# Cache Side-channel Attacks and Defenses of the Sliding Window Algorithm in TEEs

Zili KOU
Hong Kong University of Science and Technology
zkou@connect.ust.hk

Sharad Sinha
Indian Institute of Technology Goa
sharad@iitgoa.ac.in

Wenjian HE
Hong Kong University of Science and Technology
wheac@connect.ust.hk

Wei ZHANG
Hong Kong University of Science and Technology
wei.zhang@ust.hk

***Abstract*—Trusted execution environments (TEEs) such as SGX on x86 and TrustZone on ARM are announced to protect trusted programs against even a malicious operation system (OS), however, they are still vulnerable to cache side-channel attacks. In the new threat model of TEEs, kernel-privileged attackers are more capable, thus the effectiveness of previous defenses needs to be carefully reevaluated. Aimed at the sliding window algorithm of RSA, this work analyzes the latest defenses from the TEE attacker's point of view and pinpoints their attack surfaces and vulnerabilities. The mainstream cryptography libraries are scrutinized, within which we attack and evaluate the implementations of Libgcrypt and Mbed TLS on a real-world ARM processor with TrustZone. Our attack successfully recovers the key of RSA in the latest Mbed TLS design when it adopts a small window size, despite Mbed TLS taking a significant role in the ecosystem of ARM TrustZone. The possible countermeasures are finally presented together with the corresponding costs.**

***Index Terms*—cache side channel, RSA, TrustZone**

## I. Introduction

RSA [1] decryption is a widely used public-key cryptosystem for secure applications, though, its software implementations continue to be revealed vulnerable to cache side-channel attacks. We first demonstrate how previous side-channel attackers leak private keys of RSA decryption. Then, the motivation for this work is presented followed by our contributions.

### A. Previous Works

The sliding window algorithm is usually adopted to compute modular exponentiation in RSA. It implements a window scanning from the most significant to the least significant bits. Whenever the conditions are met, the algorithm accesses different multipliers to conduct modular multiplications "based on the value of the current window", which forms secret-dependent memory accesses that can be exploited by cache side-channel attackers. In the user space, cache side-channel attackers [2]–[4] monitor the behaviors of cache hierarchies, locate the target cache sets occupied by multipliers, and repeatedly conduct cache side-channel measurements. A precise profile of the access pattern of multipliers can be recovered into the private key based on attacker's knowledge. However, in practical it is unavoidable to detect some noise so profiling multiple traces are usually needed and partial key recovery [5], [6] is also introduced to help eliminate misalignments, bit-reverses, etc. In the kernel space, TEE attackers often utilize their high privileges to interrupt victim programs and conduct cache side-channel attacks with much higher precision. E.g., SGX-Step [7] achieves instruction-level precision when interrupting a program in SGX, CacheZoom [8] shows how SGX amplifies the power of cache attacks, and Load-Step [9] and Ryan's work [10] interrupt TrustZone programs and demonstrate high precision cache side-channel attacks.

### B. Motivation and Research Gap

The new threat model of TEEs brings extra challenges to defending against cache side-channel attacks so existing defenses, which are designed without enough TEE-specific security concerns, may still be thoughtless and flawed. In the case of the sliding window algorithm, cache side-channel vulnerabilities revealed by academia and industry are usually fixed up by case-by-case study [11]. To protect the sliding window algorithm in the TEE scenario, a rigorous defense scheme that takes the software origin of vulnerabilities and the capabilities of TEE attackers into account are desired, without which software designers will be less confident to promise their implementations are secure against kernel-privileged attackers. Defenses against user space cache side-channel attacks [2]–[4] are already deployed by the mainstream cryptography libraries. Schwarz et al. [12] conducts cache side-channel attacks inside the SGX enclave to leak keys of RSA in the rich world, and Load-Step [9] presents the single trace attack on RSA in the TrustZone scenario, however, they work mainly from the attacker's aspect without diving into the design of defenses and they are defended by latest cryptography libraries, too.

This work systematically scrutinizes the sliding window algorithms in the mainstream cryptography libraries and evaluates the effectiveness of their defenses in TEEs. Based on our analysis, we pinpoint the vulnerability in the latest Mbed TLS library and soon verify it by cache side-channel attacks, despite Mbed TLS being one of the cryptography providers of TrustZone. Summarized, we make the following contributions:

- Scrutinize the latest cryptography libraries to evaluate the sliding window algorithm from the kernel-privileged attacker's point of view
- Support our statements by real-world cache side-channel attacks against RSA decryption on an ARM processor with the reference implementation of TrustZone
- Reveal that the RSA implementation in the latest Mbed TLS library is still vulnerable, and demonstrate a comprehensive attack methodology to recover the keys
- Propose software mitigation against our attacks and present a preliminary analysis of the penalty

### C. Experiment Platform

Attacks in this paper are conducted on a real-world board named Hikey 960 equipped with the Hisilicon Kirin 960 system-on-chip (SoC). It involves 4 A53 cores (16-way set-associative L2 cache with 512 sets) and 4 A73 cores based on ARM big.LITTLE architecture and the cache side-channel measurements are implemented on the L2 cache of an A53 core. In terms of software, Hikey 960 is currently supported by the reference implementation of TrustZone by Trusted-Firmware [13], i.e., TF-A + OPTEE + Mbed TLS, which is also why we choose Hikey 960 for evaluation. In the rich world, the buildroot OS with Linux kernel 5.5 is implemented.

### D. Threat Model and Assumptions

We adopt a common threat model for kernel-privileged attackers and totally obey the assumptions made by TrustZone. We assume the RSA decryption is implemented as a trusted service in the secure world of TrustZone, and the attacker invokes attacks and measurements by a malicious kernel driver. For the victim RSA program, we assume the RSA-2048 is implemented with the public exponent $e$ set to 65537, and the CRT mode is applied. In the cache side-channel attacks, we reproduce interrupt-based attacks such as [9], [10], though, we do not focus on the precision of interrupts, but rather on reliable analysis of the victim program. We denote an **interrupt round** as anytime when the attacker invokes an interrupt forwarding to the victim, stalling the execution of the trusted application and making some measurements and observations. During every interrupt round, Prime+Probe attack is conducted to detect the whole L2 cache of the A53 core.

## II. Preliminaries

### A. Cache Side Channel Attacks in TEEs

*1) Prime+Probe:* Prime+Probe [2] is the most common technique for eviction-based cache side-channel attacks. Before attacking, the attacker first constructs an eviction set, i.e, a group of addresses that are mapped to the same cache set. Three steps are involved in every measurement: *prime*, *wait*, and *probe*. In the *prime* step, the attacker loads the eviction set to fully occupy a cache set. During the *wait* procedure, if the victim loads data that is mapped to the same cache set, data cached by the attacker will be evicted. A longer reloading time of the eviction set can be detected in the *probe* procedure, indicating that the cache set has been previously accessed by the victim.

*2) Interrupt-based attacks:* TEE attackers usually utilize the interrupt mechanism of the OS to significantly improve the performance of attacks, and they seek to increase the interrupt frequency as it determines the precision of side-channel measurements [7], [9]. In the interrupt-based attacks using Prime+Probe, attackers periodically invoke interrupts to trusted programs and then conduct side-channel measurements (*prime*). Before releasing the interrupt threads, the side-channel contents are prepared well (*probe*) for the next interrupt round.

**Algorithm 1:** Sliding Window Algorithm

```
Input: base b, modulus m, exponent d = {d_n...d_1}_2
Output: b^d (mod m)
precompute multipliers M[2^{w-1}] to M[2^w - 1].
r ← 1, i ← n
while i > w − 1 do
    if d_i = 0 then
        r ← r^2 (mod m)              // modular square r
        i ← i − 1
    else
        repeat w times
            r ← r^2 (mod m)          // modular square r
        end
        j ← {d_i...d_{i−w+1}}_2
        r ← r × M[j] (mod m)         // modular multiply r by M[j]
        i ← i − w
    end
end
while i > 0 do
    r ← r^2 (mod m)
    i ← i − 1
    if d_i = 1 then
        r ← r × M[1] (mod m)         // process remaining bits
    end
end
return r
```

### B. RSA Cryptosystem

*1) Arithmetic Principle:* RSA cryptosystem starts by generating two random primes $p, q$ and offering $(e, N)$ as the public key, where $N = pq$ and $e$ is a (usually fixed) public exponent. For a given $e$, the RSA program computes $d = e^{-1} \pmod{(p-1)(q-1)}$ to form the private key $(d, p, q)$. One can encrypt the plaintext $m$ into the ciphertext $s$ by computing $s = m^e \pmod N$ and decrypt by computing $m = s^d \pmod N$. In real-world applications, $(N, d, p, q)$ are very large numbers so factorizing $N$ without any information about $(d, p, q)$ is computationally infeasible. To speed up the heavy computation, an optimization based on Chinese Remainder Theorem (CRT) is always deployed, which splits the original modular exponentiation into two operations with smaller operands: $m_p = s^{d_p} \pmod p$ and $m_q = s^{d_q} \pmod q$, where $d_p = d^{-1} \pmod{(p-1)}$ and $d_q = d^{-1} \pmod{(q-1)}$. The plaintext $m$ can be finally reconstructed with the input $(p, q, m_p, m_q)$.

*2) Sliding Window Algorithm:* As shown in Algorithm 1, the general implementation of the sliding window algorithm obeys the Montgomery reduction [14] to ease the computing of modular exponentiation, i.e., inputs base $b$, exponent $d$, and modulus $m$ and outputs $b^d \pmod m$. A window of size $w$ is implemented to scan the bits of the exponent. To optimize the modular multiplication, the multipliers are precomputed as $M[2^w - 1]$. The algorithm starts with an intermediate result $r$ and conducts two operations based on the values of the current window: modular square $r$ (for times) or modular multiply $r$ by one of the multipliers $M[i]$. As a consequence, cache side-channel attackers can observe the memory access pattern to figure out when and which multiplier has been loaded by the victim and finally recover the exponent bits based on the attackers' knowledge of the algorithm.

### C. Partial Key Recovery

The partial key recovery takes effect when only part of the key bits are known, and the correct key is recovered based on the potential arithmetic principles of cryptosystems. A survey of techniques for the partial key recovery is made in [5], within which we utilize *Branch-and-Prune* [15] to enable our attacks. *Branch-and-Prune* works when $d_p$ and $d_q$ are partially known in RSA with CRT optimization, and this technique is generalized in cache side-channel attacks by [16], [17].

*1) Branch-and-Prune:* For CRT exponents $d_p$ and $d_q$, it holds $ed_p = k_p(p-1)+1$ and $ed_q = k_q(q-1)+1$ for some $1 \leq k_p, k_q < e$. Multiplying them together, we obtain an important equation that helps verify the correctness of $d_p$ and $d_q$:

$$(ed_p - 1 + k_p)(ed_q - 1 + k_q) = k_p k_q N \tag{1}$$

reducing Equation 1 modulo e, we get $(k_p-1)(k_q-1) \equiv k_p k_q N \pmod{e}$. In the most common case (so as the scrutinized cryptography libraries), $e$ is set to 65537. Hence, there is at most 65536 pairs of $k_p$ and $k_q$ and it is computationally feasible to brute force them, as the verification with an incorrect pair will end earlier. Branch-and-Prune starts recovering $d_p$ and $d_q$ from the lease significant bit. At a specific location, if a bit of $d_p$ and $d_q$ is unknown, the algorithm branches to create two threads, assuming the bit is '0' or '1'. Whenever $d_p$ and $d_q$ in a thread do not conform to Equation 1 at the $i^{th}$ bit, i.e., $(ed_p - 1 + k_p)(ed_q - 1 + k_q) \neq k_p k_q N \pmod{2^i}$, the algorithm prunes such a thread. Given correct $k_p$ and $k_q$, the verification will finally generate fully recovered $d_p$ and $d_q$ after iterating for every bit. Branch-and-Prune is, unsurprisingly, not omnipotent as the running time grows exponentially in the number of unknown bits. Heuristically, it fails when the known bits of $d_p$ and $d_q$ are fewer than 50% [16].

## III. Defend Sliding Window Algorithms

In this section, we first show how the traditional sliding window algorithm is broken down by cache side-channel attackers. After scrutinizing the mainstream libraries, we summarize the existing defenses and classify them into three types, i.e., exponent blinding, multiplier obfuscation, and square&multiply obfuscation, as shown in Table I. Each type of defense is explained followed by how it is disqualified (if possible).

TABLE I
Three Types of Defenses in Cryptography Libraries.

| Cryptography Library | Exponent Blinding | Multiplier Obfuscation | Square&Multiply Obfuscation |
|---|---|---|---|
| Libgcrypt 1.10.1 | ✓ | ✓ | ✓ |
| OpenSSL 3.1.0 | | ✓ | |
| WolfSSL 5.3.0* | | | |
| Mbed TLS 2.26.0 | ✓ | | |
| Mbed TLS 3.1.0 | ✓ | ✓ | |

*designed to be lightweight and portable.

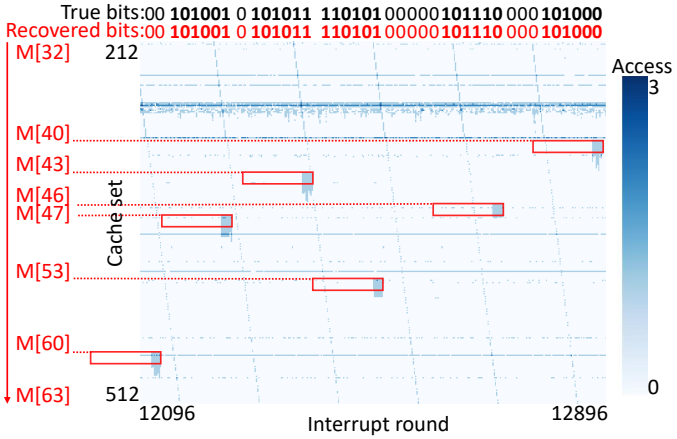

Fig. 1. Our attack against the sliding window algorithm in Mbed TLS 2.26.0.

### A. Attack Naive Implementations

Naive implementations of the sliding window algorithm expose secret-dependent memory accesses to cache side-channel attackers, i.e., loading different multipliers based on the bits value of the current window. In the user space, attackers need to observe the behaviors of the cache hierarchy to figure out the cache sets that store the multipliers. In the profile stage, attackers keep conducting cache side-channel attacks to profile the access pattern of the specific cache sets. Based on attackers' knowledge, each access of a multiplier can be decoded into a portion of key bits, and the full recovery of the exponent is finally obtained. Usually, attackers invoke the RSA decryption and profile it multiple times to overcome the noise and misalignment. E.g., Liu et al. [2] need 10-15 traces for each multiplier, and Schwarz et al. [12] fully recover the exponent with 11 traces.

### B. Exponent Blinding

As multiple traces are needed for the full recovery of exponent bits, one solution is to randomize the exponent for every modular exponentiation. Exponent blinding represents a blinding strategy that calculates a random number and adds it with the original exponent, while keeping the arithmetic result unchanged. In detail, the two operations in RSA CRT mode, $s^{d_p} \pmod{p}$ and $s^{d_q} \pmod{q}$, now becomes $s^{d_p + r(p-1)} \pmod{p}$ and $s^{d_q + r(q-1)} \pmod{q}$, and $r$, with the length of $m$ bits, is randomly generated for every invoking of the RSA decryption. The results are unchanged as $x^{(p-1)} \equiv 1 \pmod{p}$ if $p$ is prime and $x \neq 0 \pmod{p}$ (Fermat–Euler theorem). With the exponent blinding applied, cache side-channel attackers that require $n$ traces are expected to profile $2^{(m-m/n)}$ times to create $n$ collisions, which is computationally infeasible. E.g., Liu et al. [2] that need 10 traces now requires more than $2^{202}$ observations if adopts 28 bytes blinding.

**Disqualified:** However, the exponent blinding is completely breached if attackers need not profile multiple traces, instead, they can fully recover the blinded exponent by a single trace. Unfortunately, such single trace attacks are feasible for interrupt-based attackers [9], [18]. We conduct a single trace cache side-channel attack against the sliding window

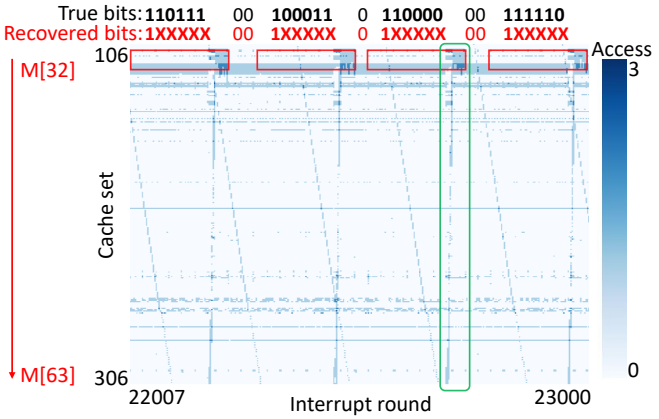


Fig. 2. Our attack against the sliding window algorithm in Mbed TLS 3.1.0.

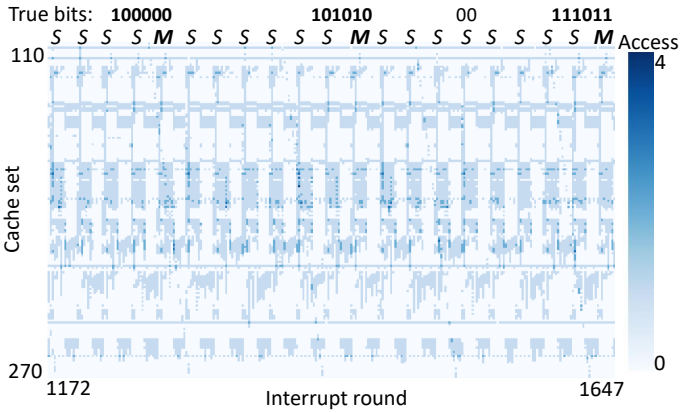


Fig. 3. Our attack against the sliding window algorithm in Libgcrypt 1.10.1, where the 'S' denotes the square and the 'M' denotes the multiplication.

algorithm in Mbed TLS 2.26.0 and generate the profile shown in Fig. 1. The block patterns are the memory accesses of the multipliers and they are clear enough to be recovered into exponent bits: 1) From which cache set a multiplier is mapped to we know the index $j$ of the multipliers. 2) Decode each multiplier $M[j]$ into exponent value $j$ in binary. 3) Fill in '0' bits between multipliers. The higher the interrupt frequency in our attack, the less noise and misalignment but also the longer execution time there would be. Again, we focus more on the feasibility rather than the ultimate performance. Specifically, the demonstrated attack in Fig. 1 achieves the full recovery from a single trace by 19228 interrupt rounds, and it takes 16.5 seconds for cache side-channel measurements and 3 seconds for the pattern recognition. The recovered exponents $d_p+r(p-1)$ and $d_q+r(q-1)$ are modular equivalent to $d_p$ and $d_q$ and can be directly used for the following decryption. To completely leak the private keys, one of the methods is to find the common prime factors between several $e(d_p+r_i(p-1))-1$, from which we know the value of $p-1$ so that successfully factorize $N$.

### C. Multiplier Obfuscation

Loading different multipliers based on the exponent bits is the main secret-dependent memory access in the sliding window algorithm. Therefore, concealing or obfuscating the access patterns of the multipliers can significantly trouble cache side-channel attackers. A straightforward idea is to make the access patterns the same no matter which multiplier is loaded. Until now, two approaches are presented: scatter-gather in OpenSSL and traverse-select in Libgcrypt and Mbed TLS. The scatter-gather implementation tries to avoid the memory accesses of the multipliers at coarser than cache line granularity, i.e., instead of being stored as consecutive bytes, each multiplier is scattered and distributed across several cache lines. As a consequence, loading every multiplier will access the same group of cache lines so that cache side-channel attackers cannot distinguish the multipliers by monitoring cache sets. The traverse-select implementation allocates a new memory area to store the multiplication operand. Whenever a multiplier $M[j]$ is needed, all the multipliers are loaded in order, though, only the $j^{th}$ loading round writes back to the allocated memory area. We conduct the same single trace cache side-channel attack against the sliding window algorithm in Mbed TLS 3.1.0, as shown in Fig. 2, the area circled in green depicts the memory access patterns of the multipliers with traverse-select applied.

**Disqualified:** The multiplier obfuscation successfully conceals which multiplier is accessed, however, it does not conceal when the multiplier is accessed, which is another secret-dependent memory access that is usually ignored by programmers. Based on attackers' knowledge of the sliding window algorithm, a multiplier is loaded when the value of the current window matches from $2^{w-1}$ to $2^w-1$, i.e., it always starts with a bit '1'. Besides, between two multiplication operations, each bit '0' requires a square operation. Therefore, there are still some bits that cache side-channel attackers are able to leak, and the percentage of known bits is determined by the window size $w$. Fig. 2 depicts the partially recovered exponent bits even though the multiplier obfuscation is applied. Statistical experiments show that around 50% bits of modular exponentiation are leaked if $w=3$, which satisfies the prerequisite of the Branch-and-Prune technique. To summarize, the multiplier obfuscation is still vulnerable when $w<4$.

### D. Square&Multiply Obfuscation

The square&multiply obfuscation aims to strictly remove the secret-dependent memory access of the multipliers, i.e., conceal both which and when the multiplier is accessed. As the multiplication operand is the output number itself in the square operation, programmers achieve the square&multiply obfuscation by adding the output number into the loading sequence of traverse-select. Hence, the square and multiplication are treated the same, i.e., the algorithm loads all the multipliers and the output number in order but only writes back the needed one. Our interrupt-based cache side-channel attack against the sliding window algorithm in Libgcrypt 1.10.1 is shown in Fig. 3 that the access patterns of traverse-select appear uniformly. With the square&multiply obfuscation applied, cache side-channel attackers even cannot leak when the multiplier is loaded therefore no bit is recovered from the profile.

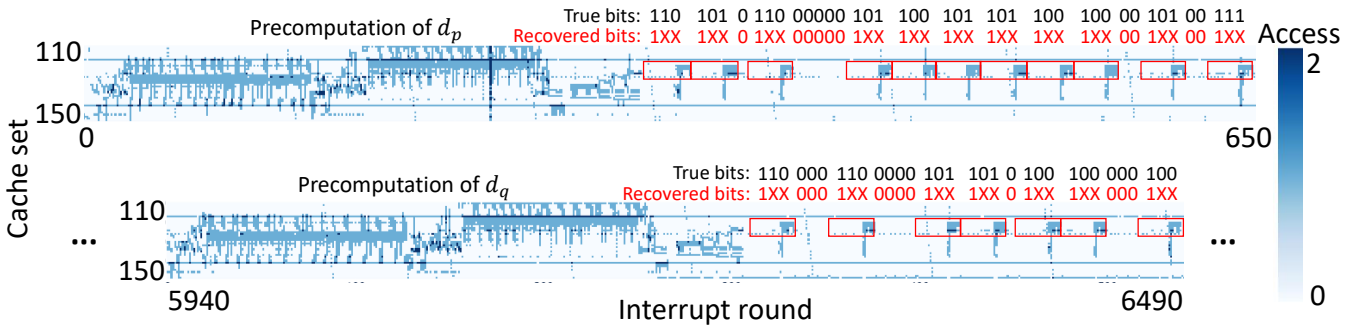


Fig. 4. Our practical attack against the 2048-bit RSA implemented by Mbed TLS 3.1.0.

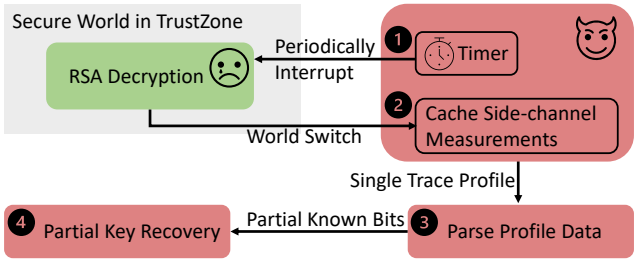


Fig. 5. Our comprehensive attack methodology.

**Drawback:** The secret-dependent memory accesses are removed as much as possible by the square&multiply obfuscation, however, this inevitably results in a huge loss of performance. In the sliding window algorithm, the square operations are much more than the multiplication operations. To deal with a 2048-bit exponent, there are 2048 square operations and fewer than $2048/w$ multiplication operations. Obviously, the performance of the sliding window algorithm suffers a lot if the traverse-select implementation is not only triggered for the multiplication but also for the square operation.

## IV. Attack Latest Mbed TLS

In this section, we utilize our analysis to point out that the sliding window algorithm in the latest Mbed TLS is still vulnerable to kernel-privileged attackers. We first demonstrate a comprehensive attack methodology and then present the experiment results.

### A. Attack Methodology

The workflow of our attack is shown in Fig. 5 that there are totally 4 steps. ❶ The RSA decryption is triggered to operate in the TEE enclave and the kernel-privileged attacker periodically invokes an interrupt to temporarily stall the execution of the victim program. ❷ During every interrupt round, cache side-channel measurements (Prime+Probe) are conducted for all the L2 cache sets. ❸ The RSA decryption is done, and so is the interrupt-based attack. The raw profile data is parsed by our python scripts, resulting in two partial known key bits of $d_p$ and $d_q$. ❹ We implement Branch-and-Prune in C and utilize it to fully recover the private keys. The recovery program will end prematurely if it runs for too long.

### B. Results

We conduct such attacks against the RSA decryption in the latest Mbed TLS library, i.e., version 3.1.0, which is equipped with the exponent blinding and the multiplier obfuscation. In the case of $w = 3$, we interrupt the RSA decryption for 12534 rounds and plot the profile as Fig. 4. The sliding window algorithm is executed twice due to the CRT optimization. Therefore, our parsing program first needs to locate the modular exponentiation of $d_p$ and $d_q$ separately, which is not difficult as the sliding window algorithm precomputes the multipliers and leaves unique memory access patterns. The identification of the multipliers is simple, as the patterns of the multipliers are visually recognizable and the profile is noiseless and free from misalignment. The red symbols in Fig. 4 depict how we partially recover the exponent bits: label '1XX' at the location of every multiplier, and fill in '0's between two multipliers. Finally, the partial known bits exponent are passed to the Branch-and-Prune program, and we verify the recovered exponents with the true $d_p$ and $d_q$.

TABLE II
Experiment Results.

| Window Size | Interrupt Round | Partial Known Bits | Execution Time |
|---|---|---|---|
| $w = 5$ | 10564 | 34.6% | 10.5 s + 3 s + $>$ 48 h |
| $w = 4$ | 11869 | 40.5% | 11.7 s + 3 s + $>$ 48 h |
| $w = 3$ | 12967 | 50.8% | 12.6 s + 3 s + 0.6 s |
| $w = 2$ | 13294 | 66.8% | 13.0 s + 3 s + 0.03 s |
| $w = 1$ | 16630 | 100% | 15.2 s + 3 s + 0 s |

When attacking the sliding window algorithm with different window sizes, we always keep the interrupt frequency unchanged. The detailed experiment results averaged by 1000 iterations are listed in Table II, including the interrupt round, the partial known bits, and the execution time (cache attacks + profile parsing + partial key recovery). Our attacks successfully recover the private keys by a single trace when $w < 4$, and the attack and the recovery procedures can finish in seconds. When $w \geq 4$, the partial known bits recovered from the attacks are too few to satisfy the prerequisite of Branch-and-Prune so the recovery program fails to complete and ends early after 2 days.

## V. Mitigation

This section discusses the possible mitigation for our attacks and the corresponding penalty. We first analyze the feasibility and the impact of limiting the window size. Then we implement square&multiply obfuscation into the latest

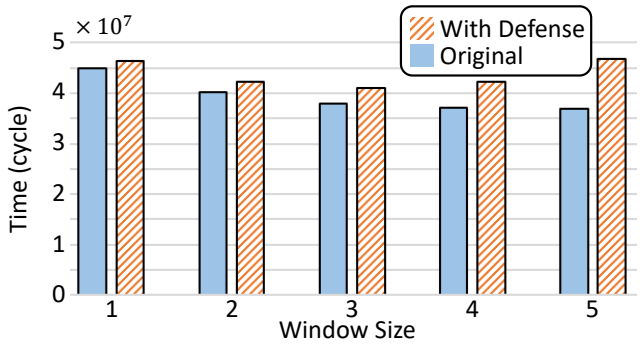


Fig. 6. The Execution time of the sliding window algorithm of Mbed TLS 3.1.0, with or without the square&multiply obfuscation applied.

Mbed TLS design and evaluate the performance loss on our tested platform.

### A. Limit Window Size

As the window size determines the number of partial known bits so that limits the success of Branch-and-Prune, one intuitive countermeasure is to avoid using a small window size. However, the window size is not a parameter that can be changed at will. On one hand, there exist some conventions of the window size, e.g., in Libgcrypt, RSA-1024 uses $w = 4$ and RSA-512 uses $w = 3$. On the other hand, a large window size burdens the memory and the cache hierarchy, i.e., 32 multipliers (when $w = 6$) need to be allocated to around 136 KB continuous memory and occupy 272 cache sets, which may be unacceptable for lightweight embedded systems. Until now, we have not discovered any notice about setting the window size for security purposes in the latest cryptography libraries. For Mbed TLS, as the important component in the ecosystem of TrustZone that may be adopted by a diverse range of devices, we recommend adding optional limitations rather than strictly forcing $w > 4$.

### B. Import Square&Multiply Obfuscation

Another countermeasure is to import the square&multiply obfuscation into Mbed TLS, and the performance cost becomes the main concern. We only make simple modifications for a preliminary evaluation of the performance, i.e., adding the output number into the loading sequence of traverse-select and invoking traverse-select for every square and multiplication. In addition, the set-up of the evaluation platform may not be general and convincing, as the cache hierarchy impacts performance a lot. For a more rigorous evaluation, we leave it to future works. We implement the sliding window algorithm of Mbed TLS, with and without the square&multiply obfuscation, to compute the 2048-bit modular exponentiation and assign it to operate on an A53 core. The execution time averaged by 1000 iterations is shown in Fig. 6 that the performance degrades for all cases. The possible reason accounting for the huge performance loss when using a large window size suchlike $w = 5$ is that a larger window size generates more multipliers to be sequentially loaded during traverse-select. Interestingly, the selection of a larger window size, which is previously an optimization, now suffers from the square&multiply obfuscation defense. Therefore, we believe careful analysis of the trade-offs is needed when importing the square&multiply obfuscation.

## VI. Conclusion

In this paper, we discuss the capabilities of kernel-privileged attackers against the sliding window algorithm. A systematic analysis of the latest defenses adopted by the mainstream cryptography libraries is summarized, based on which we pinpoint the vulnerability of the sliding window algorithm implemented by the latest Mbed TLS. Our comprehensive attack methodology and the analysis of the mitigation offer practical inspiration to both attackers and defenders.

## Disclosure

The vulnerability is assigned CVE-2021-46392 as the public identifier. Mbed TLS confirmed and patched the vulnerability well. We appreciate Mbed TLS's helpful feedback and professional handling of our report.

## References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[3] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[4] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud," *Cryptology ePrint Archive*, 2015.

[5] G. De Micheli and N. Heninger, "Recovering cryptographic keys from partial information, by example," *Cryptology ePrint Archive*, 2020.

[6] K. G. Paterson, A. Polychroniadou, and D. L. Sibborn, "A coding-theoretic approach to recovering noisy rsa keys," in *ASIACRYPT*. Springer, 2012, pp. 386–403.

[7] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *SysTEX*, 2017.

[8] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *CHES*, 2017.

[9] Z. Kou, W. He, S. Sinha, and W. Zhang, "Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+ evict," in *2021 58th ACM/IEEE DAC*. IEEE, 2021, pp. 979–984.

[10] K. Ryan, "Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 181–194.

[11] "MbedTLS security disclosures." [Online]. Available: https://tls.mbed.org/security

[12] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *DIMVA*. Springer, 2017, pp. 3–24.

[13] "Trusted firmware." [Online]. Available: https://www.trustedfirmware.org/

[14] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[15] N. Heninger and H. Shacham, "Reconstructing rsa private keys from random key bits," in *Annual International Cryptology Conference*. Springer, 2009, pp. 1–17.

[16] D. J. Bernstein, J. Breitner, D. Genkin, L. Groot Bruinderink, N. Heninger, T. Lange, C. v. Vredendaal, and Y. Yarom, "Sliding right into disaster: Left-to-right sliding windows leak," in *CHES*, 2017.

[17] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[18] A. C. Aldaya and B. B. Brumley, "When one vulnerable primitive turns viral: Novel single-trace attacks on ecdsa and rsa," *Cryptology ePrint Archive*, 2020.