lines followed by a few unmodified cache lines and then further two modified consecutive cache lines would potentially represent the two 128 byte regions for R_0 (first two cache lines) and R_1 (last two cache lines). Finally, a set S of all those memory blocks which show the specific update sequence searched by CompareMatch is returned. This algorithm is iteratively repeated until a reasonably small set of memory block(s) (e.g., one 4 kB page) is

Algorithm 4 Pseudo code for the second phase of attack

Input: S: Application's memory space. (from Algorithm 3); n: # of snapshots to cover one full encryption period.
Output: k: Application's secret key.
Start:

1: $V = (s_1, \cdots, s_n) \mid s_i = \text{TakeSnapshot}(S), 1 \leq i \leq n$
2: $Th = \text{ComputeThreshold}(V)$
3: $W = \emptyset$, $k = (0, \cdots, 0)$
4: $V = \text{RemoveUnchangedSnapshots}(V)$
5: **for** $i = 1$ **to** $|V| - 1$ **do**
6: $\quad R_{x_i} = \text{Correlate}(s_i, s_{i+1}, Th)$ ▷ $x_i \in \{0, 1\}$
7: $\quad W = W \cup R_{x_i}$
8: **end for**
9: $i = 1$, $j = 0$
10: **for** $(R_{x_i}, R_{x_{i+1}}) \in W$ **do**
11: $\quad$ **if** $R_{x_i} = R_0$ and $R_{x_{i+1}} = R_1$ **then** $k_j = 1$
12: $\quad$ **else if** $R_{x_i} = R_1$ and $R_{x_{i+1}} = R_0$ **then** $k_j = 0$
13: $\quad$ **end if**
14: $\quad i = i + 2; \quad j = j + 1$
15: **end for**
return k

identified which is expected to contain the victim application's address space.

3.4 Step 2: Distinguishing Local Variables R_0 and R_1

Once the application's memory space is found, we need to link two distinct regions within this address space to the variables R_0 and R_1 in order to determine the key bits from the order of their updates. For this purpose, a set V of n snapshots of the application's space is computed as shown in Algorithm 4. Notice that n is large enough to cover one full encryption period. The ComputeThreshold procedure computes a histogram of the updates performed inside the application's memory over all the snapshots of set V. Figure 3.2 shows one such histogram for a 4kB page of victim's

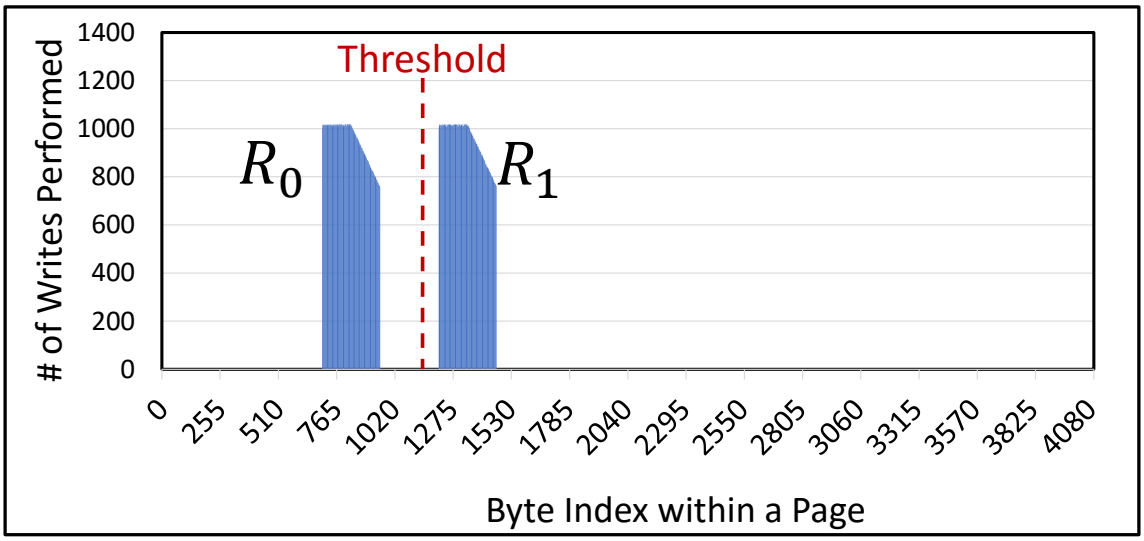

Figure 3.2: A histogram of # of writes to individual bytes in the victim's memory page. A clear distinction is shown between the regions corresponding to variables R_0 and R_1.

memory. It can be seen that almost all the updates are performed at two distinct regions spanning over only a few cache lines within the page. These two regions correspond to the variables R_0 and R_1 respectively[2]. The *inactive* region between R_0 and R_1 represents a threshold which is later used by CORRELATE procedure to determine whether a change in two successive memory snapshots corresponds to an update in R_0 or R_1 etc.

3.5 Step 3: Inferring the Secret Key

After computing the set of snapshots V and the threshold Th, we enter the final phase of inferring the secret key (starting from step 4 in Algorithm 4). Up to this point, the sequence V contains pairs of snapshots that represent changes in R_0 and

[2]We can tell whether R_0 or R_1 comes first in the memory layout from the declaration order of these variables in the actual implementation of the exponentiation algorithm (cf. line 1 in Algorithm 2).

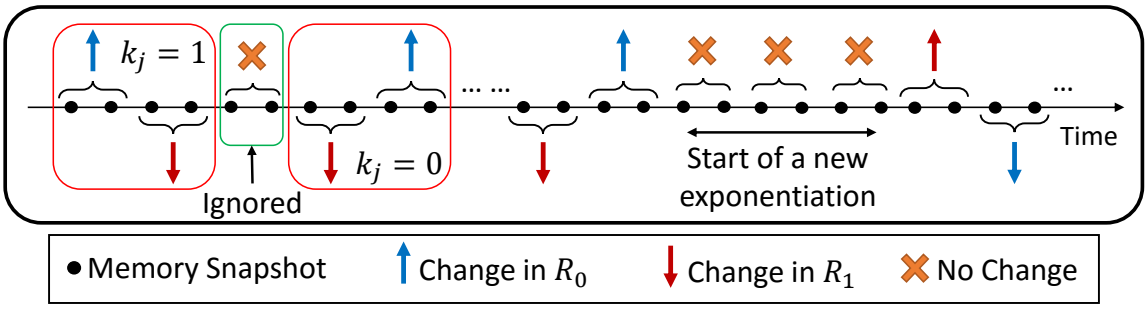

Figure 3.3: Inferring the secret key via observing the sequence of snapshots and the changes in variables R_0 and R_1. The pairs of snapshots which do not show any change are ignored.

R_1, and also the pairs which represent no change, as shown in Figure 3.3. The reason why some pairs do not show any change is because our snapshot frequency is higher than the rate at which the application updates its data. This allows us to learn the write access pattern at a fine granularity.

In order to learn the write access pattern, first the pairs of unchanged snapshots from the sequence V are removed by the procedure RemoveUnchangedSnapshots(V). The resulting sequence V only contains pairs which always represent a change, either in R_0 or R_1. Now, each pair of two successive snapshots is correlated to an update in either R_0 or R_1 by Correlate procedure using the threshold computed earlier.

As mentioned earlier, Montgomery ladder algorithm performs computations upon local variables, where the order of variable updates is based on the secret exponent bits (cf. Algorithm 2). Therefore, judging from the order of updates made in R_0 and R_1, each pair of updates $(R_{x_i}, R_{x_{i+1}}) \in W$ is linked back to the corresponding value of the secret key bit k_j as shown in figure 3.3. As the set W contains the history of all the updates to R_0 and R_1 for a complete encryption, therefore all the key bits can be inferred through the above mentioned process.

4 Attack Demonstration

4.1 Experimental Setup

Our experiment setup uses two computer systems, one being the attacker and the other being the victim. In our experiments, the victim system is *DELL XPS 8700*, comprising of *Intel Core(TM) i7-4790 3.60GHz* processor that uses *Ubuntu 14.04.3 LTS* operating system with a *Linux kernel 3.19.0-43-generic*, and has 16GB of main memory. The attacker machine is a *64-bit Windows 10* based system having 8GB of main memory. A PCI adapter module, called *USB Evaluation Board* [23], is connected to the victim via the PCI-Express slot and acts as a compromised DMA device (cf. Figure 3.1). This DMA device, together with PCILeech software [20], allows the attacker to monitor victim's memory and/or take its snapshots. To implement our attack, the PCILeech software has been extended to first find the application's address space in the victim's memory (cf. Algorithm 3), and then attack the identified address space to infer the secret key (cf. Algorithm 4)). The above mentioned attacking algorithms run *while* the victim application is executing.

We have written our own C++ implementation of the Montgomery ladder based exponentiation algorithm[3] for large input sizes (128 Bytes or more) that runs on the victim system. The victim system has a BIOS version *A11* which supports *write-through* enabled L1 and L2 caches while disabling the L3 cache by default. Besides caches, any data modifications in the *register file* should also be propagated to the DRAM. The register file (usually of size 64-128 Bytes) is used by the processor to temporarily hold the operands and results during computations. Since our imple-

[3]Available at : https://github.com/meriniamjo/RSA-Montgomery-Ladder-Implementation

mentation uses multiple temporary variables and function calls for proper execution of the algorithm, the large "active" working set of the application cannot fit into the register file and results in *register spills*. Hence, any updates made by the application are immediately propagated – through register file and caches – down to the main memory as each multiplication/squaring write operation in performed. Section 4.2 explains the step by step details about how the attack is launched.

4.2 Experimental Results

In order to take memory snapshots via PCI module and the PCILeech software, the attacker first needs to load a kernel module into the victim system via the PCI module itself. Notice that the attacker does not require any extra privileges to do so. We use the following command via PCILeech software to load the kernel into victim's DRAM. When the kernel is loaded, an address is spitted out by the software, which shows where the module resides in the victim's memory. Loading the kernel into memory is a rapid process and takes only a few milliseconds to complete the process.

```
D:\>pcileech kmdload -kmd LINUX_X64
KMD: Code inserted into the kernel
KMD: Execution received - continuing ...
KMD: Successfully loaded at 0x1b54a000
D:\>_
```

In the meantime, the Montgomery's ladder exponentiation algorithm is run on the victim machine using a 128 byte (1024 bits) message along with a secret key of 64 bytes (512 bits).

```
[user@victim]$ ./montgomery_exponentiation
```

With the application running and the kernel module loaded into victim's memory, we proceed to find the potential regions in the DRAM which are being accessed frequently by taking multiple snapshots. To retrieve these snapshots, we issue the *pagefind* command shown below which uses the loaded kernel module's address to access the victim's full memory.

We integrated the *pagefind* command into the PCILeech software to iteratively find regions getting modified persistently. *pagefind* narrows down the selected regions to a single page by constantly monitoring and comparing the changes being made, and returns the address of the page where application's array data structures are defined. This step corresponds to *Application's Address Space Identification* phase of the attack (cf. Algorithm 3) and is the most time consuming phase. To read the whole memory, comparing their respective snapshots and narrowing down to a single page of 4KB from 16GB search space takes ∼3 minutes and 30 seconds.

```
D:\>pcileech pagefind -kmd 0x1b542000
Matching Pattern ...
Page Finding:  Successful.
Total_Time = 210199 Milliseconds
Victim Page Address :  0xd271c000
D:\>_
```

As shown above, from the first phase we retrieve the address of the page where application's data structures are stored. Proceeding towards our second and third step namely *Distinguishing Local Variables* and *Inferring the Secret Key* (cf. Section 3.4, 3.5), we use another integrated command *pageattack*. It first takes a predefined

number of snapshots of the application page provided by the first step, and distinguishes the message (R_1) and algorithm result (R_0) from the rest of the stale data, residing on the memory page. It then uses the order of changes in R_0 and R_1 to infer the secret key.

```
D:\>pcileech pageattack -min 0xd271c000
Attack Successful.
Total_Time = 3596 Milliseconds
Inferred Key is:
1a 4b 28 41 e6 27 d4 7d
72 c3 40 79 be 1f 6c 35
ca 3b 58 b1 96 17 04 ed
22 b3 70 e9 6e 0f 9c a5
7a 2b 88 21 46 07 34 5d
d2 a3 a0 59 1e ff cc 15
2a 1b b8 91 f6 f7 64 cd
82 93 d0 c9 ce ef fc 85
D:\>_
```

This final step takes ∼3.6 seconds to complete and returns the complete 512 bit secret key learned from only the write access patterns. Combing the times associated with all the attack phases, the total attack time comes out to be ∼3 minutes, 34 seconds.

5 Leakage under Caching Effects

In view of our proposed attack on Montgomery ladder based exponentiation algorithm, the updates to the application data should always be available in the DRAM of the victim system before an attacker issues a memory snapshot request. This is only possible if the victim system has *write-through* enabled cache hierarchy or the caches are disabled altogether. Whereas, on the other hand, modern processors typically consist of large on-chip *write-back* caches where the updates to application's data are only be visible in DRAM once the data is evicted from the last level cache (LLC). Thus in the attack proposed in Section 3, the caching effects are not catered for, which introduce 'noise' to the precise write-access sequence inferred earlier, hence making the attacker's job difficult. A possible workaround to deal with such caching effects is to collect several 'noisy' sequences of memory snapshots and then run correlation analysis on them to learn the precise write-access pattern. Furthermore, if the adversary is also a user of the same computer system, it can flush the system caches frequently to reduce the noise in write-access sequence even further.

Another (more efficient) attack scenario under write-back caches would be when the application has a strided memory access pattern that causes contention over the cache sets, and hence forces its own data to be evicted to make room for the new data in the cache. In the following subsection, we discuss how such a strided memory access pattern can lead to evictions to the DRAM which could potentially leak private information.

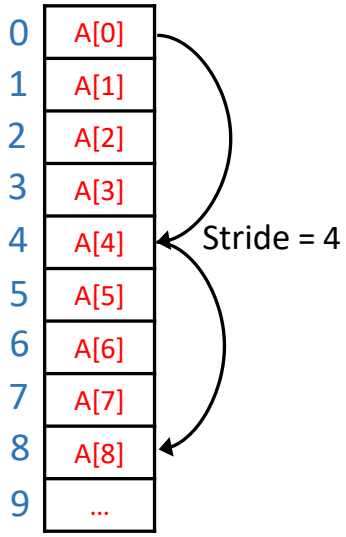

Figure 5.1: A strided memory access pattern with a stride of 4.

5.1 Memory Striding and Cache Set Contention

A strided memory access pattern is the one where each request to the memory is for the same number of bytes, and the access pointer is incremented by the same amount between each request. An array accessed with a stride of exactly the same size as the size of each of its elements results in accessing contiguous locations in the memory. Such access patterns are said to have a stride value of 1. Figure 5.1 shows a *non-unit* striding access pattern in which the elements $0, 4, 8, 12, \cdots$ of an array A are accessed. This access pattern has a stride value of 4.

Consider a simple system which has a 2-way set-associative *write-back* cache with a total capacity of 8 cache lines, as shown in Figure 5.2. The strided access pattern from Figure 5.1 accesses every $4i_{th}$ element of the array A, where $i = 0, 1, 2, \cdots$. Assuming that each element of A is of size equal to the cache line size, for a simple *modulus based* cache hash function, the elements $A[0], A[4], A[8], \cdots$ are mapped to

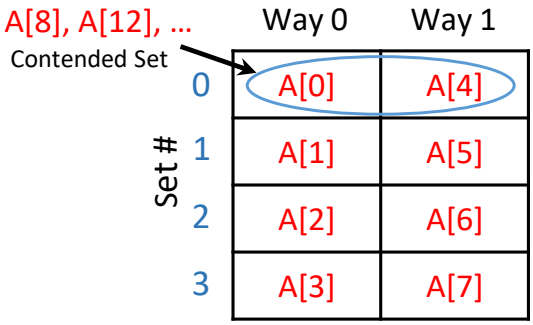

Figure 5.2: A 2-way set-associative cache with contention on a single set.

the same set causing contention over *set* 0. Since the cache associativity is only 2, this access sequence causes evictions from the cache when both ways of set 0 contain valid cache lines. Similarly, elements $A[1], A[5], A[9], \cdots$ map to *set* 1, and this access sequence will cause evictions from set 1, and so on. In other words, such write-access sequences are still propagated almost immediately to the next level in the memory hierarchy (e.g., DRAM) even under write-back caches, which could potentially leak information. This is an artifact of the cache implementation combined with the striding access pattern of the application.

It must be noted that, not all evictions result in updates to the main memory. Typically, only *dirty* cache lines, caused by data writes, evicted from the cache are propagated to the main memory. *Clean* evictions from the cache are simply discarded resulting in no change in the main memory since it already contains a clean copy of the data.

Assume that an application generates two distinguishable striding write-access patterns that result in contention at two different cache sets, leading to evictions from the cache. Consequently, the resulting write-access access sequence will be revealed to an adversary who is capable of monitoring changes in the main memory,

potentially resulting in privacy leakage.

5.2 Striding Application: Gaussian Elimination

In Section 5.1 we discussed, using a toy example, how a strided access pattern can lead to information leakage. Now we present a realistic example which has such a striding access pattern, and later in Section 5.3 we show how such a pattern can be exploited to learn private information. We consider the application of *Gaussian Elimination* of large binary matrices carrying substantial amount of information. Clearly, these large matrices cannot fit into the caches, therefore there will be cache evictions as a result of Gaussian elimination operations.

Gaussian elimination a.k.a. row reduction is a method for solving system of linear equations by the use of matrices in the form $Ax = B$. Row reduction is done by doing a series of elementary row operations which modify the matrix until it forms an upper triangular matrix, i.e., elements underneath the main diagonal are zeros. Different types of elementary row operations include swapping two rows, multiplying a row by a non-zero number and adding a multiple of one row to another. The upper triangular matrix formed out of these operations will be in row echelon form. When the leading coefficient (pivot) in each row is 1, and every column containing the leading coefficient has zeros elsewhere, the matrix is said to be in reduced row echelon form. The Gaussian elimination algorithm consists of two processes, one being *forward elimination* that converts the matrix to row echelon form and the other is *backward substitution* that calculates values of the unknowns. These processes result in solving the linear equation.

Gauss-Jordan elimination uses a similar approach for finding the inverse of a

matrix. For a $n \times n$ square matrix S, elementary row operations can be applied to reduce the matrix into reduced echelon form, and furthermore, for computing the matrix inverse if it exits. Initially, the $n \times n$ identity matrix I is augmented to the right of S, forming a $n \times 2n$ block matrix $[S|I]$. Now, upon applying the row operations, the left block can be reduced to the identity matrix I if S is invertible. This gives S^{-1} which is the right block of the final matrix. In a nutshell, we continue performing row operations until $[S|I]$ becomes $[I|S^{-1}]$.

Consider that the matrix under elimination is stored in a *column-contiguous* manner in the computer system's main memory. In other words, each column occupies a contiguous chunk of memory equal to the column size, after which the next column resides, and so on. When consecutive elements of a row of this matrix are accessed during a row operation, the corresponding memory access pattern results in a striding sequence, where the stride length is equal to the column size. If the stride length is such that it creates contention on particular cache sets corresponding to particular rows, this would reveal the modified row, which in turn could potentially leak the binary matrix itself (cf. Section 5.3).

5.3 Attacking McEliece Public-Key Cryptosystem

McEliece public key cryptosystem [24], an asymmetric encryption algorithm, uses an error correcting code for a description of the private key. This encryption uses a fast and efficient decoding algorithm, namely a Goppa code and hides the structure of the code by transformation of the generator matrix. This transformation yields the public key and the structure of the Goppa code together with the transformation parameters, which further provides the trapdoor information. For a linear code C, generator

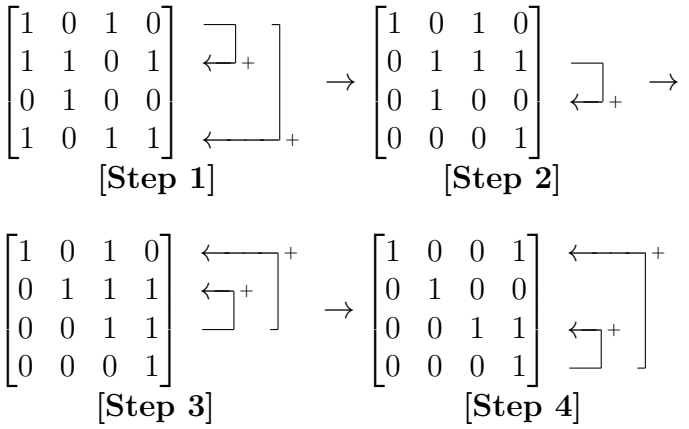

Figure 5.3: The Gaussian Elimination process on a 4×4 binary matrix.

matrix G, random invertible matrix S and random permutation matrix P, the matrix $G^* = SGP$ is made public while P, G, and S form the private key. A message m is encrypted along with a random error vector using the equation $c = mG^* + e$, where c refers to the ciphertext. In the decryption process, we compute $c^* = cP^{-1}$, decode c^* to m^* by the decoding algorithm, and lastly compute $m = m^*S^{-1}$. Notice, that S is a private binary matrix whose inverse is used to recover the message m. Any system carrying out this encryption/decryption process could either store the matrix inverse (for better performance) or calculate the inverse during the run time. However, the latter could lead to the leakage of the binary matrix via write-access patterns during the inverse computation. In this section we will demonstrate how performing Gauss-Jordan elimination [25] on the binary secret matrix S could lead to its complete exposure as a consequence of cache striding and cache set contention as shown in section 5.1.

For the ease of demonstration we consider a 4×4 binary matrix. The elements stored in the main memory are column contiguous. We assume that each element

of the matrix is *cache line aligned* for performance reasons. In other words, each element is stored in a unique cache line in order to avoid false sharing within a cache line. Considering a system with a 2-way 4-set associative cache, each row of the matrix is mapped to one cache set due to the cache structure (cf. Figure 5.2). The elimination process to obtain the inverse of the binary matrix $S_{4\times4}$ is shown step by step in Figure 5.3. After these row operations, we obtain an identity matrix $I_{4\times4}$.

In each of the above 4 steps, the corresponding pivot row is added to another row or rows. For example, in **Step [1]** row 1 is added to row 2 and row 4. Similarly, row 2 is added to row 3 in **Step [2]**. As a row operation is performed, the elements of the target row are modified and result in cache line evictions, since accessing a whole row causes contention over the corresponding set it is mapped to, and a cache set can only store 2 elements of a row at a time. For instance, in **Step [1]**, row 2 and row 4 are modified causing contention and evictions from set 1 and 3 respectively. Consequently, an adversary can learn the identifier of the row being updated during each row operation by monitoring the address space in which any updates take place, and then linking it back to the row number. Now, by definition of the elimination algorithm, all column elements corresponding to rows that undergo addition operations can be inferred as 1s, and the remaining ones as 0s. Hence, in each of **Step [1]**, **Step [2]**, **Step [3]** and **Step [4]**, we infer the corresponding pivot column to be $C_1 = \{1, 1, 0, 1\}$, $C_2 = \{0, 1, 1, 0\}$, $C_3 = \{1, 1, 1, 0\}$ and $C_4 = \{1, 0, 1, 1\}$ respectively.

Notice that C_2, C_3, and C_4 obtained in the above steps show the respective intermediate forms of the corresponding columns of S during the elimination process. These values, however, can be used to recover the original column values of matrix S through *back substitution* process, as shown in Figure 5.4. In this process, each

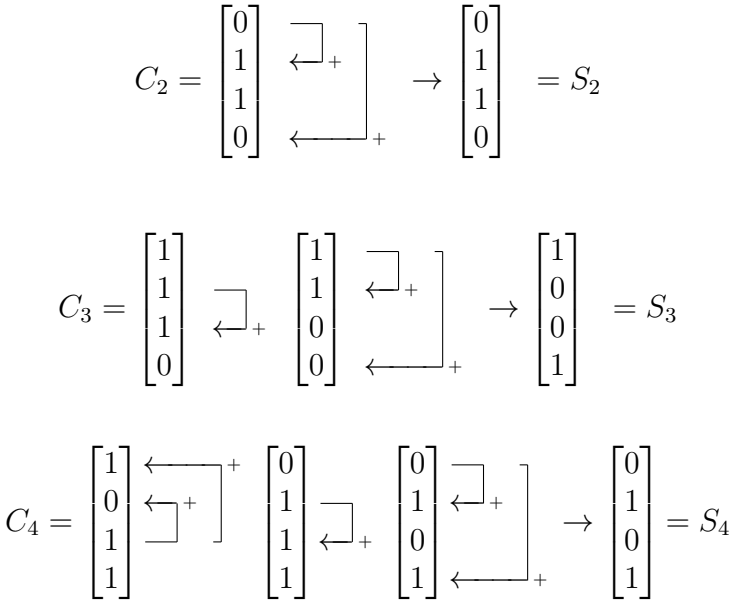

Figure 5.4: Back Substitution process to recover secret binary matrix S.

column C_i undergoes the row operations performed (and inferred) in each of the steps **Step [i-1]**, **Step [i-2]**, $\cdots$, **Step [1]**, precisely in this order. For example, C_2 undergoes addition of row 1 to rows 2 and 4, while C_3 performs addition of row 2 to row 3 along with the addition of row 1 to rows 2 and 4. Upon completion of the back substitution process, the complete secret matrix S is recovered by the adversary.

6 Discussion

6.1 Potential Threats in Database Applications

Database applications are ubiquitously being used to facilitate simultaneous updates and queries from multiple users. Such databases and cloud servers could store high volumes of data regarding an organization's operations. Examples include databases that record contact and credit information etc. about employees in an organization. In such databases, the *key* or the *record locater* of a piece of information, for instance, can be the social security number of the employee. Leaking the SSN of an employee from above mentioned databases can be detrimental [26]. In the following, we discuss how write-access patterns could potentially leak private information stored in two commonly used data structures.

Linked List

Linked list data structure is a collection of a group of data elements, called nodes, which together represent a sequence. Consider the users' encrypted private information stored in a singly linked list based on the increasing order of their social security numbers (SSN). The linked list is stored in the victim system's disk. The adversary's goal is to collect as many valid SSNs (which represent identities of real humans) from this linked list as possible.

Assume that two attackers are working in tandem and both of them are users of the linked list database mentioned above. Since for each search/update operation, the linked list is traversed started from the first node up to the node being searched/updated, all these nodes will be loaded from the disk into the DRAM in the specific

ascending order. Meanwhile, since the adversaries can monitor the DRAM snapshots throughout this process, they learn the exact order of each node in the linked list. Now, the adversaries can insert/update their own information in the database while monitoring the resulting memory write pattern. This allows them to link their own SSNs to two particular nodes in the linked list, separated by potentially a small number of other nodes. Crucially, this gives the adversaries a potentially small range of valid SSNs bounded by the two adversarial SSNs. Next time, whenever a user's information is updated which results in modifying a node between two adversarial nodes, the adversaries can brute force the small range of SSNs to find the valid SSN of the particular user.

Binary Search Tree (BST)

Binary search tree (BST) data structure stores data in memory while allowing fast lookup, addition and removal of the stored data. To perform a lookup/update operation, BST looks for a key by traversing the tree from root to leaf while choosing left or right child at each level based on the key to be searched.

Consider a similar scenario as in Section 6.1 where the data is stored in a BST instead, and assume that the attackers know the initial layout of the BST in the victim system's memory. Since traversing the tree involves only read accesses, this won't leak any information in our model. However, in case of write accesses (inserting or updating a node), the position where the node is inserted or updated will be leaked. Following a similar strategy as Section 6.1, the adversary can deduce a range of SSNs by inserting his own nodes in the BST. A larger group of individuals acting as adversary can deduce further smaller ranges of SSNs resulting in easier attack.

Notice that during this type of attack, the subtree corresponding to the range of SSNs found by the adversary should not be evicted from DRAM. If, for some reason, the pages corresponding to the vulnerable subtree are swapped out to disk by the OS, the layout of the subtree in the DRAM could be different once it is loaded in DRAM next time. In this case, the adversary will need to relearn the BST layout in DRAM by monitoring several updates over a long period of time.

6.2 Future Work: Countermeasures for Our Attack

One approach to prevent privacy leakage via write-access pattern leveraging DMA based attacks, as demonstrated in this paper, could be to block certain DMA accesses through modifications in the DRAM controller. However, this approach poses complexity in terms of how to determine which accesses to allow and which ones to block. Furthermore, it requires this 'extended' DRAM controller to be included in the trusted computing base (TCB) of the system which is undesirable.

Another strong candidate is Oblivious RAM which is a well known technique to prevent privacy leakage via memory access patterns. Although, current so called *fully functional* ORAMs, which obfuscate both read and write patterns, offer a possible countermeasure (at a cost of performance penalty) against the attack we demonstrated. However, the extra protection (read pattern obfuscation) offered by these approaches is an overkill for current attack scenario and incurs redundant performance penalties.

A better alternative could be a *write-only* ORAM [27, 28, 29] which only obfuscates write-access patterns and not the reads. This technique offers far better performance than a fully functional ORAM under such weaker adversarial models.

It has been shown that *write-only* ORAM has an optimal asymptotic communication overhead of $O(1)$ as compared to the fully functional ORAM schemes, which are asymptotically $\Omega(\log n)$ [27].

7 Conclusion

Privacy leakage via purely write-access patterns is less obvious and not extensively studied in the current literature. We demonstrate a real attack on Montgomery's ladder based modular exponentiation algorithm and infer the secret exponent by just learning the write access patterns of the algorithm to the main memory. We adapt the traditional DMA based exploits to learn the application's write access pattern in a reasonable time. Our attack takes just 3 minutes and 34 seconds to learn 512 secret bits from a typical Linux based victim system. A possible attack on McEliece public-key cryptosystem has also been presented. We discuss some possible countermeasures to prevent such attacks. Further research towards developing efficient countermeasures is left as future work.

Bibliography

[1] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz, "Specifying and verifying hardware for tamperresistant software," in *IEEE S & P*, 2003.

[2] D. Lie, C. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *SOSP*, 2003.

[3] D. Grawrock, *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[4] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, " AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *ICS*. ACM, June 2003.

[5] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *HASP@ ISCA*, 2013, p. 10.

[6] G. E. Suh, D. Clarke, B. Gassend, M. V. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *MICRO*, 2003.

[7] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *MICRO*, 2003.

[8] B. Gassend, G. E. Suh, D. Clarke, M. V. Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *HPCA'03*.

[9] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE S&P'15*.

[10] X. Zhuang, T. Zhang, and S. Pande, "Hide: an infrastructure for efficiently protecting information leakage on the address bus," in *ACM SIGPLAN Notices*. ACM, 2004.

[11] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation." in *NDSS*, vol. 20, 2012, p. 12.

[12] M. Joye and S. M. Yen, "The montgomery powering ladder," in *CHES'02*.

[13] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *CRYPTO*. Springer, 1996.

[14] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, 2005.

[15] L. Goubin, "A refined power-analysis attack on elliptic curve cryptosystems," in *PKC Workshop*. Springer, 2003.

[16] S.-M. Yen, L.-C. Ko, S. Moon, and J. Ha, "Relative doubling attack against montgomery ladder," in *ICISC*. Springer, 2005.

[17] D. Aumaitre and C. Devine, "Subverting windows 7 x64 kernel with dma attacks," *HITBSecConf Amsterdam*, 2010.

[18] D. Maynor, "Dma: Skeleton key of computing && selected soap box rants," *CanSecWest: http://cansecwest. com/core05/DMA. ppt*, 2005.

[19] B. Böck and S. B. Austria, "Firewire-based physical security attacks on windows 7, efs and bitlocker," *Secure Business Austria Lab'09*.

[20] U. Frisk, "Pcileech: Direct memory access attack software." [Online]. Available: https://github.com/ufrisk/pcileech

[21] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," ser. ASIACCS '07.

[22] P. Stewin and I. Bystrov, "Understanding dma malware," in *DIMVA*. Springer, 2012.

[23] I. BPlus Technology, "Usb3380 evaluation board." [Online]. Available: http://www.bplus.com.tw/Adapter/USB3380EVB.html

[24] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[25] A. Bogdanov, M. C. Mertens, C. Paar, J. Pelzl, and A. Rupp, "Smith-a parallel hardware architecture for fast gaussian elimination over gf (2)," in *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS 2006), Conference Records*, 2006.

[26] S. Musil, "Sony hack leaked social security numbers and celebrity data." [Online]. Available: https://www.cnet.com/news/sony-hack-said-to-leak-47000-social-security-numbers-celebrity-data/

[27] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Toward robust hidden volumes using write-only oblivious ram," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 203–214.

[28] L. Li and A. Datta, "Write-only oblivious ram-based privacy-preserved access of outsourced data," *International Journal of Information Security*, pp. 1–20, 2013.

[29] S. K. Haider and M. van Dijk, "Flat oram: A simplified write-only oblivious ram construction for secure processor architectures," *arXiv preprint arXiv:1611.01571*, 2016.