RASSLE: Return Address Stack based Side-channel LEakage

RASSLE: Return Address Stack based Side-channel LEakage

Anirban Chakraborty[1], Sarani Bhattacharya[2], Manaar Alam[1], Sikhar Patranabis[3] and Debdeep Mukhopadhyay[1]

[1] Indian Institute of Technology Kharagpur, India,
{anirban.chakraborty,alam.manaar}@iitkgp.ac.in, debdeep@cse.iitkgp.ac.in
[2] Katholieke Universiteit Leuven, Belgium, sarani.bhattacharya@esat.kuleuven.be
[3] ETH Zürich, Switzerland, sikhar.patranabis@inf.ethz.ch

Abstract. Microarchitectural attacks on computing systems often stem from simple artefacts in the underlying architecture. In this paper, we focus on the Return Address Stack (RAS), a small hardware stack present in modern processors to reduce the branch miss penalty by storing the return addresses of each function call. The RAS is useful to handle specifically the branch predictions for the `RET` instructions which are not accurately predicted by the typical branch prediction units. In particular, we envisage a spy process who crafts an overflow condition in the RAS by filling it with arbitrary return addresses, and wrestles with a concurrent process to establish a timing side channel between them. We call this attack principle, RASSLE [1] (Return Address Stack based Side-channel Leakage), which an adversary can launch on modern processors by first reverse engineering the RAS using a generic methodology exploiting the established timing channel. Subsequently, we show three concrete attack scenarios: i) How a spy can establish a covert channel with another co-residing process? ii) How RASSLE can be utilized to determine the secret key of the P-384 curves in OpenSSL (v1.1.1 library)? iii) How an Elliptic Curve Digital Signature Algorithm (ECDSA) secret key on P-256 curve of OpenSSL can be revealed using Lattice Attack on partially leaked nonces with the aid of RASSLE? In this attack, we show that the OpenSSL implementation of scalar multiplication on this curve has varying number of *add*-and-*sub* function calls, which depends on the secret scalar bits. We demonstrate through several experiments that the number of *add*-and-*sub* function calls can be used to template the secret bit, which can be picked up by the spy using the principles of RASSLE. Finally, we demonstrate a full end-to-end attack on OpenSSL ECDSA using curve parameters of curve P-256. In this part of our experiments with RASSLE, we abuse the *deadline scheduler* policy to attain perfect synchronization between the spy and victim, without any aid of induced synchronization from the victim code. This synchronization and timing leakage through RASSLE is sufficient to retrieve the Most Significant Bits (MSB) of the ephemeral nonces used while signature generation, from which we subsequently retrieve the secret signing key of the sender applying the Hidden Number Problem.

Keywords: Return Address Stack · Microarchitectural Attack · Template Matching · OpenSSL ECC scalar multiplication · ECDSA P-256 · Lattice Reduction.

1 Introduction

The evolution of computer architecture has taken place through several inventions of sophisticated and ingenious techniques, like out-of-order execution, caching mechanism,

[1] RASSLE is a non-standard spelling for wrestle.

branch-prediction, speculative execution, and a host of other optimizations to maximize throughput and enhance performance. While it is imperative to imbibe and develop these artifacts in our modern-day machines, it is equally necessary to understand the security threats posed by these mechanisms, particularly on the execution of cryptographic programs operating on sensitive data. As the foremost criteria of these architectural optimizations have been performance, they need a closer investigation from the security point of view. With the growing impetus for security in applications where modern computing finds usages, a multitude of microarchitectural attacks have been unearthed by security researchers that exploit information leakage due to the functioning of these artifacts. These attacks have been shown to threaten the secret keys of cryptographic algorithms when run on these platforms, a risk that puts to threat several critical operations where these machines are expected to operate.

Some of the oldest attacks targeted the cache memories [Ber05, Per05, AKS07a, Aci07, BM06] where they tried to exploit the timing side-channel [Koc96] produced due to the presence of cache between very fast processor and a comparatively slow memory. More recent cache-based attacks can be categorized into three generalized techniques - Evict+Time [OST06], Prime+Probe [OST06], and Flush+Reload [YF14]. A number of variants of these attacks have been proposed over the years, which include defeating countermeasures like kernel address space-layout randomization (KASLR) [HWH13, CSH+20, DKPT17], attacks on cryptographic protocols such as RSA [Per05, IGA+15], AES [AKS07a, GII+15, IES15], ECDSA [YB14, BH09], ECDH [GVY17], setting up covert channels on the cloud [RTSS09, ZJOR11], attacks exploiting performance degradation [ABF+16], reading user inputs [GSM15, GMWM16, Hor16], etc. Another important microarchitectural component that has been the target of a multitude of attacks is the branch prediction unit (BPU). Ciphers like RSA, which contain conditional branches in their implementation, have been targeted by exploiting the BPU [AKS07b, ERAGP18, BM15]. Similarly, breaking KASLR [EPAG16] and setting up a covert channel have also been demonstrated using BPUs. More recent attacks like Spectre [KHF+19] and Meltdown [LSG+18] exploit the speculative execution subsystem and out-of-order engine present in the CPU. Another class of attacks called Microarchitectural Data Sampling (MDS) (e.g. [VSMÖ+19, MML+19, SLM+19]) surfaced more recently, which leak data across protection boundaries.

The Spectre attack exploits the speculative behaviour of modern processors to expose secret information that is otherwise inaccessible to the attacker. *SpectreRSB* [KKSAG18] proposed a new variant of Spectre attack where the authors exploited the Return Address Stack (RAS) to cause speculative execution of payload gadgets and expose sensitive information. They showed multiple variants of their attack, where they manipulate the RAS in order to create a mismatch of the return addresses. They further implemented their attack on the Intel SGX [CD16] platform, where a malicious operating system corrupts the RAS to expose data out of the enclave by forcing mis-speculation. Concurrently [MR18] proposed another attack, named *ret2spec*, abusing the RAS similarly to read user-level memory that should be inaccessible in the normal course. They proposed two variants of their attack - one where the attacker program poisons the RAS to coerce a co-located process to execute arbitrary code speculatively and another, where they leverage JIT (just-in-time) environment to read arbitrary memory in modern browsers. In a more general sense, both the papers [KKSAG18, MR18] try to leverage the fact that the RAS is a part of the speculative execution process and manipulate or poison the RAS in order to create mis-speculation.

In this paper, we propose a novel attack mechanism exploiting the Return Address Stack (RAS) - a hardware entity in modern machines meant to improve the prediction for `RET` instructions. We show how a spy can reverse-engineer the RAS to develop suitable attack vectors. In particular, we show how a covert channel can be established by exploiting the reverse-engineered knowledge of the RAS. Secondly, we show demonstrate one can

perform a timing side-channel attack by targeting the scalar multiplication on the P-384 curve in OpenSSL(v1.1.1g library). Finally, we extract the secret key from ECDSA on P-256 of OpenSSL using lattice reduction on partially leaked nonces. Collectively, we call the idea of the RAS based attack, as RASSLE, where the spy wrestles with a target code to launch effective timing attacks.

1.1 Motivation

RAS presents a side-channel artifact that has been comparatively less explored. It can fall into a class of branch prediction attacks, which can be effective even when a code is devoid of explicit loops. This is because it is related to function calls and returns from functions. There has been no prior work that established timing channels exploiting the RAS, nor has shown the effective covert channels utilizing its presence in the computing system.

Moreover, due to the now well-known mitigating techniques against Spectre attacks, the reported mis-speculation attacks using RAS can be thwarted in modern processors [Tur18, KKSAG18, MR18]. This fact motivated us to look for other avenues through which the RAS can be exploited. As RAS has a fixed size, the number of return addresses of functions that can be stored in it is limited by its capacity. Now, in case of overflow or underflow conditions (explained later in Section 2), mispredictions can occur. These mispredictions lead to a timing penalty, revealing information of various categories - covert channels, secret keys, etc. It is worth mentioning that although SpectreRSB [KKSAG18] proposes a similar attack setup where two colluding threads operate in synchronization; the paper focuses on creating mispredictions and does not establish the evident covert channel that exists due to sharing of the RAS.

Finally, despite the common wisdom of writing cryptographic codes ensuring the independence of secret key, the paper mentions an exploit in the popular OpenSSL library on a standard elliptic curve, where the number of function calls (*add*-and-*sub*) is dependent on the secret key bits. The paper makes an effort to develop a template attack for determining the secret key by timing observations by a spy exploiting the RAS.

1.2 Our Contribution

In this work, we propose a new side-channel leakage source through RAS called RASSLE which, unlike existing works, does not mistrain the RAS or abuse the speculative execution engine. Rather, we utilize the timing channel created due to overfilled RAS, which eventually causes misprediction of the return address and leaks information about the control flow of a co-located process. Our main contributions are as follows:

- We *reverse engineer the depth of RAS* using timing information and provide a generic methodology that can be used for any processor.
- We particularly exploit the fact that overflowing RAS can trigger misprediction, ultimately creating a timing side-channel. Based on the timing channel identified, we demonstrate how a *covert channel can be established between a spy and a co-residing program* by exploiting the RAS.
- We show an *exploit in the ECC scalar multiplication of P-384 in OpenSSL*, wherein the number of function calls is dependent on secret key bits. We show a template-based attack which the RASSLE spy can launch to deduce the secret key bits.
- We show how RASSLE can be adapted to a completely asynchronous running spy and victim, where the spy could manage to achieve one to one correspondence to the victim execution by tweaking the *deadline scheduler* policy from the user level. We develop this into *a key-recovery attack on OpenSSL's ECDSA signature scheme (P-256 curve-based)*. In particular, we adopt well-known lattice-based techniques to recover the ECDSA signing key from ephemeral nonces that are partially reconstructed using RASSLE.

1.3 Responsible Disclosure

While we found this efficiently working covert channel and timing vulnerability of OpenSSL P-384 and P-256 implementation in context to the shared Return Address Stacks, we have taken this opportunity to contact the security team of Intel and OpenSSL regarding the said vulnerability. We hope that by making this communication regarding this specific architectural design vulnerability to the security designers of processor manufacturing companies would lead to further collaborations in solving the problem together.

1.4 Organisation

The rest of the paper is organized as follows. Section 2 briefly discuses the working principle of RAS. Section 3 shows how to reverse engineers the RAS and introduces a novel timing side channel created thereof, based on which Section 4 proposes a proof-of-concept attack scenario. Section 5 shows how RASSLE can be utilized to perform template attack on P-384 curve in OpenSSL. Section 6 shows how to get *asynchronous* spy and victim processes to work using RASSLE, and subsequently demonstrates an attack on the ECDSA signature generation algorithm. Section 7 discusses potential countermeasures against our proposed attacks. Finally, Section 8 concludes the paper.

Additional material presented for completeness includes details of the scalar multiplication implementation in OpenSSL (Appendix A), background on ECDSA (Appendix B) and cryptanalysis of ECDSA using lattice techniques (Appendix C).

2 Return Address Stack (RAS)

In this section, we start with a general understanding of return addresses and how frequently used return addresses are stored in a specialized micro-architectural hardware component called RAS. Return instructions are a special type of indirect branch instructions that might get called from different program locations, but the target address will remain the same. For example, the `printf()` function in GNU-C library can be called from different functions of a program. Every time the same library subroutine will be invoked, and the same set of instructions will be executed. However, since they are called from different program locations, the return addresses for each corresponding function call will be different.

Now considering a speculative execution environment, for each return statement, the processor must predict the return address (usually the next instruction after the call statement), which the program counter (PC) must contain. Generally, the Branch Target Buffer (BTB) stores a mapping between the branch source and the target address for other indirect branches. But in the case of return instructions, the BTB will often mispredict since the same function can be called from different locations and thus have different return addresses. Thus to improve the prediction rate, modern processors deploy hardware-based stack call *Return Address Buffer*, or equivalently, RAS.

Return addresses are generally stored in the software stack, which is maintained in the main memory. In order to keep the return addresses closer to the processor, processor engineers incorporated hardware-based RAS, which essentially alleviates the high latency experienced when the return address is fetched from the main memory. The processor on encountering a call statement pushes the return address (the address of the next instruction in program order) to the RAS, and when a corresponding matching return statement is encountered, the processor speculatively executes the instruction stored at the address pointed by the top of RAS. At a later stage in the pipeline, the processor matches the return address with the one stored in the software stack. If both the addresses match, the execution is continued; else, discarded. Given that the return addresses are stored on the stack, the processor refers to the RAS every time it encounters a return statement.

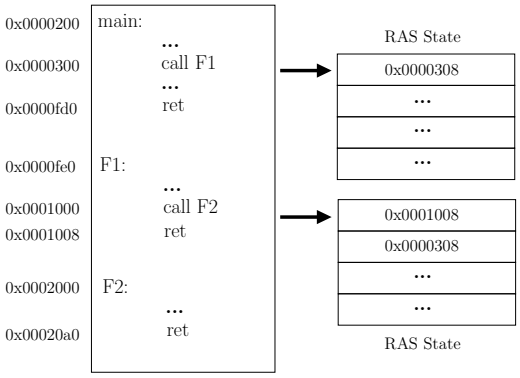


Figure 1: Example of function call and its effect on RAS.

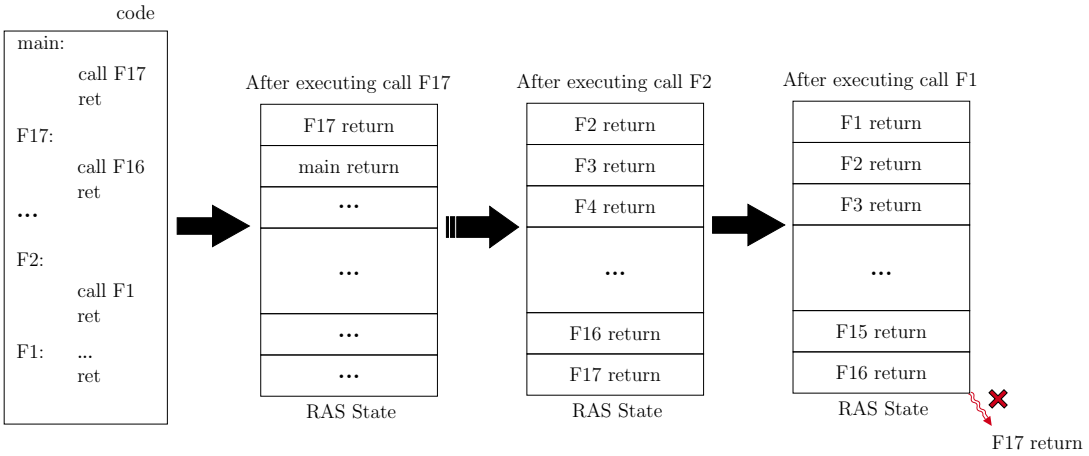


Figure 2: Reverse Engineering Return Address Stack.

Without such a hardware stack, the processor would have to spend multiple clock cycles to fetch the return address from main memory.

2.1 Working Principle

RAS is generally implemented as circular LIFO (Last In, First Out) stacks, which can store a fixed number of entries of return addresses. When a function call is executed with the stack already at its full capacity, the new entry (return address) is inserted at the top of the stack, and the oldest entry (from the bottom of the stack) is discarded. This is known as *overflow* condition. Similarly, if a series of overflow conditions drive out the valid return addresses from the stack, it causes an *underflow* condition where there are no entries available in the stack [SAMC98]. An example of the working principle of RAS is shown in Fig. 1. Consider a program consisting of main function and two function calls *F1* and *F2*. Corresponding addresses of the instructions in memory are also depicted beside the instructions in Fig. 1. When the function *F1* is called, the address for its next instruction is pushed onto the top of the stack. Inside *F1*, another function *F2* is getting called which again pushes its return address onto the top of the stack. Now, whenever a return statement is encountered, the address from the top of the stack is popped and is used as a reference for the next instruction to be fetched in the pipeline.

3 Delving into processor specific RAS reverse engineering

In this section, we develop a generic methodology to reverse engineer the capacity of Return Address Stack. The number of entries that RAS can accommodate varies for different families of processors. In case of deeply nested calls (for example, recursions), the return addresses are pushed onto the stack for each call statement. If the stack was already full, the oldest entry is discarded from the bottom, and the newest entry is added to the top of the stack. Therefore, during subsequent returns, the addresses present in the stack will be fetched quickly from this specialized hardware, thereby aiding the performance. Whereas, the return addresses, which were pushed off the stack due to overflow condition, becomes unavailable. This situation forces the processor to stall for multiple clock cycles in order to decode the actual return address from the software stack (stored in main memory and subsequently in the cache memory hierarchy), which incurs a significant latency and results in a considerable increase in the execution time. The fact that an overflow condition in RAS leads to an increase in execution time helps in reverse-engineering the length of the stack in different processors.

3.1 Observing Timing Differences over Recursive Function Calls

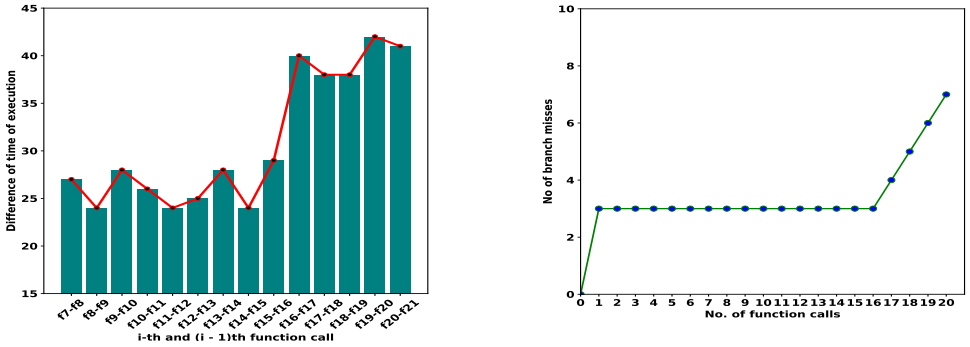
In our experimental setup, we show the results on Intel i7 7700 running Ubuntu 16.04; however, the same experiment can also be conducted on other architectures. The typical size of Return Address Stack in contemporary processors are 16, 32, and 64. Fig. 2 shows a pictorial representation of the reverse engineering approach. The idea is to execute several function calls in a nested fashion such that each function call pushes a return address onto the stack and simultaneously checking the execution time after each function performs a return. The steps may be summarized as follows:

- We start with an arbitrary number of nested function calls (without loss of generality, say 18) and check the difference in execution time for 18 calls and 17 calls separately. This provides us an estimate of the timing for the outermost function (function 18 in this case). We run the experiment for $100,000$ times (in order to ensure consistency of the measurements) and calculate the *mean difference in execution time* for 18^{th} function and 17^{th} function. The difference turns out to be *40.52* clock cycles (Fig. 3a).
- We repeat the same experiment for 17 nested calls and check the mean difference at depth 17^{th} and 16^{th} function call. The difference in execution time was close to the earlier one, *44.26* clock cycles (Fig. 3a).
- Next, we reduce the function depth by 1 and repeat the experiment. The difference in timing for depth 16^{th} and 15^{th} call was observed to be *21.43* clock cycles (Fig. 3a), which is considerably less than the previous two values.

As shown in Fig. 2, the reason for the decrease in execution time is that when the depth of nested function calls is 16 or less, the processor gets all the return address from the RAS and therefore takes considerably less time to complete execution. This signifies that on our target system, the RAS can hold up to 16 entries. However, if the depth of the nested calls is more than 16, the oldest return address will be written off (removed) from the stack. Therefore when a corresponding return statement is executed, the processor will have to stall for multiple cycles to fetch the return address from the software stack stored in the main memory.

3.2 Validating Results using Hardware Performance Counters

In the previous sub-section, we reverse-engineered the size of the RAS using timing information. Although timing information is based on the number of CPU cycles elapsed



(a) Timing difference (in clock cycles) for i^{th} and $(i-1)^{\text{th}}$ function calls

(b) Number of branch misses w.r.t depth of function calls

Figure 3: Variation of (a) timing and (b) branch misses shows the capacity of RAS.

during the execution of a particular process, the values can sometimes be influenced or disturbed by the interaction among other processes and components of micro-architecture. Therefore, we validate the results obtained by timing information using information from Hardware Performance Counters (HPCs).

HPCs are a set of special-purpose registers present in modern microprocessors. These counters come in handy for monitoring and measuring various hardware and software events during process execution. Although initially designed as a profiling tool to optimize programs and applications, HPCs have been recently used for security purposes, both in offensive and defensive ways. In this paper, we utilize the `perf-event` profiling tool to verify the results obtained in the previous sub-section. It must be noted that `perf-event` can only be accessed from administrative privilege. *Therefore, we use HPC only for validation purpose and not for the attack.*

In a speculative execution environment, the target address for a *return* instruction is predicted by referring to RAS and matched with the actual value stored in the main memory much later in the pipeline. Therefore, any wrongly predicted address or an underflow/overflow condition in the RAS will result in a branch miss event. Branch misses can be pretty accurately measured for a process using the `perf-event` tool in Linux. The event `PERF_COUNT_HW_BRANCH_MISSES` records the number of branch misses under the `PERF_TYPE_HARDWARE` type event. We use `ioctl` calls to query the performance counters. Incidentally, we found out that for the inner 16 functions, the number of branch misses is 2 while for the 17^{th} function, the number of branch misses becomes 3 and further increases by 1 for each unit increment in function depth. Fig 3b shows how the change in depth of nested function calls affects the branch miss statistics due to the presence of RAS. This verifies our claim that on our setup, RAS can hold up to 16 entries. Additionally, this reverse engineering method can be easily scaled up and generalized for any target setup.

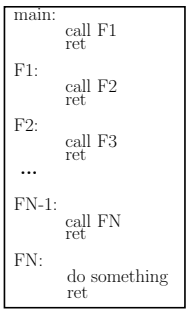
4 RASSLE: Establishing a covert channel through RAS

In this section, we exploit the fact that an overflowing RAS can result in an increase in execution time, and this difference in timing can be observed by a co-located process to establish a covert channel between two processes.

4.1 Contention-based Timing Channel over RAS

Although RAS is designed to enhance the overall performance of the processor, the security implication of such a capacity-bound hardware stack has not been explored in depth before. We explore and analyze the timing channel created through RAS and its implication on co-located processes for the first time. We present RASSLE - which literally means to wrestle or scuffle - where an adversary wrestles with a co-located victim process to occupy the RAS and, in the course, learns valuable information regarding the control flow of the victim process. For example:

- Consider two processes - Process A and Process B - running simultaneously on the same logical core on a system.
- Process A executes a series of N nested function calls (shown in the adjoining figure), where function *F1* calls *F2*, which in turn calls *F3* and so on. Therefore, for each function call, an entry is inserted to the top of the stack. Choose N such that the entire stack gets filled with the return addresses of the corresponding function calls.



Now, as the innermost function *FN* executes the return instruction, the topmost entry gets popped out, and the stack will contain the remaining $(N-1)$ entries. In order to avoid that and keep the stack occupied, we voluntarily let the process A cede the control of the CPU (yield) just before executing the return statement inside the innermost function (*FN*). The situation becomes interesting from the security point of view when we consider process B running simultaneously on the same CPU core. Consider the following sequence of events - ❶ Once again, process A executes N (large enough to fill the RAS) nested function calls. Inside the innermost function, it yields the CPU before executing the return instruction. Therefore, the entire RAS is filled with return addresses of process A with the return address for the innermost function at the top of the stack. ❷ In this scenario, if we allow process B to execute some function, or in more general terms, it executes M (could be as small as 1) functions in a nested fashion. As both processes A and B share the same RAS, the return address from process B will be pushed onto the stack, whereas $N-M$ number of return addresses for process A will be pushed out from the end of the stack [2]. This results in an underflow condition where some valid return addresses have been pushed out of the stack. ❸ As the control of the CPU returns to process A, it executes all the return instructions and measures the time elapsed for the execution of each function. Due to the introduction of M return addresses by process B, process A will experience a considerable increase in execution time between $(N-M)^{\text{th}}$ and $(N-M+1)^{\text{th}}$ functions and also for all the subsequent functions.

Therefore, paired up with the timing observation experiment in the earlier section, RAS can be exploited by Process A to understand the control flow of Process B. This timing difference could also be converted to a very reliable covert channel between these two processes. Moreover, this seemingly simple timing side-channel can have bigger security implications in the context of security-critical applications. To better understand the security threats posed by RAS, we develop a threat model and introduce our proof-of-concept implementation in the following subsections.

4.2 Assumptions on the Threat Model

We consider a multi-user environment running on a Linux-based operating system, where multiple concurrent processes are sharing the hardware. The adversary does not need any special privileges. However, it does have a requirement to relinquish the CPU after filling

[2] As the RAS is implemented as a FIFO circular queue, the entry of new address will push the last address out of the stack.

the RAS to its full capacity. This could be achieved by executing system calls such as `sched_yield` [3] and acquire CPU cycles using `rdtscp` [4] instruction. Both the instructions can be executed from user-level privilege. This exploit requires that the victim and the adversary to operate on the same logical core. This can be achieved using `taskset`, which is a common assumption in similar attack settings [KGGY20].

4.3 Establishing the Covert Channel

We present our proof-of-concept (POC) implementation using RASSLE on a toy example. Similar to the discussion in the earlier subsection, here we consider two processes and name them as *transmitter* and *receiver*. Consider the sample program of the *transmitter* process as given in Listing. 1. In this program, the *transmitter* reads from a string of binary characters ('0' or '1') in a loop and based on the value of the character processed, either executes a function (*func*) or does nothing. Also it is worth mentioning that we deliberately force the *transmitter* process to yield the CPU at the end of each loop such that the adversary process can take control and run its exploits. This is done in order to better synchronise the *transmitter* and the *receiver*.

```
char r[] = "11010101010000001110001111";
for (int i = 0; i < sizeof(r)/sizeof(r[0]); i++) {
      char c = *(r + i);
      if (c == '1')
              call func()
      sched_yield();
}
```

Listing 1: Sample transmitter program

In our POC, the covert channel is established as follows:

- ***Receiver* process:** calls N nested functions in a loop to fill up the entire RAS [5]. Inside the deepest function (i.e., 16^{th} function in our case), the *receiver* yields the CPU to the *transmitter* just before executing any return statement.
- ***Transmitter* process:** running on the same processor core as that of the *receiver*, takes control of the CPU. Depending on the value of the bit processed, the *transmitter* will either call another function, or does nothing.
  - Suppose, the bit processed is '1'; the *transmitter* program will then call the function *func* and as a result, the return address, i.e., the address of the subsequent instruction after the call statement will be pushed onto the stack. Since the stack was already full, the entry of new address will push the oldest entry out of the stack.
  - While on processing a '0', there is no impact on the RAS state. The RAS state remains unaltered as the *receiver* has left it.
- ***Receiver* process:** At this point, the control of the CPU is moved to the *receiver* which will start executing the un-finished return commands by referencing the addresses stored in the stack. Starting from the deepest function (16^{th} function), all the return statements will be executed by referring to the addresses stored in the stack. Timing latency in the outer-most function makes the covert channel complete.

[3] Note that `sched_yield` is a system function in Linux, which causes the calling thread to relinquish the CPU.
[4] The `rdtscp` instruction reads the current value of the processor's time-stamp counter.
[5] The machine we did our experiment on had RAS of capacity 16.

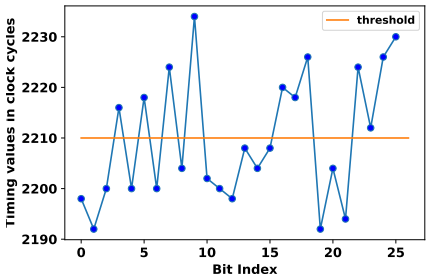


Figure 4: Timing values observed through RASSLE reveals the information transmitted. The values above the threshold indicate a '1' and values below the threshold indicate '0'. Thus it reads "000101010100000001110001111". Apart from the first two bits, the rest of the character stream can be constructed without any error.

Table 1: Comparison of POC covert channel on various systems

Sl. No.	CPU model	CPU family	RAS size	Operating System	Average Bandwidth	Comments
Machine 1	Intel i7 - 7700	Kaby Lake	16	Ubuntu 18.04	30 KBps	Uses sched_yield()
Machine 2	Intel Xeon E5-2609 v4	Broadwell	16	RHEL 7.7	12 KBps	No sched_yield() or sleep. Relies on *deadline scheduler* (discussed in Section 6)
Machine 3	Intel i5-4570	Haswell	16	Ubuntu 18.04	25 KBps	Uses nanosleep()
Machine 4	AMD Ryzen 3500U	Zen+	32	Mint OS 20	25 KBps	Uses sched_yield()

Receiving a '1': CPU while executing the final return statement will encounter a stack underflow situation and thereby require comparatively more time to complete the process. This difference in timing due to expulsion of return address from the stack will be the basis to determine whether the *transmitter* processed a '0' or a '1'.

Receiving a '0': This is inferred when no extra latency can be observed.

Fig 4 shows the distribution of timing values observed by the receiver. The threshold is empirically selected and the timing values above the threshold denote a bit '1' and below the threshold denote a bit '0'. In our experiments, the information transmitted through the covert channel had an accuracy of 75% to 85%.

In the POC just described, we assumed that the *transmitter* yields the CPU at the end of each iteration. However, this is a very restricted model because, in practice, it will be highly unlikely that an unaware *transmitter* process will yield the CPU after every iteration. Fortunately, in our experiments, we found out that using `sleep` has similar effect as `sched_yield`. We replaced `sched_yield` with `nanosleep` and put some delay (in nanoseconds) at the end of every iteration. Similar to the attack proposed earlier, the *transmitter* processes a character stream of 0s and 1s in a loop and if the bit is '1', it calls the function *func*. However, at the end of each iteration it goes into sleep for a short predefined duration. When the process goes into sleeping state, the control of the CPU moves to the other waiting processes - the *receiver* process in this case. Table 1 quotes the RAS size and bandwidth achieved by the POC and also the constraints on a few interesting setups where we made our POC to work.

5 Case Study on OpenSSL ECC Scalar Multiplication

In the previous section, we demonstrated how RASSLE can be utilized to leak information about the control flow of another process. The POC we proposed is a generic attack

technique which exploits the timing side channel to precisely extract the secret information hidden within the victim program. However, such naïve implementations are rarely seen in modern day security-critical cryptographic libraries. Therefore, to demonstrate RASSLE on a real-world setting, we target the scalar multiplication operation in NIST P-384 curve from OpenSSL. This curve has been specified by the NSA Suite B Cryptography as the recommended elliptic curve to be used in ECDSA and ECDH algorithms [Nat15, X9.03].

5.1 Elliptic Curve Cryptography and OpenSSL

Elliptic Curve Cryptography (ECC) is one of the most widely used asymmetric key algorithms based on the algebraic properties of elliptic curves over finite fields. Point multiplication or scalar multiplication is the fundamental and security-critical operation in ECC which computes $Q = [k]P$, where k is an n-bit secret scalar and Q and P are points on the elliptic curve. The security of ECC is defined by the intractability of determining the scalar k given both the points and the curve parameters. The scalar multiplication in OpenSSL is implemented using Montgomery ladder with conditional swaps and Non-Adjacent Form (wNAF) for scalar representation. The scalar k is transformed to its corresponding wNAF representation and based on this representation, a series of *double* and *add* operations are executed to perform the multiplication. These operations are further implemented by a series of `BN_add` and `BN_sub` functions (more details in Appendix A). The aggregate number of such calls are not constant, and are dependent on both the value of secret scalar and the input basepoint. Therefore, the number of times the `BN_add` and `BN_sub` functions are executed depend on the affine coordinates of the EC point, as well as the value of the secret scalar. OpenSSL also provides a support for point randomization, where the EC points are randomized in every call to the ECC function. However, in this particular work, we have demonstrated a chosen plaintext attack with the basic timing observations which demands full control on selecting the input points to the ECC operation. Thus we assume that such countermeasures have been disabled.

5.2 Timing Variation Observed through RASSLE

Similar to the lines of our POC described in Section 4, we set up the spy process to monitor the timing values. By using the reverse engineering methodology described in Section 3, the spy determines the capacity (say, n) of the RAS in the target system. Next the spy, attached to a specific CPU core, continuously executes n nested function calls in a repeated manner. Inside the n^{th} function, i.e., the innermost function, the spy deliberately yields the CPU before executing the return statement. In other words, the spy fills up the entire RAS with the return addresses of its n functions and cedes the control of the CPU without executing any return statement. Simultaneously on the same CPU core, a victim process is executing ECC scalar multiplication operation using the OpenSSL library. It is worth mentioning in this context that the POC attack assumes that the victim intentionally yields the CPU after every iteration and the attack on OpenSSL ECC is presented as a use-case of the POC. Therefore, we assume the victim executes a `nanosleep` or `sched_yield` [6] after each iteration of the scalar multiplication operation.

The objective of the spy program is to determine the number of calls made to `BN_add` and `BN_sub` functions by the Montgomery ladder implementation for each bit of the secret scalar. The spy program is running continuously and measuring its own execution time to check whether any of its return addresses have been pushed out of the stack. As the victim process executes the scalar multiplication in a bit by bit manner, it makes a number of calls to `BN_add` and `BN_sub` functions depending on the value of the scalar bit and the affine co-ordinates of the elliptic curve point. As usual, the function calls made by the

[6] In the later part, we will relax this requirement by abusing the *deadline scheduler* to perform the synchronisation between victim and the spy.

scalar multiplication operation pushes the return addresses of the corresponding functions into the RAS. As the RAS is being shared by both victim and spy, the entry of new return addresses removes equal number of entries of the spy. Therefore, the spy which is running in parallel is able to precisely deduce the total number of `BN_add` and `BN_sub` functions, being called for each bit of the scalar.

5.3 Template Attack on ECC Scalar Multiplication

Template Attacks are profiled side channel attacks where an attacker builds a profile using different combinations of the secret key on a particular device and then matches the template with the actual values gathered from the target application on a similar device. We discussed how RASSLE can be used to determine the total number of additions and subtractions performed per bit of the scalar. Using this information, we aim to perform a template attack on the widely popular OpenSSL implementation of the NSA recommended elliptic curve P-384.

In this part of the exploit, we follow an iterative linear attack procedure. The adversary iteratively progresses through the bits of the secret scalar (key) starting from the most significant bit (msb) to the least significant bit(lsb). At a particular instance, the adversary targets the i^{th} bit of the secret scalar, given the assumption that the adversary already knows the first $(i-1)$ bits, and further aims to retrieve the subsequent bits one by one. Therefore, the adversary needs to generate new set of templates for each bit position and correctly deduce that particular bit in order to proceed with the following bits. We also assume that the attacker is aware of the structure of the algorithm under attack, since the OpenSSL implementation codes are publicly available, making template formation on EC curve behavior used in the scalar multiplication reasonable.

During the template building phase, the attacker simulates the number of `BN_add` and `BN_sub` function calls for each bit. As the number of `BN_add` and `BN_sub` function calls depend on the particular bit being processed and the affine coordinates of the curve point, the attacker builds the templates for each bit based on the total number of `BN_add` and `BN_sub` functions executed for a fixed set of input plaintexts. This fixed set of plaintexts will be used by the attacker for both template generation and matching phase. More precisely, for any particular bit, say i^{th} bit, ❶ the attacker performs point multiplication using a set of unique inputs (curve points) fixing the i^{th} bit to be both '0' and '1', we denote this as $\mathcal{G}_0$ and $\mathcal{G}_1$. ❷ Next, for each input, the attacker estimates the total number of addition (`BN_add`) and subtraction (`BN_sub`) function calls made by the ECC program for i^{th} bit assuming its value to be both 0 and 1. ❸ Simultaneously, the spy process measures the execution time using RASSLE. Therefore, the attacker gathers two sets of information - total number of `BN_add` and `BN_sub` and timing information for each bit.

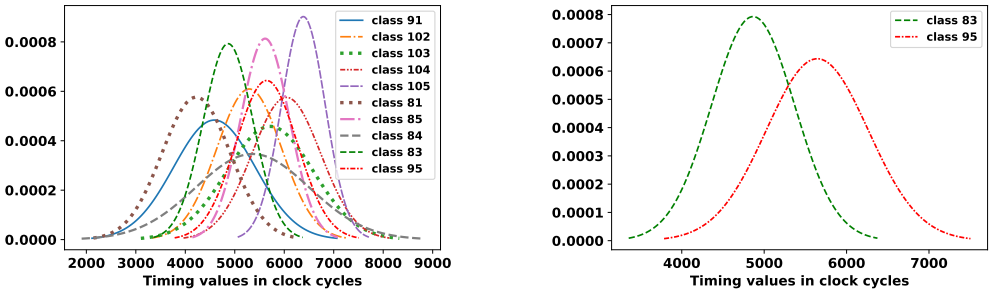
5.3.1 Template Building Phase

For simplicity, we introduce an encoding scheme to represent the total number of `BN_add` and `BN_sub` function calls as unique *classes*. Suppose, for a particular input and i^{th} bit, the Montgomery ladder of the scalar multiplication operation executes $\mathcal{X}$ `BN_sub()` and $\mathcal{Y}$ `BN_add()` function calls. We represent this class as $(\mathcal{XY})$. Based on the classes, we segregate the corresponding inputs and the associated timing values, i.e., we create hypothetical $\mathcal{BIN}$ corresponding to each class which contains the corresponding plaintext and the timing value observed from the spy. For example, fixing the i^{th} bit to be 0 yields:

$(\mathcal{XY})^{\mathcal{G}_0}$: inputs which simulate to have $\mathcal{X}$ `BN_sub()` and $\mathcal{Y}$ `BN_add()` function calls.

$\mathcal{BIN}^{\mathcal{G}_0}_{(\mathcal{XY})}$: timing values corresponding to inputs in $(\mathcal{XY})^{\mathcal{G}_0}$

Similar classification can be applied for $(\mathcal{XY})^{\mathcal{G}_1}$ and $\mathcal{BIN}^{\mathcal{G}_1}_{(\mathcal{XY})}$ for $\mathcal{G}_1$. Further, we calculate the first order moments for each $\mathcal{BIN}^{\mathcal{G}_0}_{(\mathcal{XY})}$, $\mathcal{BIN}^{\mathcal{G}_1}_{(\mathcal{XY})}$ to form our set of *candidate*



(a) Distribution of all *bins* (b) Distribution of two selected *bins*

Figure 5: Template formation phase: Distribution of timing observed using RASSLE for $(10m+n)$ classes where m = no. of subtraction and n = no. of addition.

templates. Fig 5(a) shows an example of a candidate template with distribution for multiple bins for a particular bit of the key. Depending on the set of plaintexts, the number of unique *classes* can vary. In our experiments, we found that the classes belong in the range 81 to 85, 90 to 96 and 100 to 106. So typically number of `BN_sub` function calls varies between 8 to 10 and number of `BN_add` function calls vary between 0 to 6.

As already mentioned, each *class* corresponds to a unique $\mathcal{BIN}$ which contains the associated plaintexts and the mean and standard deviation values of all the corresponding timing values. Out of these multiple $\mathcal{BIN}$s, the attacker selects a pair which contains a relatively high number of inputs, i.e., sets with high cardinality and non-overlapping timing distributions. In our experiments, we performed the template building phase using $10,000$ plaintexts, which we found to be enough to derive a reliable consistent statistic. We empirically selected two bins such that each one contains more than 150 plaintexts and there is a visible difference in their mean statistic. An example of such a pair of $\mathcal{BIN}$s is shown in Fig 5(b). For each bit, the entire template building phase took approximately 2 hours in our experimental setup. It must be noted in this context that for each bit, the template building phase will generate a whole lot of $\mathcal{BIN}$s, among them selection of two appropriate bins will qualify as good templates. Therefore, at the end of the template generation phase, we will have four $\mathcal{BIN}$s as our template - comprising of two fairly separated distributions based on the timing templates from each of $(\mathcal{XY})^{\mathcal{G}_0}$ and $(\mathcal{XY})^{\mathcal{G}_1}$ from the simulation when the i^{th} bit is 0 and 1.

5.3.2 Template Matching Phase

In template matching phase, we aim to predict the correct value of the i^{th} bit using the templates generated assuming the bit to be 0 and 1. While template generation is considered to be an *offline* phase, where the attacker simulates the scalar multiplication operation to build suitable templates, the matching phase is rather *online* where the only available information to the attacker is the timing value observed through RASSLE corresponding to each key bit. At the end of the template generation phase, the attacker has four $\mathcal{BIN}$s - two for corresponding bit assumed to be 0 and two for bit assumed as 1.

Now, the attacker observes the encryption process once again using only those inputs that were associated with these four selected classes. The crux of template matching lies in the fact that the timing values for correct i^{th} bit matches with either of the two set of templates, but not both. If the observed timing measurements over the correct bit correlates with the template generated for $\mathcal{G}_0$, then the actual value of the secret bit is revealed to be 0; else the matching works for the templates for $\mathcal{G}_1$.

Similar to the generation phase, the attacker encrypts the selected set of inputs and records the timing values observed by spy process using RASSLE. Once the online data

Table 2: No. of plaintexts and total time taken for the bit indices 348, 349 and 350 during template matching phase.

	Bit Index	No. of plaintexts	Time taken (secs)
1	348	698	76.913623
2	349	835	91.000569
3	350	752	82.466902

acquisition step is complete, the timing values are placed into one of the four $\mathcal{BIN}$s based on the plaintexts. Once the segregation of timing values into the appropriate $\mathcal{BIN}$s is complete, we take the mean and standard deviation for each of the four bins and plot the distributions, similar to the generation phase. Out of the four bins, only two bins will correspond to the actual template, thereby revealing the actual value of the bit.

5.4 Experimental Validation

We performed our experiments on Intel i7-7700 powered by Ubuntu 18.04 (kernel 4.15.0). We assume that the attacker and the victim are co-located on the same core [7], therefore sharing the same RAS for their operations. The RAS in our experimental setup has a capacity of 16, i.e., it can hold upto 16 return addresses. The spy program continuously fills up the RAS with 16 nested function calls and measures the execution time. The victim process executes the Montgomery ladder implemented using conditional swaps for each bit of the secret scalar.

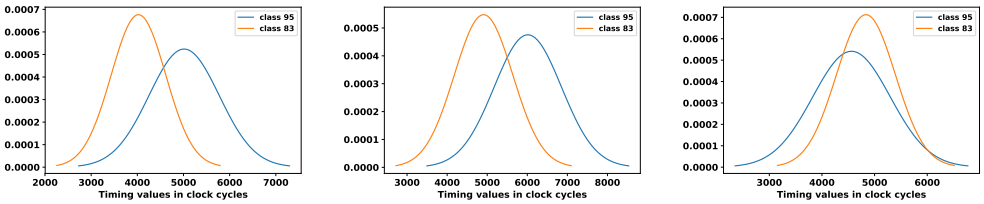
Now, suppose for i^{th} bit of the secret scalar, the ladder executes m `BN_add` functions and n `BN_sub` functions. These $(m+n)$ function calls pushes their corresponding return addresses which in turn forces the older entries out of the stack. This eventually results in increased execution time for the return statements of the spy process. The spy process observes the *difference in execution times* of all the 16 functions and observes a spike in timing value due to underflowing of the RAS.

Assuming the attacker knows the first $(i-1)$ bits, in the template generation phase, we run the scalar multiplication using a fixed set of $10,000$ plaintexts. These many inputs were chosen for no particular reason, but to make sure we get a significant consistent statistic for each of the template classes. As described earlier, the adversary uses the simulated number of `BN_add` and `BN_sub` function calls executed for the i^{th} bit to form the $\mathcal{BIN}$s from their observed timing values through RASSLE. Similar to the template generation procedure, in the template matching phase we decide on the correct guess by segregating the observed timing values into appropriate bins based on the plaintext. Fig 6 shows the results of the template generation and matching process for 350^{th} bit. Fig 6(a) shows the distribution of the selected template. The classes 95 and 83 provides a stark difference in their distributions and thus we selected these two classes as our template. Fig 6(b) and (c) shows the distributions of the aforementioned classes (or bins) for the correct estimate (0) and wrong estimate (1) of the bit value respectively. As evident from the figures, the distribution of the correct estimate matches with the actual template for the same pair of classes. Similarly, Fig 7 shows the results of another instance of template generation and matching process for 348^{th} bit for selected classes, correct estimate and the wrong estimate. Table 2 shows the number of plaintext used and total time taken in template matching phase for three consecutive bits.

6 Synchronization in a Purely Asynchronous Setup

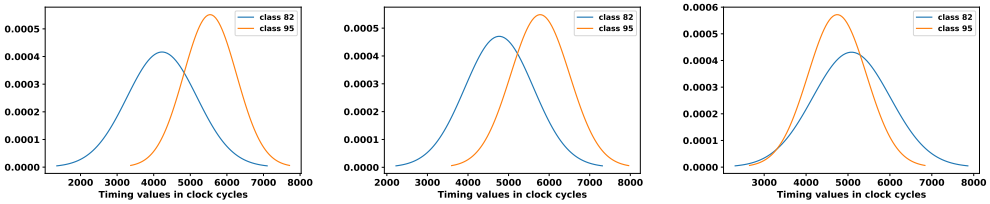
The attack on OpenSSL ECC scalar multiplication presented in Section 5 demonstrates how RASSLE can be utilized to leak information about the control flow of another process. In

[7] This can be easily achieved from user-space using `taskset` command.



(a) Template for $\mathcal{G}_0$ (b) Correct estimate - '0' (c) Wrong estimate - '1'

Figure 6: Template matching for bit index 350.



(a) Template for $\mathcal{G}_1$ (b) Correct estimate - '1' (c) Wrong estimate - '0'

Figure 7: Template matching for bit index 348.

this section, we present another use case of RASSLE in which a spy running simultaneously with victim process on the same core can precisely achieve synchronization by abusing the operating system's scheduler policy, and collect timing information at fine granular steps which is enough to retrieve ephemeral keys from ECDSA implementation of OpenSSL.

6.1 Assumptions on the Threat Model

We would continue with the earlier assumption that the spy and victim process start their execution in the same logical core. However, unlike the victim code modification requirement mentioned in Section 5, in this scenario, the victim code does not need any extra synchronization aid, which makes the attack scenario realistic. A user only needs `CAP_SYS_NICE` capability to launch the attack from userspace. This is a reasonable assumption in a multi-user environment, since a user can optimize the parameters of the *deadline scheduler* (discussed in next subsection) to achieve maximum performance benefit for its "own task" [8]. We execute both victim process and spy process simultaneously on a server-environment built on Intel Xeon CPU E5-2609 v4 (Broadwell) running Red Hat Enterprise Linux Server 7.7 (kernel 3.10.0-1062.9.1.el7.x86_64).

6.2 Getting Synchronization to work with the Deadline Scheduler

Most Linux-based operating systems offer a number of scheduling policies, which are crucial artifacts for controlling two asynchronous processes' execution. Among the available policies, the *deadline scheduler* is particularly interesting because it imposes a "deadline" on operations to prevent starvation of processes. In the deadline scheduler, each request by a process to access a system resource has an expiration time. A process holding a system resource does not need to be forcefully preempted, as the deadline scheduler automatically preempts it from the CPU after its request expiration time.

[8] `CAP_SYS_NICE` seemingly assists performance-tuning in server-environments, and has no known security-implications for the deadline-scheduler.

The operation of deadline scheduler depends on three parameters, namely 'runtime', 'period', and 'deadline'. These parameters can be adjusted using `chrt` command, which can be executed from user-level privilege by acquiring `CAP_SYS_NICE` permission[9]. *A normal user who obtains the* `CAP_SYS_NICE` *permission, does not gain any elevated privilege apart from adjusting the deadline scheduler parameters.* Obtaining scheduling privilege for user-level processes is practical in multiple real-life scenarios, where a normal user can use the privilege to efficiently utilize system resources for better performance. We observed that the covert channel described in Section 4.3 works in the presence of deadline scheduler [10] by choosing appropriate parameters instead of using `sched_yield()` or `nanosleep()`. The command to run an `<executable>` using deadline scheduler is as follows:

`chrt -d -sched-runtime` t_1 `-sched-deadline` t_2 `-sched-period` t_3 `0 <executable>`

where, t_1, t_2, and t_3 are the parameter values for 'runtime', 'deadline', and 'period' respectively. The kernel only allows scheduling with $t_1 \leq t_2 \leq t_3$. The usual practice is to set `sched-runtime` to some value greater than the average execution time of a task. We estimate the average execution time of each iteration of the target executable (ECC Montgomery ladder in this case) in terms of CPU clock cycles and convert the values into nanoseconds using CPU clock frequency. We set t_1 with the obtained value in nanoseconds. Further, we set the parameter `sched-deadline` to a value $t_2 = t_1 + \delta$ such that the ECC process leaves the CPU after execution of a single Montgomery ladder iteration. We set the parameter `sched-period` to a value $t_3 = 2 \times t_1$. The reason for such an assignment is discussed later in this section. *It should be noted that the procedure described here considers no change in the victim code and requires no use of* `sched_yield()` *or* `nop` *or* `nanosleep()` *in the victim executable to preempt it from CPU.*

In the proposed approach, a spy works in two phases - ❶ it fills up the Return Address Stack (RAS) and cedes control of the CPU; ❷ it probes the RAS by executing return statements after the victim has finished its scheduled operation. The spy process also uses multiple `nop` instructions such that operations in ❶ takes approximately equivalent time to that of one iteration of the ECC Montgomery ladder. The reason behind the use of `nop` instructions in the spy process is that the victim and the spy can then be executed with a single `sched-runtime` parameter of the deadline scheduler. We also use `sched_yield()` at the end of ❶ (i.e., in the spy process) to relinquish CPU control to the victim in order to ensure that the RAS remains full with the return addresses of the spy. As both spy and victim are running with the same 'runtime', the deadline scheduler preempts the victim process after one iteration of the ECC Montgomery ladder and gives the control of the CPU back to spy. The spy now probes the RAS by executing the return addresses, observes the execution timing through RASSLE, and again fills up the RAS. The `sched-period` signifies the periodicity of the task, which is set equal to $2 \times t_1$ to target each iteration of the ECC Montgomery ladder. In our experiments, we used $t_1 = 3600$, $t_2 = 3700$, and $t_3 = 7200$ to validate our proposed approach in the setup discussed in Section 6.1. We performed empirical parameter-optimizations to achieve a reasonable success-rate on the target platform.

6.3 Case-study on ECDSA Signature Generation Algorithm

In this section, we present a case study to illustrate how our attack strategy may be deployed to break the security of ECDSA [X9.03] - a widely used digital signature algorithm that uses Elliptic Curve Cryptography (see Appendix B for details). Our aim is to recover the secret signing key from an implementation of the ECDSA signing algorithm.

At a high level, the attack strategy against ECDSA can be divided into two steps:

[9] The permission can be provided to a user using `setcap cap_sys_nice+ep /usr/bin/chrt`.

[10] In order to inspect the underline scheduler in the working system one can use `cat /sys/block/sda/queue/scheduler`.

- **Step-1 (online):** In this online step, we use our RASSLE-based attack strategy to perform a targeted recovery of a fraction of the most significant bits of the nonces sampled by the ECDSA signing algorithm (details presented subsequently).
- **Step-2 (offline):** In the offline step, we combine the partial nonce information recovered in the online step with lattice-based cryptanalytic techniques to recover the ECDSA signing key (details presented subsequently).

6.3.1 Online Phase: Template building and matching for ECDSA on curve P-256

We first describe the online phase of our attack. We target an OpenSSL implementation of the scalar multiplication algorithm over curve P-256 used by ECDSA for nonce-generation during signing. Now, as mentioned in [ABuH+19], OpenSSL implemented a constant time scalar multiplication for NIST P-256 curve which uses secure table lookups (through masking) and fixed-window combing. Note that these security features were not present in the scalar multiplication implementation over NIST curve P-384 in OpenSSL [ABuH+19].

Our template-based attack strategy on the scalar multiplication implementation builds upon RASSLE and works despite all of the countermeasures enabled for P-256. We avoid detailing the scalar multiplication implementation in OpenSSL; it suffices to state that it is a Montgomery Ladder-based implementation that still uses `BN_add` and `BN_sub` function calls. These are precisely the function calls that we are going to track down using RASSLE.

Trace Collection and Template Building. Note that fundamentally, the scalar multiplication used by the ECDSA signing algorithm is the same as the one we targeted earlier using RASSLE in Sections 5.3 and 5.3.2. However, we need to use a different template building methodology here as compared to the one proposed earlier. This is because we now consider retrieval of bits for the underlying ephemeral nonces.

In particular, as the nonces are generated randomly during ECDSA signature generation and used only once, it is not feasible to simulate multiple `BN_add` and `BN_sub` function calls for each bit position and build templates by fixing the bit values to '0' and '1'. Therefore, we now consider building templates of a *window of unknown bits* instead of a bit-by-bit iterative approach. Dictated by the requirements of the offline lattice-based cryptanalysis phase (see Section 6.3.2 and Appendix C for more details), we build templates for $\ell > 1$ most significant bits of the nonce. For template formation, the adversary executes a spy and a *dummy* victim process performing ECC scalar multiplications simultaneously using deadline scheduler. The three parameters (i.e., `sched-runtime, sched-deadline, sched-period`) of deadline scheduler are set as mentioned in Section 6.2.

The template building process proceeds as follows:

1. For ℓ bits from the MSB position of the nonces, there can be 2^ℓ combinations of bit sequences possible. We build templates for bit sequences of each of these 2^ℓ combinations.
2. For each of the 2^ℓ bit sequences:
   - The dummy victim process performs ECC scalar multiplications using 2^ℓ nonces by changing the ℓ most significant bits while keeping the other bits same.
   - The spy, running in parallel, continuously fills up the RAS and probes it to observe the timing values through RASSLE. As the adversary requires to retrieve only ℓ MSBs of the nonce, the spy considers only those ℓ timing observations that correspond to the ℓ MSBs.

   We refer to these ℓ timing values to constitute a *trace* and to the corresponding bit positions as *trace points*; the overall process of creating the trace is referred to as the *trace collection step*.

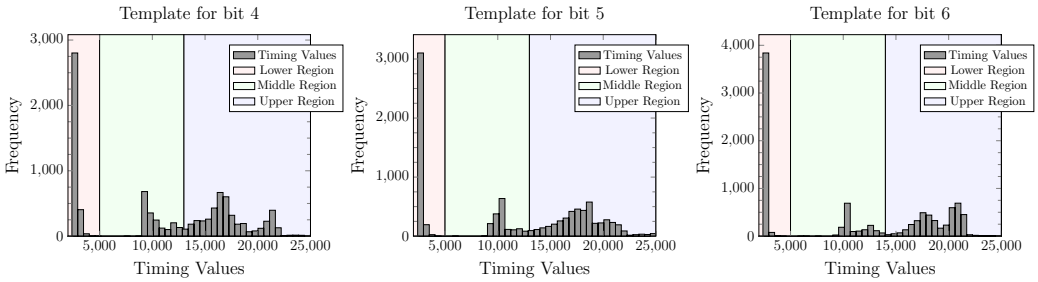


Figure 8: Three different regions observed in the templates for bit 4, bit 5, and bit 6 for bit combination '101001'.

3. In order to minimize the system noise, the trace collection step is executed $m > 1$ times ($m = 10,000$ in our case) for each of the 2^ℓ bit sequences.

4. Thus after the trace collection step, the adversary now has $m \cdot \ell \cdot 2^\ell$ timing samples for ℓ MSBs of each of the 2^ℓ bit sequences of the nonce.

5. Next, the adversary selects *medians* from each of these timing distributions as the representative template for a particular bit position of a particular sequence. Therefore, the adversary constitutes a set of templates $[ml] = \{ml_{i,j} : i \in [1, 2^\ell], j \in [1, \ell]\}$, which contains $\ell \cdot 2^\ell$ timing values (i being the sequence number and j being the bit position).

In our experiments, we build templates for a window of $\ell = 6$ MSBs. Therefore, for each of the 6 bit positions and 2^6 combinations of random nonces, the spy collects timing samples through RASSLE. Fig 8 shows the timing distribution for a particular nonce combination i ($i \in 2^6$) for three bit positions ($j = 4, 5, 6$ where $j \in 6$). It is apparent from the figure that the distribution of timing samples for each bit position can be subdivided into three regions. The most intuitive explanation of this observation is that the spy tries to achieve synchronization with the aid of deadline scheduler without explicit handles inside victim code (unlike the attack proposed in Section 5). Due to the absence of perfect synchronization mechanism, we suspect there is a mutual overlap between the timing samples of any two adjacent trace points (bit positions), which make us define three separate regions in the timing distribution:

- *lower region*: where the timing value observed for a trace point overlapped with the previous trace point and thus it quotes lesser time than the one with perfect synchronization,
- *middle region*: where perfect synchronization was achieved and the timing sample for the particular trace point remains unaffected by adjacent bits' executions,
- *upper region*: where the timing value for the trace point also overlapped with the timing observation from following trace point and thus quotes higher value than the one with perfect synchronization.

From these timing distributions for each bit of the 2^ℓ nonces, we select the median from each region as the candidate elements for template building, which leads to 3 possible template values for each bit position i of a particular bit sequence j. Therefore, at the end of the template building phase, the adversary will have $3\ell \cdot 2^l$ templates for ℓ MSB positions of the 2^ℓ nonce candidates.

Template Matching on Ephemeral Nonces. In our experiments, we conduct an end-to-end attack which can retrieve a randomly chosen ECDSA signing key with only a pair of 500 random signatures and nonces (these numbers are chosen in general based on the templating parameters for the target system; might vary on a different setup). According

Table 3: Ordering of candidate nonce combinations (6 MSBs) by Least Square Error(LSQ) method. The first element of each tuple represent the candidate nonce and the second element represent the LSQ values($\times 10^6$). The correct combination is marked in bold.

Sl. No.	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5
nonce 1	(101110, 11.5)	**(100000, 12.7)**	(101101, 16.3)	(100010, 16.5)	(100001, 16.5)
nonce 2	**(100110, 12.0)**	(100100, 12.2)	(101000, 12.5)	(100101, 12.9)	(101010, 15.4)
nonce 3	(100011, 40.2)	(100000, 42.7)	(100010, 42.8)	**(100110, 50.1)**	(100100, 53.0)
nonce 4	**(100000, 4.88)**	(100110, 8.95)	(100100, 9.23)	(100101, 9.27)	(100011, 10.17)
nonce 5	**(100001, 25.3)**	(100000, 35.1)	(100010, 39.9)	(100011, 41.2)	(111101, 49.0)

Table 4: Time taken for ECDSA key retrieval from partially leaked nonces.

Sl. No.	No. of signatures	Nonces retreived	Online Phase	Offline Phase
1	500	219	5.55s	54m 46s
2	500	238	6.66s	55m 31s
3	500	202	5.78s	61m 09s
4	500	258	5.70s	51m 37s
5	500	246	8.35s	53m 50s
6	500	219	7.60s	71m 05s

to the ECDSA algorithm (discussed in the Section 6.3), the nonces are the secret random integers that are used to perform scalar multiplication with the base point of the curve.

Similar to the template building phase, the spy craftily fills up the RAS and records the timing observations using RASSLE. We term this phase as *online* phase where the spy runs in synchronisation with the victim process by abusing the *deadline scheduler* and observes the timing leakages for ephemeral nonces. While in the *offline* phase, the attacker (or spy) tries to extract the 6 MSBs of each of these 500 nonces and eventually attempts to extract the secret signing key using Lattice Reduction (for more details, refer to Appendix C). We first segregate the timing observed by spy corresponding to the 6 MSB positions of the nonce. The objective here is to perform template matching for 6 individual bit positions of these random nonces with all the available templates. For a particular bit position, we have 2^6 templates each having three regions. Next, we select the median which has least difference with the actual observed timing value. Finally we perform a Least Square Error (LSQ) to determine the top five templates that represent the possible combinations for the 6 MSBs. Therefore, for 500 nonces, we have (500×5) candidate combinations of the 6 MSBs. Table 3 presents a typical output after the template matching phase where first 6 bits of the candidate nonces are ordered based on their respective LSQ values.

6.3.2 Offline Phase: Key-Recovery using Lattice-based Cryptanalysis

Given the "noisy" leakage samples on the "partial nonces" [11] used by the ECDSA signing algorithm, we recover the ECDSA signing key using a combination of lattice reduction algorithms and statistical mixing-and-matching of leakage samples as described in a number of previous works [Mic01, NS02, BvdPSY14, WSBS20, JSSS20]. A detailed description of the lattice-based cryptanalytic techniques is presented in Appendix C.

For implementing the lattice-reduction algorithms, we resort to using the publicly available `fpylll` library [fpy], which has in-built modules for solving the Closest Vector Problem (CVP) over lattices with integral basis matrices. The mathematical details of the reduction from recovering the ECDSA secret key from partial nonce-leakages to solving a CVP instance is presented in Appendix C. We adopt a trial-and-error approach where we randomly select 200 candidate partial nonces to: (a) create the hidden number problem instances, (b) convert them into the lattice and the target vector for the CVP problem instance, and (c) solve the CVP problem instance using the `fpylll` solver to arrive at a

[11] The "partial nonce" guess refers to the candidate msbs for the nonce corresponding to each signature.

guess for the secret key. If the guess is correct, the attack outputs the recovered secret key; otherwise, it repeats the same process for a different randomly selected set of instances. Note that the attack can trivially identify when the correct secret key has been recovered by checking if it yields the correct public key (which is available for verification). This check involves a single deterministic scalar multiplication. So when the attack terminates by outputting a secret key, we can be sure that the correct secret key has been recovered, as opposed to merely "guessing" that the correct secret key has been recovered.

Based on our experiments, for $l = 6$ (l being the number of MSBs of the nonce leaked via RASSLE), the attack requires approximately 4000 - 5000 trials, where each trial outputs a candidate secret key in the range of 0.010 - 0.015 secs. Hence, the overall lattice-based post-processing for secret key recovery takes close to an hour. Table 4 shows the detailed results of ECDSA key retrieval from the 6 MSBs of nonces leaked through RASSLE. We performed key retrieval of 6 unknown keys using 500 signatures and randomly generated nonces. The number of nonces successfully identified during our template matching phase is also shown along with the offline and online time taken for the entire operation.

7 Mitigation

As demonstrated so far, RASSLE exploits the RAS to establish a timing-based covert channel. The illustration of the covert channel is a fundamental observation, which made us explore other threat models. Thus, mitigation can be proposed at various levels. If we revisit the example of OpenSSL implementation on an algorithmic level, point randomization would have made this key recovery difficult but perhaps not impossible. It becomes more difficult for the adversary to simulate the number of intermediate function calls and build the classification as shown in this paper, but on the other hand, it might open up other vulnerabilities that are particularly not affected due to the input basepoint randomization. On the other hand, the shared hardware stack-based structure still stays exploitable. It can be abused by processes at their own wish, leading to severe security exploit across processes co-located on a cloud server or through the browser. This opens up new avenues and research opportunities that could be explored in the coming years.

8 Conclusion

In this paper, we focused on the Return Address Stack - a core component of the speculative execution subsystem in almost all modern processors. We proposed a generic methodology to reverse engineer the RAS in modern-day processors. We demonstrated that due to the underflow condition of the RAS, a subtle covert channel could be created between two co-located processes. We christened it RASSLE and presented a proof of concept work where two co-located processes communicate with each other using the shared RAS. As a practical use-case of RASSLE, we demonstrated an exploit on ECC scalar multiplication over the P-384 curve in the OpenSSL library. We showed how RASSLE can be used in a completely asynchronous setting by adjusting deadline scheduling policy from user-level privileges. We also showed an exploit on breaking the ECDSA signature generation algorithm over curve P-256 using RASSLE.

Acknowledgements

We thank Daniel Page and the other anonymous reviewers of TCHES for their valuable feedback and comments. We also thank Arnab Sarkar for discussions and insights on using the deadline scheduler. This work is partially funded by the Department of Science and Technology, Government of India under the Swarnajayanti Fellowship program, and by the Postdoctoral mandate KU Leuven (PDM), by the Flemish Government through FWO project TRAPS, and by gift-funding from Intel.

References

[ABF+16] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 422–435. Association for Computing Machinery, 2016.

[ABG10] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124. Springer, 2010.

[ABuH+19] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 870–887. IEEE, 2019.

[Aci07] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *ACM workshop on Computer security architecture*, pages 11–18, 2007.

[AKS07a] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320, 2007.

[AKS07b] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, pages 225–242. Springer, 2007.

[Ber05] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[BH09] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 667–684. Springer, 2009.

[BHH01] Dan Boneh, Shai Halevi, and Nick Howgrave-Graham. The modular inversion Hidden Number Problem. In *ASIACRYPT 2001*, pages 36–51, 2001.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[BM15] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing performance monitors for compromising keys of RSA on Intel platforms. In *Cryptographic Hardware and Embedded Systems*, pages 248–266. Springer, 2015.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*, pages 355–371. Springer, 2011.

[BvdPSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES 2014*, pages 75–92, 2014.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[CSH+20] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break it, fix it, repeat. In *ASIA CCS 2020-Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ACM/IEEE, 2020.

[DKPT17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ Abort: A timer-free high-precision l3 cache attack using Intel {TSX}. In *USENIX Security Symposium 2017)*, pages 51–67, 2017.

[EPAG16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[ERAGP18] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[fpy] A python interface for https://github.com/fplll/fplll. https://github.com/fplll/fpylll.

[GII+15] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on AES. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer, 2015.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium 2015)*, pages 897–912, 2015.

[GVY17] Daniel Genkin, Luke Valenta, and Yuval Yarom. 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 '17, page 845–858. Association for Computing Machinery, 2017.

[Hor16] Taylor Hornby. Side-channel attacks on everyday applications: distinguishing inputs with flush+ reload. *BlackHat USA*, 2016.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy (S&P)*, pages 191–205. IEEE, 2013.

[IES15] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ a: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 591–604. IEEE, 2015.

[IGA+15] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-VM RSA key recovery in a public cloud. *IACR Cryptol. ePrint Arch.*, 2015:898, 2015.

[IT02] Tetsuya Izu and Tsuyoshi Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In *International Workshop on Public Key Cryptography*, pages 280–296. Springer, 2002.

[JSSS20] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sýs. Minerva: The curse of ECDSA nonces systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):281–308, 2020.

[KGGY20] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 1–19. IEEE, 2019.

[KKSAG18] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT) 2018)*, 2018.

[Koc96] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium 2018*, 2018.

[Mic01] Daniele Micciancio. The hardness of the closest vector problem with preprocessing. *IEEE Transactions on Information Theory*, 47(3):1212–1215, 2001.

[MML+19] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading kernel writes from user space. *arXiv preprint arXiv:1905.12701*, 2019.

[MR18] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.

[Nat15] National Security Agency. Commercial national security algorithm suite. https://apps.nsa.gov/iad/programs/iad-initiatives/cnsa-suite.cfm, 2015. [Online; accessed 20-December-2020].

[NS02] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Digital Signature Algorithm with partially known nonces. *J. Cryptol.*, 15(3):151–176, 2002.

[Ope19] OpenSSL. OpenSSL cryptography and SSL/TLS toolkit. https://www.openssl.org/, 2019. [Online; accessed 25-June-2019].

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[Per05] Colin Percival. Cache missing for fun and profit, 2005.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[SAMC98] Kevin Skadron, Pritpal S Ahuja, Margaret Martonosi, and Douglas W Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 259–271. IEEE, 1998.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[Tur18] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. *URL https://support. google. com/faqs/answer/7625886*, 2018.

[VSMÖ+19] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 88–105. IEEE, 2019.

[WSBS20] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers - big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations. In *USENIX Security Symposium 2020*, pages 1767–1784, 2020.

[X9.03] ANSI X9.42-2003. Public Key Cryptography For The Financial Services Industry: Agreement Of Symmetric Keys Using Discrete Logarithm Cryptography. https://webstore.ansi.org/standards/ascx9/ansix9422003r2013, 2003.

[YB14] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014:140, 2014.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: a high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium 2014*, pages 719–732, 2014.

[ZJOR11] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on Security and Privacy (S&P)*, pages 313–328. IEEE, 2011.

A Elliptic Curve Cryptography in OpenSSL

Well known and widely used crypto libraries, such as OpenSSL [Ope19], provide the support for elliptic curve based algorithms. Although the most recent versions of these libraries are equipped with some of the countermeasures for micro architectural attacks, there are still inherent avenues which are exploited in leak secret information. A number of attacks [ABG10, BT11, ABuH+19] have been proposed over the years which shows the vulnerability of conditional branching implemented in various cryptographic open-source libraries. Most of these attacks exploit the conditional branching mechanism of Branch Prediction Unit. However, libraries sanitized with recent branch prediction attack countermeasures are immune to these simple misprediction based attacks. Due to long history of timing attacks on ECC, OpenSSL [Ope19] implemented several countermeasures without scalar-dependent branches, specially to protect the scalar multiplication operation. The scalar multiplication is implemented using Montgomery ladder with conditional swaps and Non-Adjacent Form (wNAF) for scalar representation. The scalar k is transformed to its corresponding wNAF representation and based on this representation, a series of *double* and *add* operations are executed to perform the multiplication. Further, these two operations are further subdivided into lower level field arithmetic, such as, shift, add, subtract, multiply and square operations. OpenSSL also provides a support for point randomization, where the input basepoint is randomized in every call to the ECC function. However, in this particular work, we have demonstrated a Chosen Plaintext attack with the basic timing observations, so we have discussed point randomization to be a valid countermeasure of the existing attack.

Scalar Multiplication in OpenSSL. In OpenSSL, each elliptic curve has a unique method structure which contains function pointers to fundamental ECC operations. Due to the various timing based side channel attacks proposed over the years, OpenSSL implemented a lot of countermeasures to thwart these attacks, specially for NIST curves over prime fields. However, as pointed out in [ABuH+19], the P-384 curve does not have the popular countermeasures like table look-ups, fixed-window combing, etc unlike other widely used curves. The scalar multiplication operation is implemented using the function `ec_wNAF_mul` which first transforms the scalar into wNAF representation and then executes a series of double and add operations to perform the multiplication. To provide protection against side channel attacks, OpenSSL use the Montgomery ladder algorithm with conditional swaps for scalar multiplication. The algorithm is implemented without key-dependent branches and performs the same number of addition and double operations irrespective of value of the bit processed. The conditional swaps avoid any branches or memory accesses that depend on the scalar. The Montgomery Ladder step operation is implemented using the *differential addition-and-doubling* method [IT02] as shown in Fig 9. The field multiplication (`field_mul`) is implemented by the function `BN_mod_mul` where the modular reduction is performed by the `BN_nnmod` function. Further, looking closer into the `BN_nnmod` as in Fig 10, the modular reduction executes either function `BN_add` or `BN_sub`, realised through a ternary operator, which is realised by the compiler similar to a branch statement.

The interesting part of this code snippet is that, the for each bit of the secret scalar, inspite of having no conditional statements which are directly dependent on the secret bit, yet these `BN_add` and `BN_sub` functions are conditionally invoked. The aggregate number of such calls are not constant, and are dependent on both the value of secret scalar and the input basepoint. Therefore,the number of times the `BN_add` and `BN_sub` functions are executed depend on the affine coordinates of the EC point, as well as the value of the secret scalar. Fig 11a shows how the total number of add and sub function calls varies for each bit for two different values of secret scalar denoted as *key1* and *key2*. It is worth mentioning here that instruction based attacks like the L1 instruction cache prime+probe [ABG10] might be a suitable attack technique to exploit this implementation. However, since the

```
int ec_GFp_simple_ladder_step(const EC_GROUP *group,
                              EC_POINT *r, EC_POINT *s,
                              EC_POINT *p, BN_CTX *ctx)
{
    int ret = 0;
    BIGNUM *t0, *t1, *t2, *t3, *t4, *t5, *t6 = NULL;

    if (t6 == NULL
        || !group->meth->field_mul(group, t6, r->X, s->X, ctx)
        || !group->meth->field_mul(group, t0, r->Z, s->Z, ctx)
        || !group->meth->field_mul(group, t4, r->X, s->Z, ctx)
        || !group->meth->field_mul(group, t3, r->Z, s->X, ctx)
        || !group->meth->field_mul(group, t5, group->a, t0, ctx)
        || !BN_mod_add_quick(t5, t6, t5, group->field)
        || !BN_mod_add_quick(t6, t3, t4, group->field)
        || !group->meth->field_mul(group, t5, t6, t5, ctx)
        || !group->meth->field_sqr(group, t0, t0, ctx)
        || !BN_mod_lshift_quick(t2, group->b, 2, group->field)
        || !group->meth->field_mul(group, t0, t2, t0, ctx)
        || !BN_mod_lshift1_quick(t5, t5, group->field)
        || !BN_mod_sub_quick(t3, t4, t3, group->field)
```

Figure 9: Code snippets for Montgomery Ladder step with differential addition-and-doubling in OpenSSL

```
int BN_nnmod(BIGNUM *r, const BIGNUM *m, const BIGNUM *d,
             BN_CTX *ctx)
{

    if (!(BN_mod(r, m, d, ctx)))
        return 0;
    if (!r->neg)
        return 1;
    return (d->neg ? BN_sub : BN_add) (r, r, d);
}
```

Figure 10: Code snippets for the function `BN_nnmod` in OpenSSL.

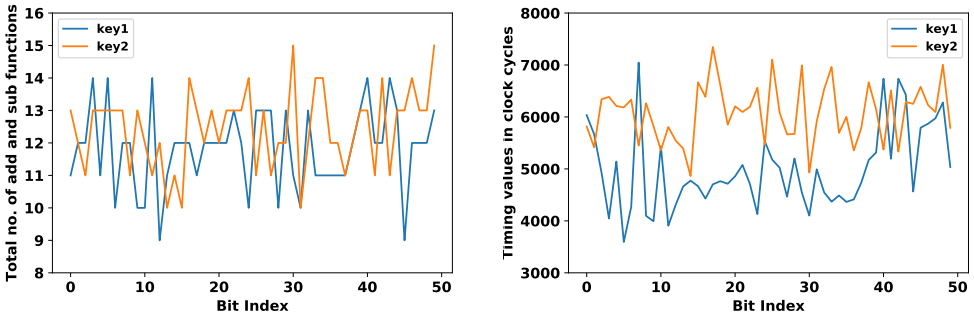
focus of this work is to demonstrate the side channel leakage created by RAS, we aim to determine the total number of `BN_add` and `BN_sub` function calls executed for each iteration of the Montgomery Ladder using the attack technique proposed in the earlier sections.

Timing Variations Observed through RASSLE. As shown in Fig 11b, the timing values observed by the spy using RASSLE varies for different secret scalars denoted as *key1* and *key2* which can be associated with the fact that total number of function calls per bit of the scalar also varies (Fig 11a).

B Background on ECDSA

In this section, we present some background material. ECDSA typically uses an elliptic curve (e.g., NIST P-256 or NIST P-384 [ABuH+19]) with a base point P of prime order q, and consists of the following algorithms:

- KeyGen: Samples a secret signing key sk uniformly in the range $[1, q-1]$ and publishes the corresponding verification key $\mathsf{vk} = [\mathsf{sk}]P$.



(a) Total number of add and sub function calls (b) Timing values observed through RASSLE

Figure 11: Number of add-sub calls and timing values observed for keys *key1* and *key2* for bit index 0 to 50.

- Sign: Samples a nonce k uniformly from $[1, q-1]$ and outputs a signature (r, s) on a message m as:

$$r = x\text{-coord}([k]P) \mod q \quad , \quad s = (h(m) + \mathsf{sk} \cdot r) \cdot k^{-1} \mod q,$$

  where h is a suitably chosen collision-resistant hash function that maps the message m to an integer in the range $[0, q-1]$.

- `Verify`: Given a signature (r, s) on a message m, computes

$$h = \mathcal{H}(m) \quad , \quad u_1 = s^{-1} \cdot h \mod q \quad , \quad u_2 = s^{-1} \cdot r \mod q \quad , \quad Z = [u_1]P + [u_2]Q,$$

  and outputs 1 if $r = x\text{-coord}(Z) \mod q$. Otherwise, outputs 0.

The online phase of our attack strategy specifically targets the scalar multiplication step for computing r in the EDCSA signing algorithm. The aim is to try and recover the ℓ most significant bits (MSBs) of the secret nonce k used by the signing algorithm.

C ECDSA Cryptanalysis from Partially Known Nonces

In this section we will briefly describe ECDSA cryptanalysis based on partially known nonces using lattice algorithms. For the ease of implementation, we have broken up the attack into a sequence of modular tasks.

C.1 A Single HNP Equation

The first sub-task is to implement a function that builds a single equation as part of a larger overall *hidden number problem* (HNP) instance [BHH01]. The equation is to be built from a single ECDSA signature and some partial information about the corresponding nonce used by the ECDSA signing algorithm.

More concretely, assume that we obtain the L most significant bits of an N-bit nonce used to generate a signature (r, s) on a hashed message $h = h(m)$ (h denotes the output of hashing the message m and mapping the hash value to an integer modulo q – in the rest of the discussion, we will only use h instead of m). Letting $x = \mathsf{sk}$, we have

$$s = k^{-1} \cdot (h + x \cdot r) \mod q,$$

which after some simple re-arrangement yields:

$$(r \cdot s^{-1}) \cdot x = k - h \cdot s^{-1} \mod q.$$

Setting $t = (r \cdot s^{-1}) \mod q$ and $z = (h \cdot s^{-1}) \mod q$, we have:

$$tx = k - z \mod q.$$

Now, let a be an integer represented by the L most significant bits of k. Then we can write:

$$k = a \cdot 2^{N-L} + 2^{N-L-1} + e,$$

where e is some *error* term bounded as:

$$0 \leq |e| \leq 2^{N-L-1}.$$

Setting $u = a \cdot 2^{N-L} + 2^{N-L-1} - z$, we have:

$$tx = u + e \mod q$$

Here t and u are known, x is the target unknown and e is small but unknown. We can think of computing (t, u) as setting up a single equation as part of a larger overall HNP instance.

C.2 Building the Overall HNP Instance

The second sub-task is to simply extend the aforementioned approach to build a larger HNP instance consisting of several equations, built from several ECDSA signatures and partial leakages from the corresponding nonces. In other words, given the following:

- N: the total number of bits in each nonce k_i;
- L: the number of known most significant bits of each nonce k_i;
- n: the number of equations to be built;
- $\{(k_{i,1}, \ldots, k_{i,L})\}_{i \in [1,n]}$: a list of the L most significant bits of each nonce k_i;
- $\{h_i\}_{i \in [1,n]}$: a list of the messages (post-hashing and mapping to an integer modulo q);
- $\{(r_i, s_i)\}_{i \in [1,n]}$: a list of the corresponding ECDSA signatures;
- q: the order of the base point P on the elliptic curve used by the ECDSA scheme;

one can create a list of values of the form $\{(t_i, u_i)\}_{i \in [1,n]}$, where we compute each (t_i, u_i) pair as described above.

C.3 HNP to CVP and subsequently solving CVP

The next step is to transform the HNP instance into an instance of the closest vector problem (CVP). Given the list $\big(\{(t_i, u_i)\}_{i \in [n]}, q\big)$, one can construct the following CVP basis matrix:

$$B_{\text{CVP}} = \begin{bmatrix} q & 0 & 0 & \ldots & 0 & 0 \\ 0 & q & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & q & 0 \\ t_1 & t_2 & t_3 & \ldots & t_n & 1/2^{L+1} \end{bmatrix},$$

and the following CVP target vector:

$$\vec{u}_{\text{CVP}} = \begin{bmatrix} u_1 & u_2 & u_3 & \dots & u_n & 0 \end{bmatrix}.$$

As was formally established in previous work [NS02, BvdPSY14, WSBS20, JSSS20], for appropriate choices of parameters N, L, n and q, given B_{CVP} and $\vec{u}_{\text{CVP}}$, a CVP solver should produce a vector

$$\vec{v} = \begin{bmatrix} v_1 & v_2 & v_3 & \dots & v_n & v_{n+1} \end{bmatrix}.$$

such that v_{n+1} is of the form:

$$v_{(n+1)} = x/2^{L+1}.$$

To solve the CVP instance, one can directly use the in-built CVP-solver from the publicly available `fpylll` library [fpy], modulo some scaling adjustments to ensure that the basis matrix has only integral entries.