Single Trace Attack Against RSA Key Generation in Intel SGX SSL

Samuel Weiser
Graz University of Technology

Raphael Spreitzer
Graz University of Technology

Lukas Bodner
Graz University of Technology

ABSTRACT

Microarchitectural side-channel attacks have received significant attention recently. However, while side-channel analyses on secret key operations such as decryption and signature generation are well established, the process of key generation did not receive particular attention so far. Especially due to the fact that microarchitectural attacks usually require multiple observations (more than one measurement trace) to break an implementation, one-time operations such as key generation routines are often considered as uncritical and out of scope. However, this assumption is no longer valid for shielded execution architectures, where sensitive code is executed—in the realm of a potential attacker—inside hardware enclaves. In such a setting, an untrusted operating system can conduct noiseless controlled-channel attacks by exploiting page access patterns.

In this work, we identify a critical vulnerability in the RSA key generation procedure of Intel SGX SSL (and the underlying OpenSSL library) that allows to recover secret keys from observations of a single execution. In particular, we mount a controlled-channel attack on the binary Euclidean algorithm (BEA), which is used for checking the validity of the RSA key parameters generated within an SGX enclave. Thereby, we recover all but 16 bits of one of the two prime factors of the public modulus. For an 8 192-bit RSA modulus, we recover the remaining 16 bits and thus the full key in less than 12 seconds on a commodity PC. In light of these results, we urge for careful re-evaluation of cryptographic libraries with respect to single trace attacks, especially if they are intended for shielded execution environments such as Intel SGX.

KEYWORDS

Controlled-channel attack; side-channel attack; RSA key generation; Intel SGX; Intel SGX SSL; OpenSSL

ACM Reference Format:
Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. 2018. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196494.3196524

1 INTRODUCTION

Side-channel attacks represent a serious threat to cryptographic implementations. Especially software-based side-channel attacks [23] are particularly dangerous, as they can be performed purely by executing code on a targeted machine. These attacks typically exploit various optimizations on the software level, e.g., optimized implementations where executed code paths depend on the processed data [30], and the hardware level, e.g., the cache hierarchy where memory accesses depend on the processed data [47, 52]. In order to prevent such attacks, implementations should favor constant-time programming paradigms [15, 31] over performance optimizations.

Although cryptographic implementations (e.g., in OpenSSL [20]) are often hardened against side-channel attacks on secret key operations such as decryption and signature generation of digital signature schemes, the process of key generation has been mostly neglected in these analyses. While power analysis attacks targeting the prime factor generation during RSA key generation have been investigated [7, 19, 48], software-based side-channel attacks have been considered out of scope for various side-channel attack scenarios. On the one hand, key generation is usually a one-time operation, limiting possible attack observations to a minimum. Especially in case of noisy side channels, e.g., timing attacks and cache attacks, targeting one-time operations such as the key generation procedure seems to be infeasible given only a single attack observation. On the other hand, key generation might be done in a trusted execution environment inaccessible to an attacker.

The situation, however, has changed with the introduction of shielded execution environments that aim to support secure software execution in *untrusted environments* and a possibly compromised operating system (OS). For example, Intel Software Guard Extensions (SGX) [17] provide hardware support that allows software to be executed isolated from the untrusted OS. While the OS cannot access memory of enclaves directly, it is still responsible for management tasks of enclaved programs such as virtual-to-physical page mapping. These management tasks enable new attack techniques such as controlled-channel attacks [12, 43, 51]. By monitoring page faults of enclaved programs, the OS can gather noiseless measurement traces of executed code paths and accessed data, although only at page-size granularity (4 KB). Therefore, the Intel SGX documentation demands side-channel security of code which is to be executed inside enclaves, in particular, to avoid leaking information through page access patterns [16, p. 35].

In light of this powerful attack technique, we investigated the RSA key generation routine of Intel SGX SSL and identified a critical vulnerability that allows to fully recover the generated private key by observing page accesses. Different from other microarchitectural attacks on RSA implementations that targeted the modular inversion [2] or the exponentiation operations [1, 4, 11, 39], the attack presented in this paper targets the RSA key generation routine and can be performed with a single trace. The identified vulnerability is due to an optimized version of the Euclidean algorithm (binary Euclidean algorithm), which features input-dependent branches for checking the correctness of the generated prime factors p and q, *i.e.*, whether $p-1$ and $q-1$ are coprime to the public exponent e, where e is usually fixed to 65 537.

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

By launching a controlled-channel attack, we recover the executed branches of the binary Euclidean algorithm running inside an enclave program and establish linear equations on the secret input, *i.e.*, the prime factors p or q. Based on these equations, we factor the modulus $N = pq$ with minor computational effort on a commodity PC, *i.e.*, in less than 12 seconds for a 8 192 bit modulus, which trivially allows to recover the private key.

Contributions. The contributions of this work are as follows:

(1) We consider an SGX setting and identify a critical vulnerability in the RSA key generation routine of OpenSSL, which relies on the binary Euclidean algorithm (BEA) to check the validity of generated parameters.
(2) We present an attack to recover most of the bits of one of two RSA prime factors, which allows to factor $N = pq$ and to recover the generated private key.
(3) We implement a proof of concept attack that recovers generated RSA keys with a single observation only.
(4) We provide a patch to mitigate the vulnerability, which is even faster than the original implementation.[1]

Outline. In Section 2, we discuss background information on Intel SGX, and related work. In Section 3, we describe the RSA key generation procedure and the binary Euclidean algorithm as implemented in OpenSSL. In Section 4, we discuss the identified vulnerability and our key recovery attack on RSA. In Section 5, and Section 6, we outline our threat model and evaluate our attack in a real-world setting. In Section 7, we discuss existing countermeasures on an architectural level and we also propose a software patch to fix the identified vulnerability. Finally, we discuss further vulnerabilities in Section 8, and we conclude in Section 9.

2 BACKGROUND

In this section, we briefly introduce the concept of Intel SGX, and we discuss related work in terms of microarchitectural attacks against the RSA cryptosystem, both in standard settings on general-purpose computing platforms as well as in Intel SGX settings.

2.1 Intel SGX

Intel Software Guard Extensions (SGX) [17] provide hardware support for software to be executed isolated from the (untrusted) OS. Thereby, SGX reduces the trust assumption to the hardware only. Hardware-level encryption of memory ensures the confidentiality and integrity of code as well as data within an enclave. Irrespective of the privilege level, memory of enclaves cannot be accessed by software external to the enclave, not even by the OS itself. This policy is enforced by the CPU.

Although in case of Intel SGX the underlying OS need not be trusted, it still performs (security) critical tasks for enclaved programs. Among these tasks are the memory management including virtual-to-physical page mapping. To prevent misconfiguration of a running enclave by the OS, the CPU validates all management tasks that might affect enclave security [33]. Furthermore, enclaved programs share other system resources, such as the underlying hardware, with untrusted processes running on the same system. This makes them vulnerable to various kinds of side-channel attacks based on these shared resources.

[1] The patch is already merged upstream by OpenSSL.

2.2 Intel SGX SSL

The Intel SGX SSL library [18] is a cryptographic library for SGX enclaves. It is built on top of OpenSSL [20], a widely used toolkit for cryptographic purposes. Since Intel SGX SSL operates on OpenSSL, it inherits all of OpenSSL's side-channel properties including mitigation techniques but also potential vulnerabilities. In particular, OpenSSL employs several side-channel countermeasures to thwart traditional side-channel attacks such as cache attacks.

2.3 Microarchitectural Attacks on RSA

Aciiçmez [1] proposed the first attack exploiting the instruction cache (I-cache) to infer executed instruction paths taken by square and multiply operations in sliding window exponentiations. In a subsequent work, Aciiçmez and Schindler [4] attacked the extra reduction step of the Montgomery multiplication routine by exploiting the I-cache. Recently, Bernstein et al. [9] showed how to use knowledge of performed sliding window operations to infer private exponents.

Percival [37] proposed to monitor the square and multiply operations during the modular exponentiation of RSA by means of a technique that later became known as Prime+Probe [47]. In an effort to thwart cache-based attacks on the modular exponentiation, OpenSSL implemented a technique denoted as scatter-gather, which has been improved in [24, 26]. The idea of scatter-gather is to store fragments of sensitive data in multiple cache lines, such that the same cache lines are fetched irrespective of the accessed data elements. Yarom et al. [53] attacked the scatter-gather technique by exploiting cache-bank conflicts [8, 47], resulting in a sub-cache-line granularity attack. For a 4 096-bit RSA modulus they required 16 000 decryptions in order to recover the key.

Another procedure that has been attacked in the context of RSA (as well as ECDSA) is the modular inversion operation, *i.e.*, computing the inverse x of an element a modulo n such that $ax \equiv 1 \mod n$. Modular inversion operations are central to public key cryptography. Therefore, in the past, software implementations relied on an optimized variant of the extended Euclidean algorithm (EEA), namely the binary extended Euclidean algorithm (BEEA) [34, Algorithm 14.57]. Based on the observation that this optimized variant executes input-dependent (*i.e.*, secret-dependent) branches, Aciiçmez et al. [2] suggested to attack the modular inversion during RSA computations by means of branch prediction analysis (cf. [3]). They speculated that all branches of an attacked application can be monitored precisely, but did not implement the attack. At the same time, Aravamuthan and Thumparthy [5] pointed out that the BEEA is vulnerable to simple power analysis (SPA) attacks. Both attacks assumed the possibility to precisely distinguish between all branches taken in order to attack the modular inversion operation.

Later on, García and Brumley [22] suggested a Flush+Reload attack on the BEEA to attack the ECDSA implementation of OpenSSL.

García and Brumley implemented the proposed attack and recovered parts of the nonce values used in subsequent signature computations, which allowed them to recover the secret key. In order to mitigate these attacks, the OpenSSL procedure computing the modular inverse has been rewritten such that it prevents branches that leak sensitive information.

Side-Channel Attacks against RSA Key Generation. So far, side-channel attacks against RSA key generation routines relied on power analysis and targeted the prime generation procedure. For example, Finke et al. [19] performed a simple power analysis attack (SPA) on the prime generation procedure, *i.e.*, the sieving process, by assuming that the power consumption reveals the number of trial divisions before the Miller-Rabin [34, Algorithm 4.24] primality test is applied. Assuming that the prime candidates are incremented by a constant value in case of a failure, Finke et al. establish equations that allow to factor the modulus. Similarly, Vuillaume et al. [48] considered differential power analysis (DPA), template attacks, and fault attacks to attack the prime generation procedure. However, Vuillaume et al. consider the Fermat test [34, Algorithm 4.9], which is rarely used in practice due to false positives (Carmichael numbers). Bauer et al. [7] also attacked the prime sieve procedure during the prime number generation. All these side-channel attacks either target the primality test or the prime generation itself and cannot be executed by only running software on the targeted machine. They all require physical access.

Differentiation from Existing Attacks on Key Generation. The attack presented in this paper differs from previous attacks on RSA key generation as follows. First, contrary to related work which target the prime generation itself [48] or the primality tests [7, 19], we target the subsequent parameter checking routine. Second, previous attacks rely on power analysis while we use a purely software-based side channel. To the best of our knowledge, software-based microarchitectural attacks on the RSA key generation procedure have not been analyzed so far.

2.4 Attacks in SGX Settings

Currently, three types of side-channel attacks have been investigated against SGX enclaves, namely controlled-channel attacks, cache attacks, and branch prediction attacks. Controlled-channel attacks only allow monitoring data accesses and execution at page granularity (4 KB), but in a noiseless manner. Contrary, cache attacks enable a more fine-grained monitoring (e.g., 64 byte), but at the cost of measurement noise. Hence, there is a trade-off between granularity and measurement noise. Branch prediction attacks can distinguish single code branches on an instruction granularity.

Controlled-Channel Attacks. Controlled-channel attacks [51] (also referred to as pigeonhole attacks [43] or page-level attacks [50]) rely on the fact that the OS manages the mapping between virtual and physical pages for all processes, including processes executed inside hardware enclaves. Hence, the OS can modify the *present* bit for page table entries (PTEs), which allows the OS to cause page faults and to precisely monitor these page faults for an enclaved process that accesses the unmapped pages during its execution. Thus, the OS can observe the memory accesses or executed code paths of an enclave at page granularity. Instead of using the *present* bit, page faults can also be triggered by making pages non-executable [50] using the *non-executable* (NX) bit, or by setting a *reserved* bit [50, 51]. As before, this allows precise monitoring of page accesses.

Xu et al. [51] used controlled-channel attacks to extract sensitive data such as images and processed texts from enclaved programs. Shinde et al. [43] studied known information leaks in cryptographic primitives of OpenSSL and Libgcrypt with respect to page-level attacks. However, Shinde et al. did not identify the information leak exploited in this paper. Xiao et al. [50] used page-level attacks to mount Bleichenbacher and padding oracle attacks on various TLS implementations.

Previous page-fault based attacks could not monitor the execution of single instructions on a page. Hähnel et al. [27] and van Bulck et al. [12] relied on frequent timer interrupts of the Advanced Programmable Interrupt Controller (APIC) in order to read and clear the *accessed* bit of the PTE. This allows to even single-step page table accesses during enclave execution. As an example they suggested to attack a string comparison function, where the APIC interrupts the SGX enclave after every single memory access (byte granularity). Thereby, they are able to determine the length of the compared strings.

Cache Attacks. Since enclaves do not share memory with other processes or even the OS, Flush+Reload attacks [52] are not directly possible against enclaved programs. Nevertheless, other techniques such as Prime+Probe [37, 47] can be applied on enclaves. For example, Götzfried et al. [25] demonstrated a Prime+Probe attack by relying on the performance monitoring unit (PMU)[2] in order to precisely observe the number of cache hits and cache misses. They targeted an AES T-table implementation executed within an SGX enclave. Similarly, Moghimi et al. [35] demonstrated a Prime+Probe attack against AES T-table implementations running within SGX enclaves. Even though both works [25, 35] consider an all-powerful attacker who compromised the OS in order to minimize the influence of noise (e.g., scheduling the enclave on one specific core, etc.), they suffer from false positives and false negatives.

Brasser et al. [11] relied on Prime+Probe to attack the decryption process of an RSA implementation running inside an SGX enclave. Schwarz et al. [39] considered a slightly different attack scenario, where also the attack process runs inside an SGX enclave. They also relied on Prime+Probe to attack an RSA implementation running in a co-located SGX enclave. Although they extract 96% of a 4096-bit RSA key within a single trace, the number of remaining bits is still impractically high for a brute-force approach. Even worse, recovery suffers from random bit insertions and deletions at unknown positions. Hence, due to the measurement noise of Prime+Probe, several measurement traces need to be gathered in both attacks [11, 39].

Although Flush+Reload cannot be applied on enclaved programs directly, van Bulck et al. [13] proposed to use Flush+Reload to attack the page table entries (managed by the OS) in order to infer what pages have been accessed by the enclave. Thereby, they defeat countermeasures that aim to detect page faults [41, 43] or that mask the *accessed* and *dirty* flags of page table entries. However, their attack comes at the cost of an even coarser-grained granularity (32 KB) since one cache line holds eight PTEs.

[2]The PMU does not monitor performance metrics inside enclaves, but Götzfried et al. [25] probe their own memory accesses with the PMU.

Branch Prediction. Branch prediction represents a special type of cache attack that exploits the branch target buffer (BTB) cache in order to learn information about executed branches [2]. Lee et al. [32] observed that SGX does not clear the branch history when switching between enclave and non-enclave mode, which enables branch shadowing attacks. Branch shadowing represents an enhanced version of branch prediction analysis (cf. [3]), which relies on the last branch record (LBR) instead of RDTSC time measurements as well as APIC timer interrupts to increase the precision.

3 RSA KEY GENERATION IN OPENSSL

The RSA public key cryptosystem [38] provides public key encryption as well as digital signatures. The RSA key generation routine of OpenSSL—implemented in `rsa_gen.c`—starts by generating two primes p and q, which are then used to compute the public modulus $N = pq$. While p and q are chosen randomly during the key generation procedure, it is common practice that the public exponent is fixed to $e = 65\,537_{10} = \texttt{0x010001}_{16}$ (cf. [10]). The private key is later computed as $d \equiv e^{-1} \mod \phi(N)$, with ϕ being Euler's totient function. For two prime numbers p and q, $\phi(N) = \phi(p) \cdot \phi(q) = (p-1)(q-1)$.

Among other checks, the key generation routine ensures that $(p-1)$ and $(q-1)$ are coprime to e, *i.e.*, that the greatest common divisor (GCD) of the public exponent e and $(p-1)$ as well as $(q-1)$ is one. These checks are performed by relying on a variant of the Euclidean algorithm, which will be attacked in this paper.

3.1 Binary Euclidean Algorithm

A well-known algorithm to compute the GCD is the Euclidean algorithm [34, Algorithm 2.104]. For two positive integers $a > b$, it holds that $\gcd(a, b) = \gcd(b, a \mod b)$. Since this algorithm relies on costly multi-precision divisions, a more efficient variant is usually implemented for architectures with no dedicated division unit, using simple (and more efficient) shift operations and subtractions.

Listing 1 depicts an excerpt of the Euclidean algorithm as implemented in OpenSSL, which is an optimized version denoted as binary GCD [34, Algorithm 14.54] that has been introduced by Stein [44]. As can be seen in Listing 1, OpenSSL uses the BIGNUM implementation for arbitrary-precision arithmetic. The functionality of each BIGNUM procedure is indicated with comments.

The binary GCD works as follows. If b is zero, a holds the GCD and the algorithm terminates. Otherwise, the algorithm distinguishes the following cases in a loop.

(1) Branch 1 (Lines 7–10): If a and b are odd, the $\gcd(a, b) = \gcd((a-b)/2, b)$. The division by 2 (implemented as a right shift) accounts for the fact that the difference of two odd numbers is always even, but 2 does not divide odd numbers.
(2) Branch 2 (Lines 13–15) and 3 (Lines 20–22): If either a or b is odd, then the even number is divided by 2 through a right shift since 2 is not a common divisor.
(3) Branch 4 (Lines 25–27): If both a and b are even, then 2 is a common divisor and, therefore, both a and b are divided by 2. In this case the resulting GCD is a multiple of 2 and the variable s holds the number of times this branch is executed.

During the execution, the algorithm always ensures that $a > b$. It swaps a and b as soon as this condition is not satisfied anymore.

Listing 1: Binary GCD (a.k.a. Stein's algorithm) in OpenSSL.

```
1  BIGNUM *euclid (BIGNUM *a, BIGNUM *b) {
2    BIGNUM *t;
3    int s = 0;
4    while (!BN_is_zero(b)) {       // b != 0
5      if (BN_is_odd(a)) {
6        if (BN_is_odd(b)) {        // a is odd, b is odd
7          BN_sub(a, a, b);         // a = a-b
8          BN_rshift1(a, a);        // a = a/2
9          if (BN_cmp(a, b) < 0) {
10           t = a; a = b; b = t; // swap a and b
11         }
12       } else {                   // a is odd, b is even
13         BN_rshift1(b, b);        // b = b/2
14         if (BN_cmp(a, b) < 0) {
15           t = a; a = b; b = t; // swap a and b
16         }
17       }
18     } else {
19       if (BN_is_odd(b)) {        // a is even, b is odd
20         BN_rshift1(a, a);        // a = a/2
21         if (BN_cmp(a, b) < 0) {
22           t = a; a = b; b = t; // swap a and b
23         }
24       } else {                   // a is even, b is even
25         BN_rshift1(a, a);        // a = a/2
26         BN_rshift1(b, b);        // b = b/2
27         s++;
28       }
29     }
30   }
31
32   if (s)
33     BN_lshift(a, a, s);          // a = a * 2^s;
34   return a;
35 }
```

A Note on the Implementation. In the source code, the function BN_gcd(...)—used to compute the GCD—calls the function euclid(...) as depicted in Listing 1, but the compiler inlines the corresponding function into BN_gcd(...). Hence, in the remainder of this paper, we will refer to BN_gcd(...) when talking about the vulnerable code.

4 ATTACKING RSA KEY GENERATION

During RSA key generation, the binary GCD variant described in Section 3 is used to ensure that $p-1$ and e are coprime. In order to do so, the algorithm depicted in Listing 1 is executed with $a = p-1$ (with p being the secret prime) and $b = e$ (the public exponent). The crucial observation is that the binary GCD executes different branches depending on the input parameters. An attacker who is able to observe the executed branches can recover the secret input value $a = p-1$ and, hence, the secret prime factor p.

Without loss of generality, we describe the attack by targeting the prime factor p, but the presented attack can also be applied to recover the prime factor q. Once we recovered either of the two prime factors, N can be factored trivially, which also allows to compute the private exponent d.

4.1 Idealized Attacker

For the sake of completeness we first consider an attacker who can precisely distinguish all executed branches of the binary GCD algorithm, including the swapping operations in lines 10, 15, and 22. This, for example, accounts for branch shadowing attacks [32] or the generalized attack described in Section 4.4.

Let a be the unknown secret input to be recovered, b the known input, and $a_i, b_i, i \geq 0$ all intermediate values calculated by the algorithm. To recover the secret input a, we build a system of linear equations, starting with $a = a_0$ and $b = b_0$. We then iteratively add equations, depending on the executed branches, as follows.

First branch: $a_{i+1} = \frac{a_i - b_i}{2}$
Second branch: $b_{i+1} = \frac{b_i}{2}$
Third branch: $a_{i+1} = \frac{a_i}{2}$
Fourth branch: $a_{i+1} = \frac{a_i}{2}$ and $b_{i+1} = \frac{b_i}{2}$

We increment i by one before proceeding with the next iteration. In addition, if a and b are swapped, *i.e.*, BN_cmp(a, b) < 0 yields true, we add the following two equations and increment i again: $a_{i+1} = b_i$ and $b_{i+1} = a_i$. The algorithm finishes after n steps with $a_n = \gcd(a, b)$ and $b_n = 0$. By recursively substituting all equations one can express the unknown a as a linear equation $a = f(a_n, b_n) = f(\gcd(a, b), 0)$, which is trivial to solve, given that $\gcd(a, b)$ is known to be 1 in case of valid RSA parameters.

4.2 Page-level Attacker

Although considering a powerful attacker who is capable of distinguishing all branches is a realistic assumption [32], we resort to a weaker assumption in the rest of this paper. We consider a page-level attacker [43, 51], who recovers the secret input a from even less observations (up to the point where the two variables are swapped) and with a coarser-grained granularity (page level).

Figure 1 illustrates an excerpt of the control flow of the binary GCD for the four important branches being executed and, for illustration purposes, also the mapping of specific functions to their corresponding code pages.[3] If an attacker can distinguish executed branches based on page-access observations, the Euclidean algorithm can be reverted and the secret input a can be recovered. Indeed, the functions BN_sub(...) and BN_rshift1(...) reside on different pages within the memory, denoted as page 1 and page 4, while BN_gcd(...) is on page 2.

Observations. If this algorithm is executed with RSA parameters ($a = p - 1$ and $b = e$), we observe the following:

(1) Since p is a prime number, $p - 1$ is even. Hence, in the first iteration, the first parameter ($a = p - 1$) is always even and the second parameter ($b = e$) is always odd, as otherwise the GCD of $p - 1$ and e cannot be 1 as required for valid RSA parameters.
(2) The execution of the first branch can be observed by consecutive accesses to the corresponding code pages of BN_sub(...) and BN_rshift1(...).
(3) The second or the third branch are executed if either a or b is odd. These two branches, however, cannot be distinguished based on code page accesses since both branches execute the functions BN_rshift1(...) and BN_cmp(...) in the same order. Nevertheless, recall that in our setting the algorithm is always executed with an odd $b = 65\,537$, which is much smaller than a. Thus, in the beginning, the algorithm will only execute the third (and the first) branch, reducing the value of a_i, but b_i remains an unchanged odd value. This is true until a_i and b_i are swapped for the first time, which is the case if $a_i < b_i$. Since each iteration reduces a_i by one bit (in general) due to the right shift operation, the first swap will approximately occur after $\log_2(p-1) - \log_2(e)$ iterations. Until then, every time we observe a single access to code page 4 we can be sure that branch 3 has been executed.
(4) The fourth branch will only be executed if the greatest common divisor of the parameters a and b is a multiple of 2. Since the parameter $a = p - 1$ is even and $b = e$ is odd, this branch will never be executed (indicated as a red branch), as otherwise we would have invalid RSA parameters.
(5) The end of a branch and the start of the next iteration can be detected by monitoring accesses to BN_gcd(...) on page 2.
(6) Although a page-level attacker is able to observe when the BN_cmp(...) function is executed, our restricted page-level attacker cannot decide whether or not the variables are swapped (*i.e.*, whether or not the conditional branch depending on the result of BN_cmp(...) is executed).[4] This is because the corresponding code for swapping the two numbers is on the same page as the binary GCD algorithm BN_gcd(...) itself. More specifically, our page-level attacker cannot decide whether BN_gcd(...) directly continues with the next iteration, or whether a and b are being swapped first.

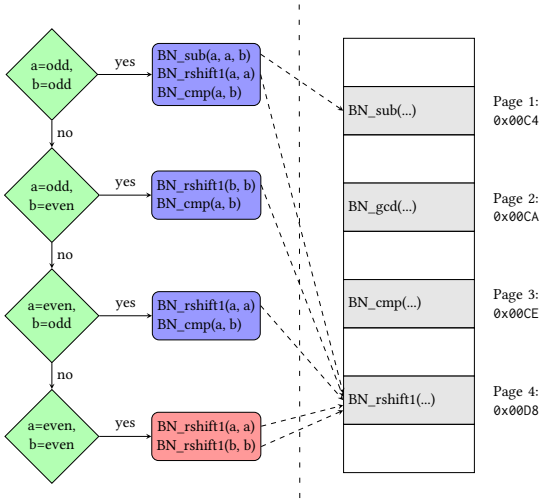

Figure 1: Excerpt of relevant control flow of binary GCD (left) and page layout (right)

These observations combined with the fact that the public exponent e is known allow us to "revert" the computations for all bits of $a = p - 1$, except about $\log_2(e)$ bits. As mentioned before, the public exponent is fixed to $e = 65\,537$.[5] This means that about $\log_2(65\,537) \approx 16$ bits of $a = p - 1$ cannot be recovered based on the accessed code pages. However, they can be easily determined based on the relations established from these observations.

[3] The mapping depicts the actual offsets of the most recent commit 899e62d of OpenSSL 1.1.0g.

[4] See Section 4.4 for a more generalized page-level attacker.

[5] This choice of the public exponent has been widely established as quasi-standard among RSA cryptosystems (cf. [10]).

As mentioned, the functions BN_sub(...) and BN_rshift1(...) reside on different pages within the memory. In our tested implementation, they are even 20 pages apart. Thus, it is very unlikely that a different compiler setting would link them to the same page, which would make them indistinguishable to a page-level attacker monitoring these functions only. Even if this would happen, one could easily distinguish them by monitoring the sub-functions which are called by BN_sub(...) but not by BN_rshift1(...), *i.e.*, BN_wexpand(...), BN_ucmp(...), BN_usub(...), etc.

4.3 Exploiting the Information Leak

We denote the sequence of page accesses observed by an attacker as $P = (p_0, \ldots, p_n)$. Without loss of generality, let us assume the same mapping from functions to code pages as in the previous example. For instance, the function BN_sub(...) resides on page 1 (`0x00C4`), BN_gcd(...) resides on page 2 (`0x00CA`), and the function BN_rshift1(...) resides on page 4 (`0x00D8`). That is, the sequence of page accesses consists of pages $p_i \in \{P_1, P_2, P_4\}$ since we are only interested in these page accesses.

In order to recover the prime factor p (or $p-1$ respectively), we observe a sequence of page accesses up to the point where the two variables are swapped for the first time. All later page accesses are discarded. We denote this number of iterations as m. Given the modulus N or its bit size $\log_2(N)$, we denote the bit size of p and q as $K = \log_2(N)/2$. Thus, m is upper-bounded by $\lceil K - \log_2(e) \rceil$. Similar as before, we build a system of linear equations based on a_i, starting with the unknown input $a = a_0$. Since $i < m$, b will remain unchanged and we only need to distinguish two branches:

Access to page 1, and page 4: $a_{i+1} = \frac{a_i - b}{2}$
Access to page 4: $a_{i+1} = \frac{a_i}{2}$

Accesses to page 2 allow to distinguish iterations. After m iterations, we express these equations by recursive substitution as a linear equation $a = f(a_m, b)$, or, more precisely

$$a = a_m \cdot c_a + b \cdot c_b$$

with known constants c_a and c_b, which result from the substitution.

Both, a and a_m are unknown. However, we additionally know that swapping occurred after m iterations, *i.e.*, $a_m < b$. Hence, we can determine the correct a by iterating over values $a_m \in [1, e)$ and evaluating the above equation. We use the resulting values a to check the GCD of $(a+1)$ and N. In case the GCD is greater than 1, we recovered a as well as the corresponding prime factor $p = a + 1$. We can then factor the modulus N by computing $q = N/p$.

As mentioned before, the iteration counter m is upper-bounded by $\lceil K - \log_2(e) \rceil$ with K being the bit size of the prime numbers. This is because each iteration reduces a_i by at least one bit due to the right shift operation. For example, a 4 096 bit RSA key will have prime numbers of length $K = 2\,048$ bits, yielding $m = 2\,032$ iterations to consider. However, a prime number which is closer to 2^{K-1} than to 2^K combined with the subtraction in branch 1 could reduce a_i by one additional bit. This would make swapping occur one iteration earlier. We would erroneously consider an incorrect equation due to swapping and determining the correct a might fail. In this case, we simply omit the last erroneous equation a_m from the recursive substitution and try to determine a again by iterating over values $a_{m-1} \in [1, e)$. As we will see in Section 6, this happens in approximately 25% of all runs, meaning that about 75% of the generated RSA keys can be recovered in the first run.

In case $p - 1$ is not coprime to e—which is the reason why the binary GCD algorithm is executed—the RSA key generation will discard this prime factor candidate p and re-generate another prime factor candidate p. Nevertheless, by observing the page fault pattern, an attacker is also able to detect this (extremely rare) case, and we run the same attack on the newly generated p.

Example. For an illustrative example let us assume the following hypothetical parameters. Let the public exponent be $e = 17 = \texttt{0x11}_{16}$ and let the two 14-bit primes be $p = 11083 = \texttt{0x2B4B}_{16}$, and $q = 9941 = \texttt{0x26D5}_{16}$, respectively, and $N = pq$. In the course of validating the selected parameters, the OpenSSL implementation calls the binary GCD function with $a = 11082$ and $b = 17$. Table 1 illustrates the executed operations for the given input parameters a and b. In the first loop iteration, a is even and b is odd, which means that the function BN_rshift1(...) will be called. In the second loop iteration, a is odd and b is odd, which means that BN_sub(...) followed by BN_rshift1(...) will be executed, and so on. Finally, the algorithm returns 1 as the GCD of $a = 11082$ and $b = 17$.

Based on a controlled-channel attack, we are able to observe accesses to pages P_1, P_2, and P_4, and to precisely recover the executed operations up to the point where a and b are swapped. We recursively substitute the recovered operations on a_i, which leads to the equations shown in the last column of Table 1. Recall that the first swap will happen *at latest* after $m = \lceil 14 - \log_2(17) \rceil = 10$ iterations. In our example, swapping is done already in iteration 9 due to a smaller p and additional subtractions. This leads to the erroneously recovered operation marked bold (and colored red) in Table 1. To recover the secret a, we start with the m-th substituted equation a_{10}, not knowing that it is erroneous. If the attempt to recover a based on a_{10} fails, we would need to fall back to equation a_9. However, in this particular case the error cancels out and we already succeed with a_{10}. Recall that $a_{10} = \frac{a}{1024} - \frac{85b}{512}$. With $b = 17$, we can rearrange it to

$$a = 1024 \cdot a_{10} + 2890 \tag{1}$$

The unknown variable a_{10} is bound by the parameter b. Since a and b have been swapped, a_{10} must be smaller than b. We try to solve this equation by iterating over $a_{10} \in [1, b)$ and checking the GCD of $a + 1$ and N. If the GCD is greater than 1, we are able to factor N. Indeed, for $a_{10} = 8$ the equation yields $a = 11\,082$ and $\gcd(a + 1, N) > 1$. Thus, we recovered the first prime $p = 11\,083$, which allows to factor N ($q = N/p = 9941$) and to recover the secret exponent $d \equiv e^{-1} \mod (p-1)(q-1)$. To see why recovery on the erroneous equation a_{10} works in this case, we compare it to the valid equation $a_9 = \frac{a}{512} - \frac{85b}{256}$, which can be rewritten as

$$a = 512 \cdot a_9 + 2890 \tag{2}$$

Here, recovering a succeeds for $a_9 = 16$. Observe that in equations (1) and (2) the first constants are only off by a factor of 2 because the erroneous operation does not introduce a subtraction but only a right shift. Hence, we hit the correct guess with $a_{10} = a_9/2 = 8$.

4.4 Generalization

The proposed attack on RSA key generation is not limited to code pages only. One could also monitor accesses to data pages, especially

Table 1: Executed and recovered operations when calling BN_gcd(...) for $a = 11082$ and $b = 17$.

a	b	Performed operation	$a < b$ swapping	Page observation	Recovered operation	Substituted equation
0010 1011 0100 1010	0001 0001	$a_{i+1} = \frac{a_i}{2}$	no	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_1 = \frac{a}{2}$
0001 0101 1010 0101	0001 0001	$a_{i+1} = \frac{a_i - b_i}{2}$	no	P_1, P_2, P_4, P_2	$a_{i+1} = \frac{a_i - b}{2}$	$a_2 = \frac{a}{4} - \frac{b}{2}$
0000 1010 1100 1010	0001 0001	$a_{i+1} = \frac{a_i}{2}$	no	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_3 = \frac{a}{8} - \frac{b}{4}$
0000 0101 0110 0101	0001 0001	$a_{i+1} = \frac{a_i - b_i}{2}$	no	P_1, P_2, P_4, P_2	$a_{i+1} = \frac{a_i - b}{2}$	$a_4 = \frac{a}{16} - \frac{5b}{8}$
0000 0010 1010 1010	0001 0001	$a_{i+1} = \frac{a_i}{2}$	no	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_5 = \frac{a}{32} - \frac{5b}{16}$
0000 0001 0101 0101	0001 0001	$a_{i+1} = \frac{a_i - b_i}{2}$	no	P_1, P_2, P_4, P_2	$a_{i+1} = \frac{a_i - b}{2}$	$a_6 = \frac{a}{64} - \frac{21b}{32}$
0000 0000 1010 0010	0001 0001	$a_{i+1} = \frac{a_i}{2}$	no	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_7 = \frac{a}{128} - \frac{21b}{64}$
0000 0000 0101 0001	0001 0001	$a_{i+1} = \frac{a_i - b_i}{2}$	no	P_1, P_2, P_4, P_2	$a_{i+1} = \frac{a_i - b}{2}$	$a_8 = \frac{a}{256} - \frac{85b}{128}$
0000 0000 0010 0000	0001 0001	$a_{i+1} = \frac{a_i}{2}$	yes → swap	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_9 = \frac{a}{512} - \frac{85b}{256}$
0000 0000 0001 0001	0001 0000	$\mathbf{b_{i+1} = \frac{b_i}{2}}$	no	P_4, P_2	$\mathbf{a_{i+1} = \frac{a_i}{2}}$	$\mathbf{a_{10} = \frac{a}{1024} - \frac{85b}{512}}$
⋮	⋮	⋮	⋮	discard		
0000 0000 0000 0001	0000 0000	Return a as the GCD				

those on which the heap buffers a and b reside. If a and b are located on different heap data pages, we can distinguish which of these buffers is accessed and, thus, which arguments are provided to the BIGNUM functions. This allows to distinguish all relevant branches, enabling the idealized attack described in Section 4.1. For example, one can distinguish branch 2 and 3 based on the input of BN_rshift1(...) in lines 13 (accessing b only) and line 20 (accessing a only) of Listing 1. Also, one can detect swapping of a and b, after which their pointers map to the opposite page, respectively. For example, if BN_is_zero(...) in line 4 accesses buffer a instead of b, or the call to BN_cmp(...) (line 9, 14 or 21) accesses b *before* a, one can infer that swapping occurred in the previous iteration. Thus, one could derive equations over all iterations and recover the key without the need for guessing values for $a_m \in [1, e)$.

Even if a and b are located on the same heap page, attacks might still be possible by carefully crafted user input that also gets copied onto the heap and, thus, shifts the targeted buffers a and b onto different heap pages. We did not investigate such generalized attacks further, since our attack already recovers the full key by monitoring page faults up to the point where a and b are being swapped.

5 THREAT MODEL AND ATTACK SCENARIO

In order to exploit the identified vulnerability, we consider an enclave that dynamically generates RSA keys, which are intended to never leave the enclave. Dynamic key generation has already broad applications in other trusted execution environments, such as trusted platform modules and smart cards. In line with SGX's threat model, the operating system (OS) is considered untrusted and compromised, trying to extract secret keys from the enclave. Although, in general, attackers in SGX settings are considered to be able to trigger enclave operations arbitrarily often by repeatedly invoking the enclave with a fresh state,[6] our attacker is naturally limited to at most one observation of the enclave's key generation, as the next invocation will generate a different, independent key. Using a noiseless controlled-channel attack [43, 50, 51], the attacker can observe page access patterns of the executing enclave.

While this is sufficient for the attack presented in this paper, we note that, without loss of generality, an attacker could also resort to different techniques. Among them are side channels using branch shadowing [32] or single-step approaches based on the APIC timer interrupts [12, 27] or even attacks with fewer or no page faults [13, 49], given that enough information can be extracted from a single execution.

Attack Scenarios. Dynamic key generation is a fundamental operation for most SGX applications. For example, scenarios like audio and video streaming with SGX [28] fall into our threat model. Here, a streaming enclave dynamically generates an RSA key pair and registers the public key at its streaming counterpart. Latter delivers all streaming content encrypted under this key, allowing the enclave to securely decrypt it and to display it to the user, all in the sphere of a possibly compromised OS. Another example is a document signing enclave, generating its own signature keys inside the enclave and issuing a certificate signing request to an external certification authority. Thereby, the enclave protects the signing key against malware. In any case, compromise of the private key could lead to signature forgery, espionage or video piracy with all its legal and financial consequences.

6 ATTACK EVALUATION

We evaluate the presented attack on an Intel Core i7-6700K 4.00 GHz platform running Ubuntu 17.10 (Linux kernel 4.13.0-37). In order to do so, we developed an SGX application that generates an RSA key based on the latest version of Intel SSL SGX.[7] We used the Linux Intel SGX software stack v1.9, consisting of the Intel SGX driver, the Intel SGX software development kit (SDK), and the Intel SGX platform software (PSW).[8] For controlling the page mapping, we used the SGX-Step kernel module as well as the corresponding SGX-Step library functions (cf. [12]). Note that we do not use the

[6] SGX does not protect against rolling back to a fresh state. This would require external persistent storage [45].

[7] We relied on the most recent commit 654f94d of Intel SSL SGX, which in turn is based on OpenSSL version 1.1.0g (https://www.openssl.org/source/openssl-1.1.0g.tar.gz).

[8] https://github.com/01org/linux-sgx.

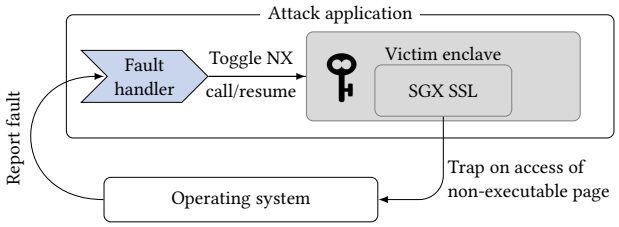

Figure 2: Basic principle of the performed attack.

single-stepping feature of SGX-Step but rather its page mapping capability. Since Intel SGX considers an untrusted OS, the application of SGX-Step is in line with the threat model. We describe the implementation details below.

6.1 Implementation Details

We consider a victim enclave using the Intel SGX SSL library to generate an RSA key pair. The enclave is hosted by a malicious attack application that interacts with the OS to manipulate page mappings and to record page accesses within the corresponding fault handler. Figure 2 depicts the principle of the attack. After this recording step, the collected trace of page accesses is evaluated to recover the secret key.

SGX Enclave Application (Victim Enclave). We developed an enclave program that generates a single RSA key using the Intel SGX SSL library and outputs the public parts only, *i.e.*, the modulus N. Therefore, we implemented an ECALL function for invoking key generation and an OCALL function which prints the modulus of the generated key to the standard output. Recall that the public exponent is fixed to $e = 65\,537$. The project is built in pre-release hardware mode, *i.e.*, it uses the same compiler optimizations as a production enclave in release mode and yields the same memory layout. Without loss of generality, the enclaved program does not perform any other tasks apart from generating the RSA key.

Attack Application. Based on the SGX-Step framework [12], we developed an attack application that enables and disables executable regions (pages) of the enclave program. Therefore, it toggles the NX bit of the page table entries belonging to the code pages to be traced. Without loss of generality, one could also use the present bit or a reserved bit [50, 51] for the same purpose. The application registers a fault handler (via a `sigaction` standard library function call) which is executed whenever the enclave encounters a segmentation fault (due to a non-executable page). This fault handler conveniently serves as the basis to monitor page faults, which later on allow to recover the executed code paths.

6.2 Mounting the Attack

In order to determine the pages of interest, *i.e.*, the ones where the BN_gcd(...), BN_sub(...), and BN_rshift(...) functions are located, we dissect the enclave binary by means of `objdump`. In our case, `objdump` reveals the following page frame numbers: `0x00CA` for BN_gcd(...), `0x00C4` for BN_sub(...), and `0x00D8` for BN_rshift1(...). When starting the victim enclave, the attack application disables the execution of the BN_gcd(...) page by setting the non-executable (NX) bit in the corresponding page table entry. This causes the enclave to trap as soon as it attempts to execute this page.

When the fault handler function is executed for the first time, *i.e.*, when a page fault (segmentation fault) occurs, we start recording subsequent page faults. On the one hand, we enable execution of the current page which caused the page fault by clearing its NX bit in order to allow the enclave to continue. On the other hand, we also disable the other pages of interest by setting their NX bits. Whenever the page fault handler is triggered, we record the accessed page and toggle the non-executable bits accordingly. Thus, we are able to precisely monitor each access to these pages.

Our practical evaluation confirmed that we observe the following page fault patterns. Executing branch 1 leads to consecutive page faults on `0x00C4` (BN_sub(...)) and `0x00D8` (BN_rshift1(...)), interleaved with page faults on `0x00CA` (BN_gcd(...)), whereas executing branch 3 leads to a page fault on `0x00D8` (BN_rshift1(...)) only. When the attack application finished gathering the page faults, we process the page-fault sequence from left to right and build up an equation system according to the rules established in Section 4.3. That is, whenever we observe consecutive page accesses to page `0x00C4` and page `0x00D8`, we add $a_{i+1} = (a_i - b)/2$, while for a single access to page `0x00D8` we add $a_{i+1} = a_i/2$. Based on these equations we run a SageMath script in order to recursively substitute the equations, recover the remaining bits by solving the equation for a_m, and finally to recover the RSA private key.

The execution time of the whole attack including the gathering of the page-fault trace as well as the parsing of the gathered trace is negligible, even when attacking larger RSA keys. Causing page faults on the above mentioned pages slightly increases runtime and gathering the page-fault traces terminates immediately. Compared to normal key generation, running the attack causes moderate overall slowdowns of 65 ms (15,5%) for 4,096 bit keys and 248 ms (5,87%) for 8,192 bit keys due to the intentionally induced page faults. The biggest share of the execution time is consumed by the generation of the two random primes, *i.e.*, the random number generation and the primality test, during RSA key generation.

6.3 Key Recovery Complexity

We developed a simple script for SageMath[9] that iterates over all possible values for $1 \leq a_m < 65\,537$, evaluates $a = f(a_m)$, and checks the GCD of $a + 1$ and N. If it is not equal to 1, p can be recovered.

Figure 3 illustrates the complexity for the task of recovering the remaining bits. The complexity has been averaged over 100 runs per modulus size and the computations are evaluated with SageMath on an Intel Xeon E5-2660 v3 (2.60GHz). The area plot (right x-axis) indicates that in about 75%–80% of all cases, the prime factors can be recovered at the first attempt, considering $m = \lceil K - \log_2(e) \rceil$ equations. In only about 20%–25% of all cases the first attempt fails due to an early swapping in the binary GCD algorithm. In this case, we need to remove the last equation a_m and restart the search in the range $1 \leq a_{m-1} < 65\,537$. The asymptotic complexity of the key recovery is $O(1)$. This means that the number of iterations is bound by the public exponent e, which is a constant value. In contrast, the computation time of the GCD for candidates a increases due to

[9] http://www.sagemath.org/

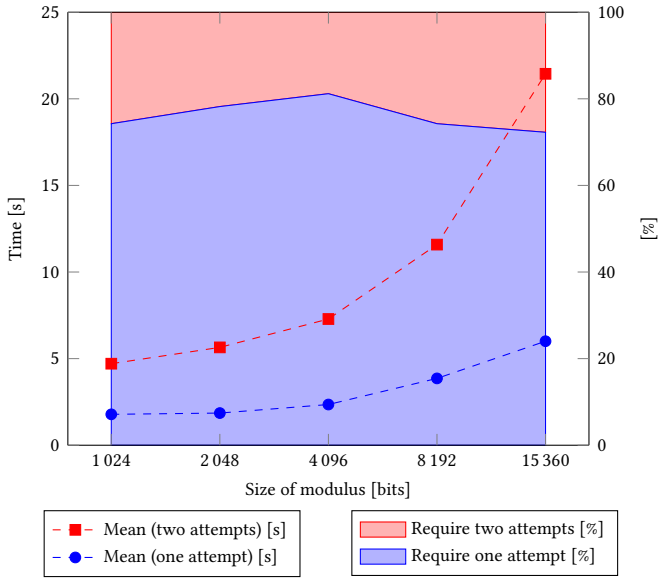

Figure 3: Key recovery complexity for different bit sizes of the modulus N.

the larger bit sizes of the modulus N. In 75% of all cases, a 8 192-bit modulus can be factored in less than 5 seconds on average, after gathering the measurement trace. In only 25% of all cases, we need approximately 12 seconds on average. Although 15 360-bit RSA keys (providing 256-bit security according to NIST [36]) are currently not being used in practice, we provide the results here for the sake of completeness.

7 COUNTERMEASURES

Architectural Countermeasures. In order to mitigate controlled-channel attacks, various architectural countermeasures have been proposed. Shinde et al. [43] introduced the notion of page-fault obliviousness, which means that the OS is still able to observe page faults, but the observable page-fault pattern is independent of the input and the executed code paths. They proposed a software-based approach incurring a significant performance overhead. This can be reduced by additional hardware support which guarantees to deliver page faults directly into the enclave [42]. Another proposal denoted as SGX-LAPD [21] considers large pages (*i.e.*, 2 MB instead of the usual 4 KB) in order to reduce the overall number of page faults. The enclave relies on the `EXINFO` data structure, which tracks page fault addresses of an enclave, to verify that the OS indeed provides large pages. Their solution is based on a dedicated compiler as well as a linker in order to generate the corresponding code for large-page verification inside enclaves. Strackx et al. [46] propose hardware modifications allowing to preload all critical page mappings in the translation lookaside buffer (TLB) whenever entering the enclave. Moreover, they protect the TLB mapping from being tampered during enclave execution.

Detect Frequent Page Faults. Shih et al. [41] observed that transactional synchronization extensions (TSX) can be used to detect exceptions such as page faults and report them to enclave-internal code only, rather than to the OS. They proposed T-SGX, in which they execute blocks of enclave code inside TSX transactions. If an exception is thrown, the transaction aborts and the enclave decides whether or not to terminate its execution. Chen et al. [14] proposed an alternative approach to detect side-channel attacks within enclaves, *i.e.*, detecting frequent page faults and aborting the execution. In order to so, they rely on the execution time within the enclave as an indicator of an ongoing side-channel attack. Since timers are also accessed through the untrusted OS, they implement a reference clock inside the enclave. The reference clock itself (a timer variable) is protected by means of TSX.

Detecting page faults does not prevent stealthier attacks that come without the need for page faults [13, 49]. These attacks derive page access patterns either by monitoring the *accessed* and *dirty* bits of page table entries or by mounting cache-attacks like Flush+Reload attacks on page table entries.

Randomization. Seo et al. [40] propose SGX-Shield which randomizes the memory layout of enclaves in a multi-stage loading step. While primarily intended as a countermeasure against runtime attacks, it also raises the bar for controlled-channel attacks.

Prevent Input-Dependent Code Paths. The most straightforward approach to prevent the attack described in this work is to fix the RSA key generation procedure at the implementation level. We propose an appropriate patch in the following subsection.

7.1 Patching OpenSSL

Listing 2 shows our proposed patch for OpenSSL. Instead of relying on BN_gcd(...) to ensure that $p-1$ and e are coprime, *i.e.*, that the GCD of $p-1$ and e is one, we compute the modular inverse of $p-1$ modulo e using a side-channel protected modular inversion algorithm (BN_mod_inverse(...)). The inverse only exists if $\gcd(p-1, e) = 1$. Hence, if BN_mod_inverse(...) signals (through an error) that the inverse does not exist, we know that $\gcd(p-1, e) \neq 1$.

Listing 2: Patch for RSA key generation in OpenSSL.

```
diff --git a/crypto/rsa/rsa_gen.c b/crypto/rsa/rsa_gen.c
index 4ced965..4051933 100644
--- a/crypto/rsa/rsa_gen.c
+++ b/crypto/rsa/rsa_gen.c
@@ -41,6 +41,7 @@ static int \
 rsa_builtin_keygen(RSA *rsa, int bits, BIGNUM *e_value,
 {
     BIGNUM *r0 = NULL, *r1 = NULL, *r2 = NULL, \
            *r3 = NULL, *tmp;
     int bitsp, bitsq, ok = -1, n = 0;
+    unsigned long error = 0;
     BN_CTX *ctx = NULL;

     /*
@@ -88,16 +89,25 @@ static int \
 rsa_builtin_keygen(RSA *rsa, int bits, BIGNUM *e_value,
     if (BN_copy(rsa->e, e_value) == NULL)
         goto err;

+    BN_set_flags(rsa->e, BN_FLG_CONSTTIME);
+
     /* generate p and q */
     for (;;) {
         if (!BN_generate_prime_ex(rsa->p, bitsp, 0, \
             NULL, NULL, cb))
             goto err;
         if (!BN_sub(r2, rsa->p, BN_value_one()))
```

```
             goto err;
-        if (!BN_gcd(r1, r2, rsa->e, ctx))
-            goto err;
-        if (BN_is_one(r1))
-            break;
+        // Inverse only exists if GCD = 1
+        if (BN_mod_inverse(r1, r2, rsa->e, ctx))
+            break; // GCD = 1
+        else {
+            error = ERR_peek_last_error();
+            if (ERR_GET_LIB(error) == ERR_LIB_BN &&
+                ERR_GET_REASON(error) == BN_R_NO_INVERSE)
+                    ERR_clear_error(); // GCD != 1
+            else
+               goto err; // Another error occurred
+        }
         if (!BN_GENCB_call(cb, 2, n++))
             goto err;
     }
@@ -110,10 +120,17 @@ static int \
 rsa_builtin_keygen(RSA *rsa, int bits, BIGNUM *e_value,
         } while (BN_cmp(rsa->p, rsa->q) == 0);
         if (!BN_sub(r2, rsa->q, BN_value_one()))
             goto err;
-        if (!BN_gcd(r1, r2, rsa->e, ctx))
-            goto err;
-        if (BN_is_one(r1))
-            break;
+        // Inverse only exists if GCD = 1
+        if (BN_mod_inverse(r1, r2, rsa->e, ctx))
+            break; // GCD is 1
+        else {
+            error = ERR_peek_last_error();
+            if (ERR_GET_LIB(error) == ERR_LIB_BN &&
+                ERR_GET_REASON(error) == BN_R_NO_INVERSE)
+                    ERR_clear_error(); // GCD != 1
+            else
+                goto err; // Another error occurred
+        }
         if (!BN_GENCB_call(cb, 2, n++))
             goto err;
     }
```

In order to ensure that the side-channel protected implementation of the inversion algorithm is called, we need to set the BN_FLG_CONSTTIME flag on the public modulus e. This ensures that BN_mod_inverse(...) internally calls the protected function BN_mod_inverse_no_branch(...), which does not contain branches that leak sensitive information.

Performance Impact. An appealing benefit of our proposed patch is that it is even faster than the vulnerable implementation.[10] We benchmarked 10 000 coprimality checks for a random number a and $e = 65\,537$, and provide the corresponding cumulative execution times in Table 2. As can be seen in the table, our patch is by one to two orders of magnitudes faster than the original implementation on our test machine. On an Intel Core i7-5600U 2.6 GHz CPU (notebook), the speedup exceeds even a factor of 500 for 8 192 bit numbers. The reason for this massive speedup is that inversion, as implemented in OpenSSL, uses the original Euclidean algorithm with $\gcd(a, b) = \gcd(b, a \mod b)$. This algorithm requires far less loop iterations (e.g., between 5 and 13 iterations for 8 192-bit numbers) than the binary GCD (≈ 8192 iterations). The original Euclidean algorithm relies on a costly modular reduction in each iteration, which was the initial motivation to use the binary GCD instead, which avoids these costly modular reductions. Yet, the original Euclidean algorithm is in fact significantly faster because OpenSSL leverages the x86 `div` instruction to perform the expensive modular reductions directly in hardware.

[10] Note that we do not need to compute the GCD but only check whether or not it is 1.

Table 2: Performance comparison for 10 000 runs on an Intel Core i7-6700K (upper half) and an i7-5600U (lower half).

Bit size of a	BN_gcd(a, e)	BN_mod_inverse(a, e)
1 024	0.25 s	0.02 s
2 048	0.69 s	0.03 s
4 096	2.07 s	0.03 s
8 192	6.97 s	0.05 s
1 024	1.18 s	0.03 s
2 048	3.78 s	0.04 s
4 096	14.04 s	0.06 s
8 192	54.64 s	0.10 s

Nevertheless, the performed check whether the $\gcd(p-1, e) \neq 1$ handles a corner case in RSA key generation, which is highly unlikely to happen in practice. Hence, the corresponding check is in general only executed once per generated prime factor and, thus, two times during the RSA key generation.

8 FURTHER VULNERABILITIES

RSA X9.31. Further investigation of the OpenSSL source code revealed that the prime derivation function based on the ANSI X9.31 standard [29] (BN_X931_derive_prime_ex(...)) is also vulnerable to the presented attack. Similar as in the default RSA key generation procedure implemented in `rsa_gen.c`, the generated primes p and q are verified, *i.e.*, that $p-1$ and $q-1$ are coprime to the public modulus e. Hence, the exact same attack technique also applies to the X9.31 implementation. Irrespective of whether or not this implementation is actually used (ANSI X9.31 has already been withdrawn in [6]), we suggest to patch this implementation. The patch presented in Section 7 also applies here.

Furthermore, there are two additional usages of the vulnerable BN_gcd(...) function, namely in RSA_X931_derive_ex(...) and RSA_check_key_ex(...). In these cases, the GCD is not used as mere security check but to factor out the GCD of the product $(p-1)(q-1)$. Since the calculated GCD is never 1, our patch using the inversion algorithm cannot be applied here. Instead, we suggest to add a constant time implementation of the GCD algorithm, which is resistant against software side-channel attacks. Ideally, this implementation is even faster than the binary GCD implementation (cf. the performance analysis of our proposed patch in Section 7).

RSA Blinding. While our attack highlights a critical vulnerability in RSA key generation, other algorithms also need careful evaluation with respect to single-trace attacks. For example, we found a vulnerability in the generation of RSA blinding values used to thwart side-channel attacks on sensitive RSA exponentiation. The vulnerability causes preparation of the blinding value to fall back to an exponentiation implementation vulnerable to side-channel attacks. Similar to the attack presented in this paper, a controlled-channel attacker could attempt to recover the blinding value from a single trace and subsequently peel off the side-channel protection offered by blinding. The OpenSSL team fixed this issue in response to our findings by using the side-channel protected exponentiation algorithm appropriately.

8.1 Responsible Disclosure

We responsibly notified Intel as well as OpenSSL about our findings and provided a patch to fix the RSA key generation, as shown in Listing 2. In response, OpenSSL patched the RSA key generation vulnerability in commit `8db7946e`. Also, the RSA blinding vulnerability was fixed in commit `e913d11f`.[11]

9 CONCLUSION

In this paper, we investigated the RSA key generation routine executed inside SGX enclaves under the aspect of microarchitectural side-channel attacks. Our investigations revealed a critical vulnerability inside Intel SGX SSL that allows to recover the generated RSA secret key with a single observation using a controlled-channel attack. More specifically, the observable page fault patterns during the RSA key generation allow to recover the prime factor p and, thus, to factor the modulus N. To the best of our knowledge, this represents the first microarchitectural attack targeting the RSA key generation process by means of a software-based attack.

Ironically, the vulnerability is due to an optimized binary GCD algorithm that should improve the performance compared to the original Euclidean algorithm but in fact is significantly slower on Intel x86 platforms. Nevertheless, our work demonstrates that software-based microarchitectural attacks on shielded execution environments such as Intel SGX represent a severe threat to key generation routines and need further consideration.

ACKNOWLEDGMENTS

This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET — Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

REFERENCES

[1] Onur Aciiçmez. 2007. Yet Another MicroArchitectural Attack: : Exploiting I-Cache. In *Computer Security Architecture Workshop – CSAW*. ACM, 11–18.
[2] Onur Aciiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *Cryptography and Coding – IMA 2007 (LNCS)*, Vol. 4887. Springer, 185–203.
[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Asia Conference on Computer and Communications Security – AsiaCCS 2007*. ACM, 312–320.
[4] Onur Aciiçmez and Werner Schindler. 2008. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In *Topics in Cryptology – CT-RSA 2008 (LNCS)*, Vol. 4964. Springer, 256–273.
[5] Sarang Aravamuthan and Viswanatha Rao Thumparthy. 2007. A Parallelization of ECDSA Resistant to Simple Power Analysis Attacks. In *Communication System Software and Middleware – COMSWARE 2007*. IEEE, 1–7.
[6] Elaine Barker and Allen Roginsky (NIST). 2015. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. http://doi.org/10.6028/NIST.SP.800-131Ar1. (2015). NIST Special Publication 800-131A, Revision 1.
[7] Aurélie Bauer, Éliane Jaulmes, Victor Lomné, Emmanuel Prouff, and Thomas Roche. 2014. Side-Channel Attack against RSA Key Generation Algorithms. In *Cryptographic Hardware and Embedded Systems – CHES 2014 (LNCS)*, Vol. 8731. Springer, 223–241.

[11] https://github.com/openssl/openssl.git

[8] Daniel J. Bernstein. 2005. Cache-Timing Attacks on AES. Available online at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf. (April 2005).
[9] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. 2017. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In *Cryptographic Hardware and Embedded Systems – CHES 2017 (LNCS)*, Vol. 10529. Springer, 555–576.
[10] Dan Boneh. 1999. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the American Mathematical Society (AMS)* 46 (1999), 203–213.
[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Workshop on Offensive Technologies – WOOT 2017*. USENIX Association.
[12] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *System Software for Trusted Execution – SysTEX 2017*. ACM. In press.
[13] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium 2017*. USENIX Association, 1041–1056.
[14] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 7–18.
[15] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE Symposium on Security and Privacy – S&P 2009*. IEEE Computer Society, 45–60.
[16] Intel Corporation. 2017. Intel Software Guard Extensions Developer Guide. https://software.intel.com/en-us/sgx-sdk/documentation. (2017).
[17] Intel Corporation. 2017. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. (2017).
[18] Intel Corporation. 2017. Using the Intel Software Guard Extensions (Intel SGX) SSL Library. https://software.intel.com/en-us/sgx/resource-library. (2017).
[19] Thomas Finke, Max Gebhardt, and Werner Schindler. 2009. A New Side-Channel Attack on RSA Prime Generation. In *Cryptographic Hardware and Embedded Systems – CHES 2009 (LNCS)*, Vol. 5747. Springer, 141–155.
[20] OpenSSL Software Foundation. 2017. OpenSSL – Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. (2017).
[21] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. 2017. SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *Recent Advances in Intrusion Detection – RAID 2017 (LNCS)*, Vol. 10453. Springer, 357–380.
[22] Cesar Pereida García and Billy Bob Brumley. 2017. Constant-Time Callees with Variable-Time Callers. In *USENIX Security Symposium 2017*. USENIX Association, 83–98.
[23] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016), 1–27. https://doi.org/10.1007/s13389-016-0141-6
[24] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. 2009. Fast and Constant-Time Implementation of Modular Exponentiation. In *Embedded Systems and Communications Security – ECSC 2009*.
[25] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *European Workshop on System Security – EUROSEC 2017*. ACM, 2:1–2:6.
[26] Shay Gueron. 2012. Efficient Software Implementations of Modular Exponentiation. *J. Cryptographic Engineering* 2 (2012), 31–43.
[27] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX Annual Technical Conference – USENIX ATC 2017*. USENIX Association, 299–312.
[28] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Hardware and Architectural Support for Security and Privacy – HASP*. ACM, 11.
[29] American National Standards Institute. 1998. Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA). (1998).
[30] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO 1996 (LNCS)*, Vol. 1109. Springer, 104–113.
[31] Robert Könighofer. 2008. A Fast and Cache-Timing Resistant Implementation of the AES. In *Topics in Cryptology – CT-RSA 2008 (LNCS)*, Vol. 4964. Springer, 187–202.
[32] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium 2017*. USENIX Association, 557–574.

[33] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Hardware and Architectural Support for Security and Privacy – HASP*. ACM, 10.

[34] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. 1996. *Handbook of Applied Cryptography*. CRC Press.

[35] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2017 (LNCS)*, Vol. 10529. Springer, 69–90.

[36] Elaine Barker (NIST). 2016. Recommendation for Key Management, Part 1: General. http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4. (2016).

[37] Colin Percival. 2005. Cache Missing for Fun and Profit. http://daemonology.net/hyperthreading-considered-harmful/. (2005).

[38] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21 (1978), 120–126.

[39] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA 2017 (LNCS)*, Vol. 10327. Springer, 3–24.

[40] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Network and Distributed System Security Symposium – NDSS 2017*. The Internet Society.

[41] Ming-Wi Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium – NDSS 2017*. In press.

[42] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2015. Preventing Your Faults From Telling Your Secrets: Defenses Against Pigeonhole Attacks. *arXiv ePrint Archive, Report 1506.04832* (2015).

[43] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 317–328.

[44] J. Stein. 1967. Computational Problems Associated with Racah Algebra. *J. Comput. Phys.* 1 (1967), 397–405.

[45] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *USENIX Security Symposium 2016*. USENIX Association, 875–892.

[46] Raoul Strackx and Frank Piessens. 2017. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. *CoRR* abs/1712.08519 (2017).

[47] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology* 23 (2010), 37–71.

[48] Camille Vuillaume, Takashi Endo, and Paul Wooderson. 2012. RSA Key Generation: New Attacks. In *Constructive Side-Channel Analysis and Secure Design – COSADE 2012 (LNCS)*, Vol. 7275. Springer, 105–119.

[49] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Conference on Computer and Communications Security – CCS 2017*. ACM, 2421–2434.

[50] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Conference on Computer and Communications Security – CCS 2017*. ACM, 859–874.

[51] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy – S&P 2015*. IEEE Computer Society, 640–656.

[52] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium 2014*. USENIX Association, 719–732.

[53] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Cryptographic Hardware and Embedded Systems – CHES 2016 (LNCS)*, Vol. 9813. Springer, 346–367.