applied sciences

MDPI

Article

Fast and Secure Implementation of Modular Exponentiation for Mitigating Fine-Grained Cache Attacks

Youngjoo Shin

School of Computer and Information Engineering, Kwangwoon University, 20 Kwangwoon-ro, Nowon-gu, Seoul 01897, Korea; yjshin@kw.ac.kr; Tel.: +82-2-940-5130

Received: 5 July 2018; Accepted: 3 August 2018; Published: 5 August 2018

Abstract: Constant-time technique is of crucial importance to prevent secrets of cryptographic algorithms from leakage by cache attacks. In this paper, we propose Permute-Scatter-Gather, a novel constant-time method for the modular exponentiation that is used in the RSA cryptosystem. On the basis of the scatter-gather design, our method utilizes pseudo-random permutation to obfuscate memory access patterns. Based on this strategy, the resistance against fine-grained cache attacks is ensured, i.e., providing the higher level of security than the existing scatter-gather implementations. Evaluation shows that our method outperforms the OpenSSL library at most 11% in the mainstream Intel processors.

Keywords: cache attack; cache side-channel attack; constant-time cryptographic algorithm; rsa cryptosystem; scatter-gather implementation; modular exponentiation

1. Introduction

Cache attacks, such as Prime+Probe [1–4] and Flush+Reload [5–10], exploit the usage of CPU cache as a side channel to infer secret information of victim applications. Due to its high resolution, the cache attack is very effective in attacking cryptographic algorithms [11–15]. By monitoring secret-dependent patterns in memory access or control flow, an adversary can successfully extract private keys in an implementation of the cryptographic algorithms. Thus, it is necessary to consider constant-time programming when implementing cryptographic software secure against cache attacks. The constant-time programming is an implementation technique that ensures the cryptographic algorithm has constant patterns during the execution irrespective of an input (i.e., secret) in its implementation.

Scatter-gather [16] is a constant-time programming technique for the RSA algorithm [17], which is used in OpenSSL library [18]. The RSA encryption/decryption (or sign/verify) are basically performed as modular exponentiation, in which the exponent is a private key (or a singing key). For computational efficiency, several multipliers are pre-computed, stored as a table in the memory and accessed later during the exponentiation. In a naive lookup-based implementation, multipliers are located in separate memory lines, so accessing them would cause observable unique access patterns, which is susceptible to cache attacks. Scatter-gather technique revises the arrangement of multipliers on the table so that any multipliers are accessed with the constant pattern.

The current implementation of the scatter-gather technique has the underlying assumption that cache adversaries only observe the access pattern at the granularity of cache line (i.e., 64 bytes) [19,20]. However, such assumption was broken as more fine-grained cache attack has been recently discovered. This cache attack, dubbed *Cache-bleed* [21], exploits the cache-bank conflict between hyper-threads to observe the secret-dependent access pattern at the bank level during the gathering phase.

In this paper, we propose Permute-Scatter-Gather, a novel constant-time method for the RSA modular exponentiation, which is resistant against fine-grained cache attacks. Based on the scatter-gather design, our technique employs a pseudo-random permutation for locating multipliers in a scattered memory layout. Such permutation actually obfuscates the memory access pattern, thus prevents any adversaries even mounting fine-grained cache attacks from inferring the secret from the observations. Furthermore, our novel technique for *constant-time permutation* allows the permutation itself to have the constant-time property, making more secure against cache attacks.

Our evaluation shows that the proposed method outperforms the existing countermeasure, implemented in the recent version of OpenSSL, at most 11% in the mainstream processors. It is also shown that the Permute-Scatter-Gather can be easily adopted with the OpenSSL without significant effort, increasing the practicality of the proposed method.

The rest of this paper is organized as follows. Background is presented in Section 2. Details on the Permute-Scatter-Gather, and their evaluations are given in Section 3, and Section 4, respectively. Finally, we conclude the paper in Section 5.

2. Background

2.1. Scatter-Gather Implementation

The main operation of RSA decryption (or sign) is the modular exponentiation; calculate $b^e \bmod n$ for a secret exponent e. OpenSSL library performs the modular exponentiation by a fixed-window exponentiation algorithm [22] (See Algorithm 1). In a pre-computation phase, the algorithm computes a set of multipliers $m_i = m^j b \bmod n$ for $0 \leq j < 2^w$, where w is a window size. In an exponentiation phase, it scans each fraction of e of size w from $e_{\lceil k/w \rceil}$ to e_0. For each digit e_i, it multiplies r, the intermediate result from squaring, by the pre-computed multiplier m_{e_i}. In the OpenSSL library, the window size is set to $w = 5$, so there are 32 multipliers in total.

Algorithm 1 Fixed-window exponentiation

Require: k-bit exponent $e = \sum_{i=0}^{\lceil k/w \rceil} e_i \cdot 2^{wi}$, window size w, base b, modulus n
Ensure: $b^e \bmod n$

```
1:  procedure EXPONENTIATION(w, b, n, e)
2:      // Pre-computation phase
3:      m_0 ← 1
4:      for i ← 1 to 2^w − 1 do
5:          m_i ← m_{i−1} · b mod n
6:      end for
7:
8:      // Exponentiation phase
9:      r ← 1
10:     for i ← ⌈k/w⌉ − 1 to 0 do
11:         for j ← 1 to w do
12:             r ← r^2 mod n
13:         end for
14:         r ← r · m_{e_i} mod n
15:     end for
16:     return r
17: end procedure
```

Scatter-gather implementation is a constant-time programming technique to avoid secret-dependent access at the cache line granularity [23]. Instead of storing multipliers consecutively in memory, it scatters

each multiplier across multiple cache lines (Figure 1). When using the multiplier (i.e., in gathering phase), the fragments of the required multiplier are gathered to a buffer for the multiplication.

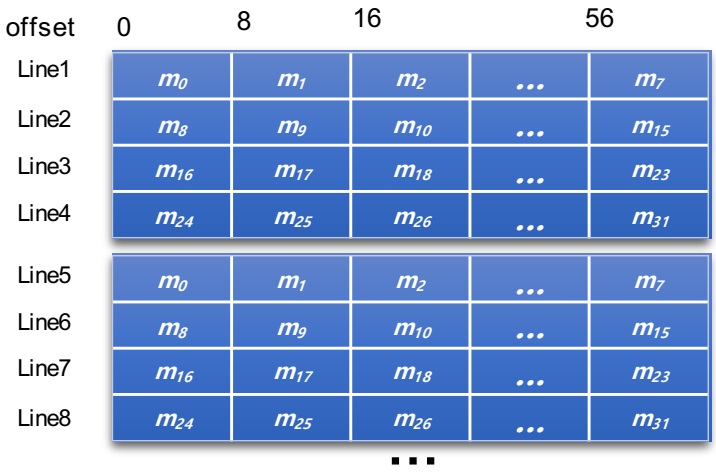

Figure 1. Memory layout of the multiplier table in OpenSSL.

2.2. Fine-Grained Cache Attack and Its Countermeasure

2.2.1. Fine-Grained Cache Attack

In Intel processors, a cache line is divided into multiple cache banks, each of which has part of the line specified by the line offset. In such cache design, concurrent requests to the same line can be served in parallel if the requested offsets are on the different banks. However, requests to the same bank would cause a cache line conflict, resulting in observable execution delay [24,25]. Such conflict at a cache line introduces fine-grained cache attacks such as Cache-bleed [21]. This kind of attacks exploits a bank level timing channel introduced by the cache line conflict. The granularity of the channel allows distinguishing between memory accesses within the same cache line.

With this attack, an adversary can infer which multipliers are accessed during the gathering phase in the exponentiation. It was shown that the scatter-gather implementation of OpenSSL library of the version 1.0.2f is vulnerable to the fine-grained cache attack, allowing the full recovery of RSA private keys [21].

2.2.2. Constant-Time Gather Procedure

The root cause of the OpenSSL's vulnerability to fine-grained cache attacks comes from that with the bank-level granularity, it has secret-dependent memory access in gathering phase. To mitigate the attack, the vulnerable version of the OpenSSL library (i.e., the version 1.0.2f) has been patched in the later version 1.0.2g so that all secret-dependent accesses are eliminated. More specifically, in the modified gathering process, all the multipliers laid on a single memory line are loaded into four 128-bit SSE (Streaming SIMD Extensions) registers (e.g., `xmm0`-`xmm3`). The relevant multiplier is then selected among them by masking the register values accordingly. The masks are necessarily calculated on-the-fly based on the index of the multiplier to be used.

The OpenSSL's countermeasure requires modifications of two gathering functions, `bn_gatter5()` and `bn_mul_mont_gather5()`, in the source file `bn/x86_64-mont5.s`.

This results in 10–20% performance drops of the modular exponentiation in RSA algorithms.

3. Permute-Scatter-Gather Implementation

In this section, we give details on the Permute-Scatter-Gather (or Permute-SG in short), the proposed method for secure modular exponentiation against fine-grained cache attacks, which is also faster than constant-time gather procedure.

3.1. Threat Model

Cache attacks often target secret keys of a victim process performing an encryption algorithm. In this paper, we assume that an adversary is a process which is co-resident on the same machine as the victim process. Due to the memory protection provided by modern operating systems, an adversary process is prohibited to view the content of the victim's memory. Despite of the process isolation, however, a logical processor is shared among processes, by which the adversary can exploit the cache-bank conflict. By mounting the fine-grained cache attack, the adversary tries to learn about the victim's secret key. We also assume that the adversary is able to execute arbitrary programs on a processor core shared with the victim process. However, as we mentioned above, the adversary does not have access to the victim's memory space.

In our threat model, we do not require that the target executable binary (e.g., OpenSSL library) running in the victim be kept secret. That is, the adversary has sufficient information on a logical structure of the binary such as the control flow and the exploitable locations. However, the adversary has no information about the runtime states (e.g., secret keys or permutation tables) of the executable, which are located on the data section of the binary in the victim's process.

3.2. Overview and Design Goals

The idea of the Permute-SG is basically to unlink the index of a multiplier from its memory location, thereby making it infeasible to figure out the multiplier used during the exponentiation. For this, the proposed method obfuscates the memory locations of the multipliers through a pseudo-random permutation. Specifically, given an index idx and a pseudo-random permutation P, the location of the multiplier is determined by the permuted index $\mathsf{idx' = P(idx)}$. In this way, all the 32 multipliers are rearranged in the table according to their permuted indices. By mounting cache attacks, an adversary might get the trace of $\mathsf{P(idx)}$. However, he/she cannot infer which multipliers are actually used from the obtained trace.

We construct the Permute-SG technique with consideration of achieving the following design goals:

- *Resistance against fine-grained cache attacks.* No information about the actually accessed multiplier should be revealed to adversaries who can observe memory accesses with bank-level granularity.
- *Computational efficiency.* Performance degradation in modular exponentiation due to applying this method should be minimized.
- *Adaptability.* It should be easily integrated into the existing implementation (e.g., OpenSSL library) without significant modification of source codes.

3.3. Implementation

The Permute-SG is augmented with ease to the OpenSSL's scatter-gather implementation (i.e., the version of 1.0.2f). The procedure of the Permute-SG for the modular exponentiation is performed through the following steps:

1. Permute step: In this step, a permutation P is randomly generated from $\mathcal{P}$, the set of all permutations. The generation process is conducted along with the precomputation phase of modular exponentiation algorithm (Algorithm 1).
2. Scatter step: This step is the same as the scatter procedure in the OpenSSL, except that the scattering location of a multiplier with an index idx is determined by $\mathsf{P(idx)}$.

3. Gather step: This step is the same as the gather procedure in the OpenSSL, except that the gathering location of a multiplier with an index idx is determined by P(idx).

3.3.1. Challenging Issue

As described above, we can easily integrate the Permute-SG technique into the OpenSSL library, thus adaptability, one of our design goals, is trivially achieved. However, it is not trivial to achieve the other two design goals together when implementing the technique. That is, evaluating P with an index idx is a time consuming operation and it occurs at every scatter and gather step. This may lead to the significant performance degradation. The optimal solution is to implement the evaluation procedure using a permutation table. By looking up the table with idx, the value of P(idx) can be retrieved just within a few CPU cycles. For security perspective, however, the lookup operation with the permutation table is subject to the fine-grained cache attack. This is because the memory access to the table reveals the index of the used multiplier during exponentiation. Therefore, implementing the permutation with regard to efficiency and security is a challenging problem.

3.3.2. Constant-Time Permutation

We overcome the challenging problem by implementing *constant-time permutation*. It is a lookup-based technique that always has constant memory access pattern irrespective of the accessed index, thus revealing no information to adversaries. For the computational efficiency, the constant-time permutation is implemented in a x86 assembly. Since a memory access is a costly operation, the number of access needs to be minimized for the constant-time lookup procedure. We achieve this by utilizing only a single SSE load instruction. By doing so, the memory access time for the lookup can be confined to just a single CPU cycle in the case of the table being loaded to a L1 cache [26].

To load a permutation table into a SSE register by a single load instruction, we have to fit the size of the table within the width of the register. In most Intel x86 processors, SSE registers are 128 bits in length (Recent Intel processors support Advanced Vector Extension (AVX), in which the size of registers are more than 128 bits in length. For our technique to be widely deployed, we only consider SSE instructions in this paper). Please note that there are 32 multipliers in total, and thus the size of each index should be at least 5 bits in length. This indicates that a room of 160 bits is needed in the table to store all the indicies, which is larger than the size of the SSE register. We solve this problem in a way that the four leftmost bits of the index are stored in the table instead of all the bits being stored. This makes the four bits of the index to be permuted while the remaining rightmost bit is left unchanged during the permutation process.

Figure 2 illustrates the process of constant-time permutation. We have PermTab, an array with a length of 128 bits, which is divided into two 64-bit permutation tables, $\mathsf{PermTab_H}$ and $\mathsf{PermTab_L}$. The address of PermTab is 16 bytes aligned so that a single load instruction can load both tables into a SSE register. Two pseudo-random permutations P_0 and P_1, which are generated independently in the Permute step, are set up to those tables respectively. Each table contains a permuted list of partial indices of 4 bits in its slots $s_0, s_1, ..., s_{15}$ according to the permutation.

In the permutation process, the value of the four leftmost bits of idx, denoted by X in Figure 2, is used to lookup the values of the corresponding slots in the tables simultaneously. For instance, the case of $X = 2$ would make concurrent lookups to $\mathsf{PermTab_H}$ and $\mathsf{PermTab_L}$ with the same slot s_2, resulting $P_0(X)$ and $P_1(X)$. The remaining rightmost bit of idx, denoted by Y, is used to select the one among them. As a result, the permuted index idx' is constructed from $P_Y(X)$ and Y, where $P_Y \in \{P_0, P_1\}$ as shown in Figure 2. The memory location of the multiplier is then determined by the permuted index idx'.

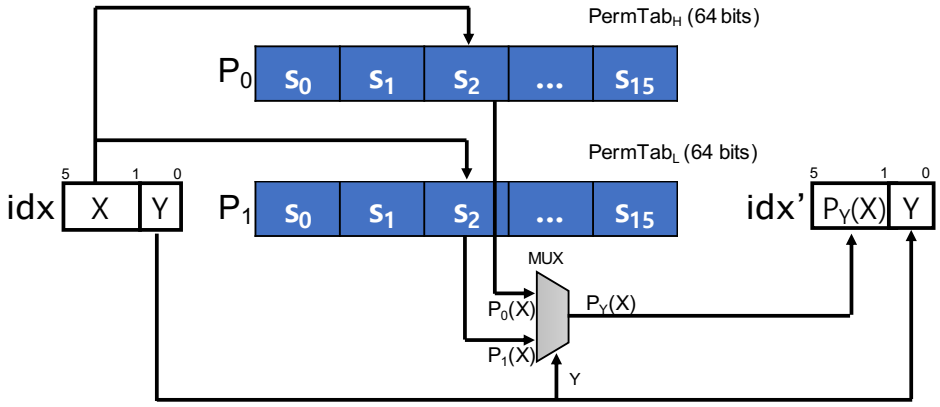

Figure 2. The process of constant-time permutation

Listing 1 presents the implementation of the constant-time permutation. The source code is written in perlasm, a x86 assembly language in the form of a perl script. In lines 1–2, the 16 bytes array of PermTab, which comprises PermTab$_H$ and PermTab$_L$, is loaded into a xmm1 register. In lines 3–9, the slots from PermTab$_H$ and PermTab$_L$, corresponding to the four leftmost bits of the index idx (denoted by $idx), are selected in the xmm1 register, and values in those slots are loaded to r10 and r11 registers, respectively. In lines 10–17, one of the values is chosen from r10 and r11 according to the rightmost bit of $idx, and saved to rax register. Finally, in lines 18–20, the permuted index idx' is produced from the value in rax and the rightmost bit of idx, and then loaded into $idx as an output.

Listing 1: The assembly of constant-time permutation.

```
1:  lea     .LPermTab(%rip),%rax
2:  movdqa  0(%rax), %xmm1
3:  mov      $idx, %rax
4:  shr      \$1, %rax
5:  shl      \$2, %rax
6:  mov      %rax, %xmm0
7:  psrlq   %xmm0, %xmm1
8:  pextrq  \$1, %xmm1, %r10
9:  mov      %xmm1, %r11
10:  and     \$1, $idx
11:  not     $idx
12:  add     \$1, $idx
13:  mov     %r11, %rax
14:  xor     %r10, %r11
15:  and     $idx, %r11
16:  xor     %r11, %rax
17:  and     \$15, %rax
18:  and     \$1, $idx
19:  shl     \$1, %rax
20:  add     %rax, $idx
```

4. Evaluation

4.1. Resistance Against Fine-Grained Cache Attacks

Suppose that an application $\mathcal{V}$ executes a modular exponentiation which is implemented with the Permute-SG technique. $\mathcal{V}$ might be a RSA application that performs a decryption with a RSA private key. By leveraging fine-grained cache attacks, an adversary $\mathcal{A}$ attempts to know the information

of the multiplier (i.e., the index idx) which is used when $\mathcal{V}$ conducts the gathering phase (i.e., Gather step in Section 3.3). $\mathcal{A}$ may observe the memory offset accessed by $\mathcal{V}$ at fine-grained granularity. The offset, however, only reveals the information of P(idx). Unless $\mathcal{A}$ knows the permutation P, he/she cannot infer idx from P(idx).

$\mathcal{A}$ may attempt to learn idx by observing the memory access to the array PermTab. As described in Section 3.3, the access to the permutation table occurs in a single load instruction (Line 2 in Listing 1) and is independent on the value of idx. Therefore, it is infeasible to know the index of the accessed multiplier by observing access to the permutation table.

4.2. Adaptability

The Permute-SG is designed to be easily augmented to the existing scatter-gather implementation of the OpenSSL library. As described in Section 3.3, the modification is only required in the library at the part of the precomputation of modular exponentiation as well as the part of locating the multiplier in Scatter and Gather steps.

4.3. Computational Efficiency

We conducted some benchmarks to evaluate the computational efficiency of the proposed method. For this, an OpenSSL library of the version 1.0.2f is modified by replacing its scatter-gather part with our Permute-SG implementation. We selected this version since it is vulnerable to fine-grained cache attacks [21]. The benchmarks were performed on a server equipped with a Xeon E5-2620v4 processor (Broadwell) and a PC with a Core i7-7820HQ processor (Kaby Lake). We used a benchmarking tool included in the OpenSSL framework, and measured the speed of the RSA signing and verifying operations for each implementation.

Table 1 and Figure 3 show the benchmarking results. The terms 'SG' and 'SG-Const' refer to the unmodified OpenSSL libraries of version 1.0.2f and 1.0.2g, respectively. Both have the scatter-gather implementation, of which the SG is vulnerable to fine-grained cache attacks while the SG-Const has a countermeasure with constant-time gather procedure (See Section 2.2.2). In Figure 3, the benchmarking results are illustrated in a relative manner to give an intuitive comparison. The SG shows the fastest performance result, which comes at the cost of lacking the countermeasure against the fine-grained cache attacks. Among the implementations with the countermeasure, the Permute-SG is the fastest in all the benchmarking cases. In Broadwell processor, the Permute-SG shows almost the same performance as the SG, and is 11% faster than the SG-Const for signing operation in RSA 4096-bits. Because of the microarchitectural difference, the Permute-SG shows a little performance degradation in Kaby Lake processor, in which it still outperforms the SG-Const. It is worth noting that in RSA 1024-bits, all the implementations show the same performance, because the scatter-gather is only applied to more than RSA 2048-bits in OpenSSL.

Table 1. The result of performance evaluation.

(a) Benchmark on Xeon E5-2620v4 (Broadwell)						
RSA Bits	**SG**		**SG-Const**		**Permute-SG**	
	sign/s	verify/s	sign/s	verify/s	sign/s	verify/s
1024	6698.2	96,683.1	6702.4	96,869.6	6555.4	96,262.6
2048	903.8	28,983.1	868.3	27,985.6	902.7	28,934.8
4096	126.4	7835.7	113.3	7400	125.7	7831.7
(b) Benchmark on Core i7-7820HQ (Kaby Lake)						
RSA Bits	**SG**		**SG-Const**		**Permute-SG**	
	sign/s	verify/s	sign/s	verify/s	sign/s	verify/s
1024	7442.1	108,685.8	7101.9	97,294.2	7544.7	103,288.5
2048	999.6	31,684.9	889.0	28,972.2	911.6	29,650.9
4096	140.9	8748.2	116.3	7877.3	128.9	7977.5

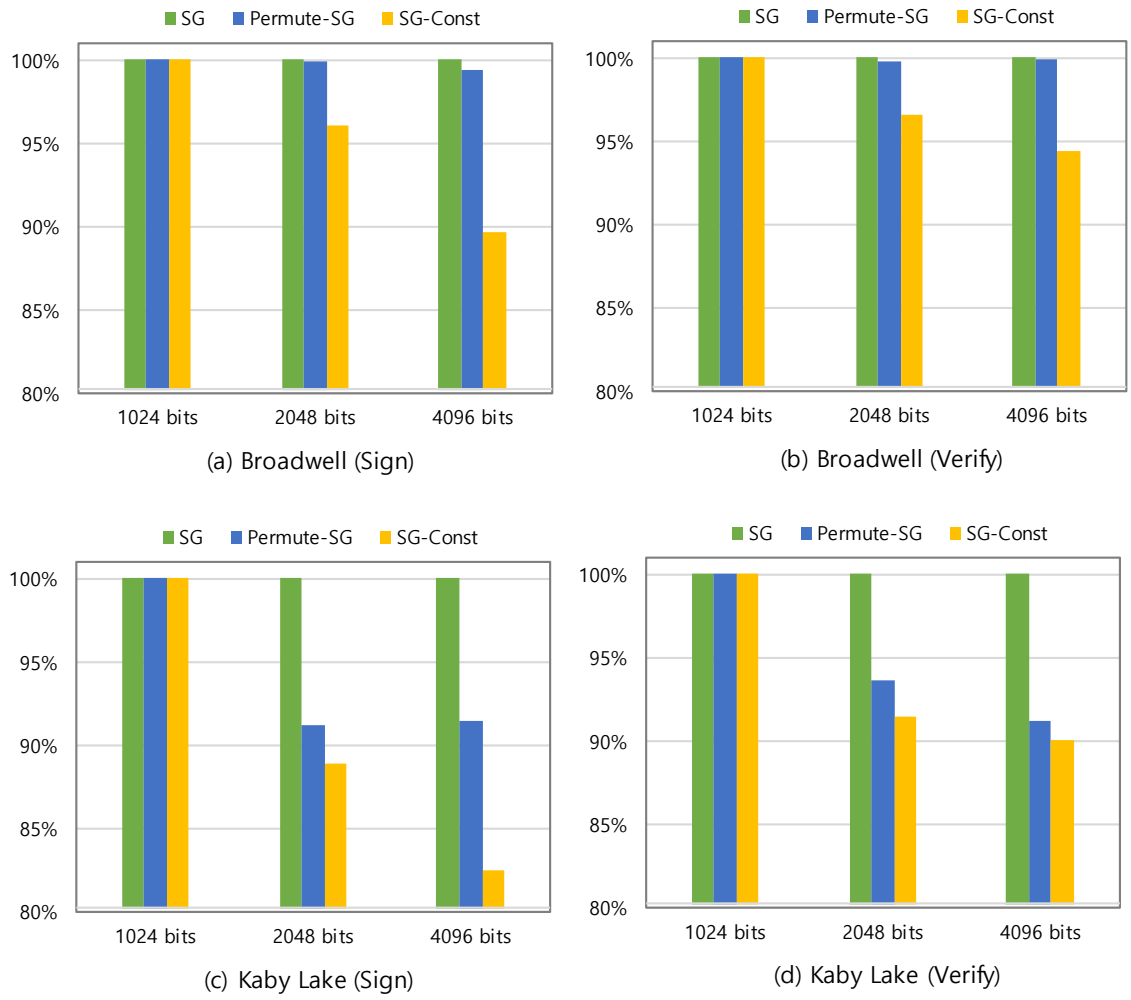

Figure 3. Comparison in the benchmarking results of scatter-gather implementations

5. Conclusions

In this paper, we proposed Permute-Scatter-Gather, a novel constant-time method for the modular exponentiation in the RSA cryptosystem. Based on the scatter-gather design, we utilized pseudo-random permutation in the construction to obfuscate memory access patterns so as to mitigate

fine-grained cache attacks. Throughout rigorous evaluations, we showed that our method provides the required security, computational efficiency as well as adaptability, making it practicable in real world applications.

Author Contributions: Y.S. wrote this article. He performed analysis on the proposed method in terms of both security and performance as well.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No.2017R1C1B5015045) and by the MISP (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).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Liu, F.; Yarom, Y.; Ge, Q.; Heiser, G.; Lee, R.B. Last-level cache side-channel attacks are practical. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–20 May 2015; pp. 605–622.
2. Irazoqui, G.; Eisenbarth, T.; Sunar, B. S$A: A shared cache attack that works across cores and defies VM sandboxing—And its application to AES. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 591–604.
3. Gulmezoglu, B.; Irazoqui, G.; Eisenbarth, T.; Sunar, B. Cache Attacks Enable Bulk Key Recovery on the Cloud. In Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES 2016), Santa Barbara, CA, USA, 17–19 August 2016; pp. 368–388.
4. Disselkoen, C.; Kohlbrenner, D.; Porter, L.; Tullsen, D. PRIME+ABORT: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In Proceedings of the 26th USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; pp. 51–67.
5. Yarom, Y.; Falkner, K. Flush + Reload: A High Resolution , Low Noise , L3 Cache Side-Channel Attack. In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 719–732.
6. Yarom, Y.; Benger, N. *Recovering OpenSSL ECDSA Nonces Using the Flush + Reload Cache Side-Channel Attack*; IACR Cryptol. ePrint Archive, Report 2014/140; International Association for Cryptologic Research: Santa Barbara, CA, USA, 2014.
7. Berk, G.; Inci, M.S.; Irazoqui, G.; Eisenbarth, T.; Sunar, B. A Faster and More Realistic Flush + Reload Attack on AES. In Proceedings of the International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2015), Graz, Austria, 14–15 April 2015; pp. 111–126.
8. Gruss, D.; Spreitzer, R.; Mangard, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In Proceedings of the 24th USENIX Security Symposium, Washington, DC, USA, 12–14 August 2015; pp. 897–912.
9. Gruss, D.; Maurice, C.; Wagner, K.; Mangard, S. 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), Berlin, Germany, 13–14 July 2016; pp. 279–299.
10. Zhang, Y.; Juels, A.; Reiter, M.K.; Ristenpart, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In Proceedings of the 2014 SIGSAC ACM Conference on Computer and Communications Security (CCS 2014), Scottsdale, AR, USA, 3–7 November 2014; pp. 990–1003.
11. Ge, Q.; Yarom, Y.; Cock, D.; Heiser, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *J. Cryptogr. Eng.* **2018**, *8*, 1–27. [CrossRef]
12. Garman, C.; Green, M.; Kaptchuk, G.; Miers, I.; Rushanan, M. Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage. In Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX, Austin, TX, USA, 10–12 August 2016.
13. García, C.P.; Brumley, B.B. Constant-Time Callees with Variable-Time Callers. In Proceedings of the 26th USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; pp. 83–98.

14. Genkin, D.; Valenta, L.; Yarom, Y. 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), Dallas, TX, USA, 30 October–3 November 2017; pp. 845–858.
15. Kaufmann, T.; Pelletier, H.; Vaudenay, S.; Villegas, K. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015. In Proceedings of the 15th International Conference on Cryptology and Network Security (CANS 2016), Milan, Italy, 14–16 November 2016; pp. 573–582.
16. Gopal, V.; Guilford, J.; Ozturk, E.; Feghali, W.; Wolrich, G.; Dixon, M. Fast and Constant-Time Implementation of Modular Exponentiation. In Proceedings of the Embedded Systems and Communications Security, Niagara Falls, NY, USA, 27 September 2009.
17. Ronald, L.; Rivest, A.S.; Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **1978**, *21*, 120–126.
18. OpenSSL. *OpenSSL, Cryptography and SSL/TLS Toolkit*. Available online: https://www.openssl.org/ (accessed on 5 August 2018).
19. Brickell, E. Technologies to improve platform security. In Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems, Nara, Japan, 28 September–1 October 2011.
20. Brickell, E. The impact of cryptography on platform security. In Proceedings of the CT-RSA 2012, San Francisco, CA, USA, 27 February–2 March 2012.
21. Yarom, Y.; Genkin, D.; Heninger, N. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *J. Cryptogr. Eng.* **2017**, *7*, 99–112. [CrossRef]
22. Bos, J.; Coster, M. Addition chain heuristics. In Proceedings of the Conference on the Theory and Application of Cryptology CRYPTO, Santa Barbara, CA, USA, 20–24 August 1989.
23. Ernie Brickell, G.G.; Seifert, J.P. Mitigating cache/timing based side-channels in AES and RSA software implementations. In Proceedings of the RSA Conference 2006 Session DEV-203, San Jose, CA, USA, 13–17 February 2006.
24. Fog, A. *The Microarchitecture of Intel, AMD and via CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers*; Technical University of Denmark: Lyngby, Denmark, 2016.
25. Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual (April 2018)*; Intel: Santa Clara, CA, USA, 2018.
26. Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals (March 2018)*; Intel: Santa Clara, CA, USA, 2018.