Timing Attacks on Cryptographic Algorithms

Timmy Xiao, Yosef Mihretie

May 2022

1 Introduction

Today's modern processors have many innovations that have allowed computers to be faster. Although these changes in computer architecture have resulted in better efficiency and performance, a lot of private data that is processed by these devices can be leaked through hardware components such as branch predictors and caches. These components would be known as side-channels and leaking information through them would be known as a side-channel attack. A recent and notable group of attacks known as Spectre[5] affects many modern processors today due to their use of speculative execution. Side-channel attacks are dangerous because they could render theoretically secure cryptographic algorithms useless through leaked information. In this work, we highlight the threat to cryptographic code from Spectre style attacks and give an overview of defenses that can be deployed in hardware and software.

2 Side-channel Attacks

2.1 Flush+Reload

Modern processors have caches to reduce the latency of obtaining data from memory. In multi-core processors, caches are shared between cores, but this can sometimes, although rare, lead to the cache and memory to be in a inconsistent state[1]. CPU manufacturers solve this problem by making a instruction that just flushes the cache, so the next load instruction must be fetched from memory. However this lead to the ability for adversaries to explicitly flush the cache to perform timing attacks, since loading from a cache line is significantly faster than loading from memory. The attacker would just flush contents of cache, wait for the victim to performance an operation, measure the times to reload lines to see which operations were being done, and then repeat to see the sequence of operations [3]. During a reload of some address, if the adversary gets a shorter access time than usual, then they can conclude that the victim has accessed the same cache block.

2.2 Spectre

Spectre is a group of vulnerabilities that are caused by timing attacks due to processors having the ability to perform branch prediction with speculative execution and caching such results [5]. Speculative execution hides memory latency by performing work ahead of time that might be unneeded. This is usually employed in branch prediction so results would be already be calculated if the correct branch is used and discarded if the wrong branch is used. However, mispredicted speculative execution can result in reading memory in an area that is considered private. This is usually caused by malicious user input that creates an out-of-bounds access in an array that goes to a private piece of data. Although the results are discarded as the branch was mispredicted, the data can still linger in the cache. The processor's cache would now have the private data, making the cache a side-channel, and the attacker can now use timing attacks, similar to flush+reload, on the cache to extract information about the private data.

3 Possible Attacks

3.1 Threat model

We have a trusted cryptographic software running on a machine along with an untrusted program. This captures a wide range of setups ranging from browsers that have cryptographic modules running along a javascript code from a webpage that can not be trusted and cloud service providers running programs from various users on the same machine, some of which might be cryptographic in addition to the cryptographic software provided by the cloud service. The cryptographic software contains keys and derivatives of keys that are critical for its security. The untrusted program can invoke a cryptography method by using one of the API points defined. This could be a request to decrypt, encrypt, verify, sign a message etc. This affords enough flexibility to capture the CCA and CPA security protocols. An adversary that controls an untrusted program would like to leak in part or completely the secret keys or values tainted with the secret keys to aid in cryptanalysis. This can be achieved by using some of the side-channel attacks discussed above.

3.2 Flush+Reload attack on RSA

RSA relies on modular exponentiation which is basically calculating $g^x \bmod n$. To decrypt, one must raise their ciphertext c to their decryption key d modulo n or $c^d \bmod n$. To efficiently calculate c^d, most implementations use a technique called exponentiation by squaring [3]. In observation, we loop through the binary bits of our exponent. We start at $x = 1$, and repeatedly square x to be the new x in each step. In addition, we check if the bit is set; if it is set, we also multiply by the base and set that to be x. When we are setting, we also reduce to do modulo operation. A pseudocode implementation is shown here:

Algorithm 1 Exponentiation by Squaring

Input: base b, exponent e, modulo m
$x \leftarrow 1$
for $i \leftarrow |e| - 1, \ldots, 0$ **do**
    $x \leftarrow x * x$ (Square)
    $x \leftarrow x \bmod m$ (Reduce)
    if e[i] $= 1$ **then**
        $x \leftarrow x * b$ (Multiply)
        $x \leftarrow x \bmod m$ (Reduce)
    end if
end for
return x

Notice for unset bits, we only square and reduce, while for set bits, we square, reduce, multiply by base, and reduce again. If we can determine the sequence of operations, we can reveal the exponent or the decryption key in RSA. Investigating the RSA implementation in GnuPG 1.14.13[2], we would find these operations in the folder `mpi`.

Operation	**Function**	**Location**
square	`mpih_sqr_n`	`mpi-internal.h:240`
reduce	`mpihelp_divrem`	`mpi-internal.h:256`
multiply	`mul_n`	`mpi-mul.c:113`

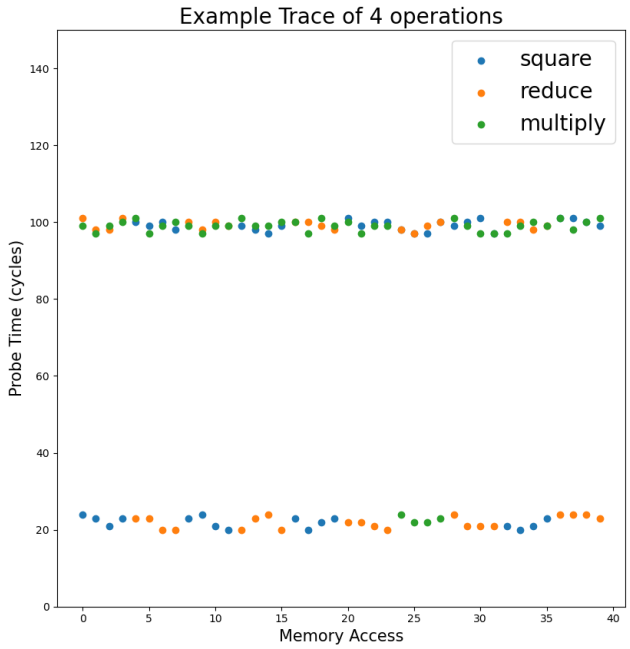

Figure 1: Sample Trace for a flush+reload attack on exponentiation by squaring for 4 operations. One could see the sequence being square+reduce, square+reduce, square+reduce+multiply+reduce, and square+reduce. They can then conclude that the bits here would be 0010.

Now an attacker would share the same pages as the process that is running GnuPG. It would then flush the cache, and then reload each function into the cache and measure the times it took to load them. Shorter times would be seen as the operation being done. By measuring the sequence of operations, the attacker could possibly recover the binary bits of the decryption key, leaking that information through a side-channel.

3.3 Sample Spectre Attack

```
1 static int rsa_ossl_private_decrypt(int flen, const unsigned char *
    from, unsigned char *to, RSA *rsa, int padding)
2 {
3     BIGNUM *f, *ret;
4     f = BN_CTX_get(ctx);
5     num = BN_num_bytes(rsa->n);
6     ...
7     if (flen > num) {
8         ERR_raise(ERR_LIB_RSA, RSA_R_DATA_GREATER_THAN_MOD_LEN);
```

```
9           goto err;
10      }
11      ...
12
13      /* do the decrypt */
14      if (!rsa->meth->rsa_mod_exp(ret, f, rsa, ctx))
15              goto err;
16      }
17      ...
18  }
19
20  ## this is different from what is in the codebase ##
21  static int rsa_ossl_mod_exp(BIGNUM *r0, const BIGNUM *I, RSA *rsa,
        BN_CTX *ctx)
22  {
23      ...
24      some_variable = f(rsa->p);
25      I[f(rsa->p)] = x ;
26      ...
27  }
```

The above code snippets are taken from the openssl implementation of RSA. The last function does exist in the code base, but it is highly complicated and they appear to have made the important defensive techniques which we will discuss in section 5. The code here is a highly modified snippet to help us demonstrate the attack process. It is the same function that indirectly gets called at line 49.

We will first start with Spectre variant 1 [5].

1. Imagine there is a victim program running on the same core as the openssl service program and it requests *rsa_ossl_private_decrypt* service by invoking the corresponding API end.

2. The victim then manipulates a shared cache to make the parameter flen uncached.

3. By influencing the branch prediction mechanism, the attacker makes the program speculate that the if condition at line 38 evaluates to false. This condition doesn't get resolved until flen can be fetched from memory which can be a lot of cycles.

4. Meanwhile, the speculative execution could get to line 44 where a call to *rsa_mod_exp*, which is an indirect call to *rsa_ossl_mod_exp*, is made. This call accesses I, a user provided array, with an index that is tainted with the secret value p.

5. When flen is finally read and the condition resolved, the speculative pipeline will be squashed. This will however leave the cache with the artifacts of the speculative execution.

6. The attacker can now probe the shared cache to recover the secret value p partially or completely. This can be done in a way that is similar to the flush+reload attack above.

4 Defense

In this section, we discuss defensive features that aid in strengthening the security of cryptographic applications at the level of hardware, compiler, operating system, and the applications themselves.

4.1 Hardware

Many of the side channel attacks mentioned above arise due to the performance enhancing features of the hardware. Thus fixing them inevitably involves the hardware and it does so usually at the cost of performance. In this subsection, we discuss a few ideas that are explored elsewhere and speculate about the future.

Isolating cryptographic executions is the first solution we explore. Spectre and Meltdown presume the victim and the attacker share a memory or execution unit through which they can establish a side-channel. If we completely isolate the cryptographic software by running it on a core dedicated to it, this can be avoided. The most extreme form of this is to have cryptographic processing units. This usually also comes with performance benefits. This is already the case in Apple devices which include The Secure Enclave for such processing [6].

The other defensive hardware technique we explore is using more secure caching protocols. Many side-channel attacks assume there is a shared level of cache that is inclusive. They further assume that the adversary program can effect ejection of cache lines in the private cache of the core running the victim program. This is usually done by executing an eviction on the shared, inclusive level of cache. We suggest as a mitigation to Spectre and other similar attacks an improved, security-aware caching protocol. A recent work came up with a protocol called SHARP that makes shared caches induce as little evictions in the private caches of the cores that didn't cause the eviction[**sharp**]. This is done by processing core valid bits. In a shared cache line, core valid bit i is set if core i contains this line in its private cache. When the victim tries to evict a certain cache line by accessing a conflicting address, the replacement policy first tries to evict the lines at the conflicting address that are from the requesting core. This work is a good start in what we think should be given more attention in the future.

Focusing on meltdown, hardware manufacturers like Intel have issued patches to resolve the danger. We can guess these patches target not granting memory accesses speculatively, i.e. until the necessary memory permission checks are performed.

Tagged architecture is an approach to hardware design that strives to enforce security protocols in hardware. Its target is memory-safety: memory locations store metadata about access information along with hardware features that allow processing the metadata to enforce the security protocols with low performance overhead. As mentioned above, memory-safety is essential for the security of any software program. This is specially true for cryptographic applications that routinely process data presented by the user. With respect to

spectre, spectre variant 1 relies on being able to guess a memory location that is mapped in the address space of the cryptographic program that it can leverage as a side-channel after getting it cache. Having a memory safety vulnerability will make it easy to find such an address and escalate a spectre vulnerability into a full chain exploit. Thus, ensuring memory-safety in hardware while preserving performance would be a great feature to have. We are not aware of a complete product of such a processor, but we came across interesting instruction set extensions for tagged architecture that we thought were interesting [**comet**]. The architecural memory protection can also be implemented partially. That is the program can annotate the cryptographic keys, the compiler can figure out the associated data that needs to be protected and use the tagged memory locations for these. This is an interesting area of research for the future.

4.2 Compiler

Compilers modify code in the process of generating binaries to improve performance and security. They do this by performing static and dynamic analysis on an intermediate representation of the code. Thus they have a potential in improving the security of cryptographic code is specifically crucial. By reordering or inserting new code, they can help prevent side-channel attacks. In this section, we first discuss some general guidelines for how compilers could be utilized in ensuring the security of cryptographic code. We will then provide a review of recent work in the literature on the topic.

Our ideal compiler would allow the programmer to annotate variables that hold security critical data. This is primarily keys and other values tainted with keys. Once this is done, the compiler would reorder and insert new code to make it hard to carry out a side-channel attack. For instance, spectre variant 1 relies on the target value being used to index into an array. This has to follow a branch statement and is executed speculatively. The access to the array, which is cached, is the side-channel. If the key is annotated, the compiler can prevent this chain by for instance making sure blocks that follow a branch statement with a predicate that is influence by user input and that use derivatives of keys as indices into arrays are not executed speculatively. Alternatively, it can reorder the code so that even if a block is executed speculatively, it is unlikely to reach to the line where the key is used before it can get squashed. It can do so by reordering code to move the key access as far from the block entry as possible. The compiler gets to do decide where variables get stored in memory, when to fetch them to register and when to spill them. This is crucial for security. We recommend compilers keep keys in registers as much as possible. Additionally, it is important to make sure keys and their derivatives do not share a cache line with data that can possibly be controlled by the user. For instance, in the openssl program the *rsa* struct contains both private and public keys of the *rsa* protocol. If its memory layout is not idea enough to have a public key share a cache line as a private key, then the private keys will be loaded to memory while let's say the adversary invokes a verify function. Knowing the public keys, the adversary is further assisted in probing the cache to find the private keys.

Therefore compilers could pay series attention about what data shares cache lines with the private keys. We also suggest keeping as much distance between buffers and keys in memory. Moreover, we suggest the memory layout should be randomized. Compilers can also take the responsibility of erasing artifacts of secure keys from the cache after a cryptographic function that used them was invoked.

Since the spectre paper was published, researchers have looked into using compilers to defend against them. One of the first defenses to come out was that by Microsoft Visual's C/C++ compiler. Their approach is to insert $lfence$ instruction before vulnerable blocks. These prevent speculative execution. Nonetheless, their approach for detecting vulnerable blocks was extremely ineffective. One test found that they can detect about 2 in 15 cases. Another interesting work is the Speculative Load hardening mitigation. This technique inserts hardening instructions that zero out the pointers with data dependency with branch conditions. This mitigation addresses only variant 1 of Spectre. Moreover, its overhead has been reported to be as high as 36%. One interesting work we found does so by using static analysis to locate the blocks that follow a branch and could leak a key and using the $fences$ to prevent their speculative execution. An interesting work we came across that aligns with our ideas is BLADE [**blade**]. BLADE is a compiler solution that does some of the things we suggested above in a rigorous manner. BLADE allows programs to annotate variables that are critical for security, it then performs static analysis possible leaks. BLADE prevents this leaking in speculative execution by cutting the data flow from the protected variable to the leaking statement until the branch his resolved. Note that this does not require stopping speculative execution, thus it is a performance solution. Moreover, it analyzes the programmer specified protect variables using a max-flow algorithm to determine if the same security can be achieved by a smaller set of protect statements.

4.3 Operating System

The operating system also is a critical part of security. The operating system is responsible for enforcing memory isolation of processes. Memory safety mitigation features like Address Space Layout Randomization are also usually done at the operating system level. The OS is also generally responsible for pseudo-random number generation, which is an essential component of cryptographic protocols. These are all critical pieces of security and should be ensured to the best degree possible. In this section however, we focus on how operating systems can help mitigate dangers from Spectre.

The first important feature we suggest is security-aware scheduling. In the model we laid out earlier, the victim program (a cryptographic software in this case) needs to share a level of cache with the attacking program for a spectre style attack. The operating system can mitigate this by scheduling cryptographic programs on cores that don't share a cache level with an untrusted program. Besides sharing cache levels, influencing the branch prediction mechanism of the victim program is an important part of spectre. Operating systems

should take all of this while scheduling programs. We suggest a privilege based approach where an untrusted code is run in the lowest privilege possible and cryptographic code higher in the hierarchy as much as possible. When scheduling, a lower privilege program should not be allowed to share resources with a higher privilege program.

Finally we discuss Address Space Layout Randomization. Address Space Layout Randomization is an important component in preventing memory corruptions from leading into full exploits. Besides that however, it also plays a part in preventing side-channel attacks. Many of the side-channel attacks we talked about rely on the adversary knowing the address of a memory location that is to be used as a channel through the cache. This is the array in Listing 1. If ASLR is implemented sufficiently well, then the attacker can not guess where these arrays are and thus can not monitor the cache efficiently.

4.4 Application software

In this section, we discuss how programmers can write cryptographic code that is not easy to exploit using side-channel attacks. The first obvious point is to have solid memory safety. Memory corruption unfortunately can not be completely eliminated. We suggest however using relatively memory-safe languages like Rust and Go. We also suggest using machine verification to check memory safety. Thorough testing using fuzzing and symbolic execution tools is also important. Lastly, we recommend deploying all mitigation techniques including CFI, ASLR and Stack protection as much as possible.

One way for software to mitigate side-channel attacks is to use a technique called blinding. Blinding is a technique used where the application computes a function for an encoded input without knowing the real input or the real output. The client who gives the encoded input would get an encoded output back that the client should be able to decode. In the case of RSA, we would be blinding the exponent [4]. When we encrypt, we would add to make our new exponent be $e + r\phi(N)$, where e would be the encryption key, r a new randomly generated value, and N is our modulus. To decrypt, one would calculate the modular inverse of $e + r\phi(N)$ to decrypt as normal. This would make the square and multiply operations uncorrelated with the original input. However, this also implies that the blinding component must also be secure and safe from side-channels.

As noted above, spectre relies on an array that is cached which the victim has to index into using a derivative of a target value. It also requires a victim program to have a branch condition that is influenced by user input. We suggest the following coding practices to reduce the risk of exploitation from these attacks. First, never use a value tainted with the secret key to index to an array the address of which the attacker could know. If there is a cipher-text to be decrypted which can come from an adversary, make a copy of it before working on it. Second, if possible at all don't use key tainted values immediately after branches. If you have to, put the key tainted value access as far from the entry of the block as possible. This reduces the chance a speculative execution would

have operated on the tainted value before it is rolled back. Third never use a user input or user input tainted value as a predicate in a branch, specifically if just after the branch there are operations on the key. Finally, use machine-checking to guarantee the memory safety, correctness and absence of timing vulnerabilities in the cryptographic code. Tools like jasmin can assist in this [**jasmin**].

5 Future work

A comprehensive compiler strategy to quantify the risk from side-channel attacks as well as the relation between this risk and performance, allowing the programmer to set a desired level of protection. Taking this to the extreme, we envision a cryptographic domain specific language with an inbuilt verification system to verify correctness, memory-safety and side-channel attack protection as well as many of the compiler features we recommended above. Future research should also explore hardware level defenses like better cache protocols and memory tagging. Additionally, operating system level defenses like security-aware scheduling appear to be a promising avenue.

References

[1] Daniel G. Bobrow et al. "TENEX, a Paged Time Sharing System for the PDP - 10". In: *Commun. ACM* 15.3 (Mar. 1972), pp. 135–143. ISSN: 0001-0782. DOI: 10.1145/361268.361271. URL: https://doi.org/10.1145/361268.361271.

[2] Werner Koch. *[Announce] GnuPG 1.4.13 released*. 2012.

[3] Yuval Yarom and Katrina Falkner. *Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*. 2013.

[4] Jake Edge. "Breaking Libgcrypt RSA via a side channel". In: (2017).

[5] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[6] "The Secure Enclave, Apple Support". In: (2021).