# Cache-Timing Attacks on RSA Key Generation

Alejandro Cabrera Aldaya
Universidad Tecnológica de la Habana (CUJAE)
Habana, Cuba
aldaya@gmail.com

Cesar Pereida García
Tampere University of Technology
Tampere, Finland
cesar.pereidagarcia@tut.fi

Luis Manuel Alvarez Tapia
Universidad Tecnológica de la Habana (CUJAE)
Habana, Cuba
lalvarezt89@gmail.com

Billy Bob Brumley
Tampere University of Technology
Tampere, Finland
billy.brumley@tut.fi

## ABSTRACT

During the last decade, constant-time cryptographic software has quickly transitioned from an academic construct to a concrete security requirement for real-world libraries. Most of OpenSSL's constant-time code paths are driven by cryptosystem implementations enabling a dedicated flag at runtime. This process is perilous, with several examples emerging in the past few years of the flag either not being set or software defects directly mishandling the flag. In this work, we propose a methodology to analyze security-critical software for side-channel insecure code path traversal. Applying our methodology to OpenSSL, we identify three new code paths during RSA key generation that potentially leak critical algorithm state. Exploiting one of these leaks, we design, implement, and mount a single trace cache-timing attack on the GCD computation step. We overcome several hurdles in the process, including but not limited to: (1) granularity issues due to word-size operands to the GCD function; (2) bulk processing of desynchronized trace data; (3) non-trivial error rate during information extraction; and (4) limited high-confidence information on the modulus factors. Formulating lattice problem instances after obtaining and processing this limited information, our attack achieves roughly a 28 % success rate for key recovery using the empirical data from roughly 10K trials.

## KEYWORDS

applied cryptography; public key cryptography; RSA; side-channel analysis; timing attacks; cache-timing attacks; OpenSSL; CVE-2018-0737

## 1 INTRODUCTION

Side-channel analysis (SCA) continues to be a serious threat against the security of systems and cryptography libraries. Specifically, microarchitecture attacks and cache-timing attacks are gaining more traction due to the severe architecture flaws recently discovered in many microprocessors [28, 33]. Cache-timing attacks are attractive for attackers and researchers due to the ability to perform them semi-remotely and without special privilege. So far, practical cache-timing attacks have been developed against multiple cryptosystems, including but not limited to DSA [39], ECDSA [38], DH [18] and RSA [51]. As a countermeasure against this type of attack, cryptography library developers such as OpenSSL and forks integrate in their codebase algorithms that execute in constant-time independently of the input values. During recent years, several researchers discovered and exploited flaws in these mitigations.

SCA research focuses mainly on cryptographic operations such as encryption, decryption, key exchange and signature generation. All of them have in common the repeated use of the private key as input during some step of the algorithm execution, thus being able to observe and capture the leakage over several runs. In contrast, SCA research targeting key generation seems to be neglected, presumably due to two dubious assumptions: (1) keys are only generated once during the initial stage in a secure environment, isolated from any possible threats; and (2) single trace attacks pose too many challenges, e.g. noise, and are not feasible. But technology trends shift and unfortunately this is no longer the case, a recent report on SSL certificate issuers [1] suggest Let's Encrypt is now the largest certificate issuer, with an adoption rate of more than 50% from web sites using SSL/TLS, thus highlighting the security needs against SCA on key generation algorithms. For that reason, our motivation comes from the Let's Encrypt [2] certificate authority, where RSA key generation is a common, regular, and semi-predictable operation for web server automated certificate renewal.

In this work, we present a methodology developed to identify the use of known side-channel vulnerable functions in cryptography libraries such as OpenSSL. Using our methodology, we disclose several vulnerabilities affecting OpenSSL RSA key generation implementation. Moreover, we present the first practical single trace cache-timing attack against the binary GCD step used during RSA key generation leading to complete RSA private key recovery. The root cause of the vulnerability is the GCD callee function not supporting the constant time flag, compounded by the parent function's failure to enable it. More precisely, our attack focuses on the execution of the non constant-time binary GCD algorithm to test the coprimality between the integers $p - 1$ and $q - 1$, and the public exponent $e$. Finally, this work serves as a reminder that cryptography libraries should strive for a secure by default approach, thus avoiding several side-channel attacks that still might be lurking in the codebase.

Our contributions in this work include:

- We develop a methodology to identify insecure code paths through known side-channel vulnerable functions still in use by cryptography libraries.
- We identify and exploit a flaw discovered with our methodology in OpenSSL that allows a practical single trace cache-timing attack against RSA key generation.

[1] https://nettrack.info/ssl_certificate_issuers.html
[2] https://letsencrypt.org/

- We combine several techniques from cache-timing attacks and power analysis to obtain traces during binary GCD execution and process them in order to obtain a sequence of shifts and subtraction operations, i.e. algorithm state, related with prime values $p$ and $q$.
- Building on existing RSA key recovery work, we propose a novel error correction algorithm for noisy RSA primes that allows us to recover roughly 50 % of bits for each prime.
- We implement a lattice attack that factors RSA-2048 keys knowing 522 bits of one prime.
- We implement an end-to-end attack and tested it for 10K independent keys achieving roughly a 28 % success rate, with room for improvement.

# 2 BACKGROUND

## 2.1 The RSA Cryptosystem

RSA is a public key cryptosystem invented in 1978 [43]. An RSA public key is a tuple of integers $(N, e)$ where $p$ and $q$ are primes and $N = pq$ holds, and furthermore $ed = 1 \bmod (p-1)(q-1)$ holds, implying both $e$ and $d$ are odd. For the remainder of this paper, we restrict to standardized RSA-$n$ that mandates for $n$-bit $N$ both $p$ and $q$ have bit-length $n/2$ and furthermore $2^{16} < e < 2^{256}$ holds [1]. For efficiency reasons, $e = 65537$ is the most common choice.

The private key is the tuple $\mathbf{sk} = (p, q, d, dp, dq, iq)$ where the latter three are CRT values not relevant to this work. For well-chosen parameters, recovering the private key from the public key is believed to be as hard as factoring $N$ [23]. Regarding security, the current minimum recommended RSA key size is 2048 bits, implying that $p$ and $q$ are 1024-bit primes. Applications of RSA in cryptography include public key encryption and digital signatures.

The secrecy of $p$ and $q$ is mandatory for RSA, moreover partial knowledge can lead to polynomial-time factoring algorithms. In his groundbreaking work, Coppersmith [15] proved that knowing half the bits of one prime suffices to factor $N$ in polynomial time, a critical point in our attack (see Section 5).

**Algorithm 1:** OpenSSL RSA key generation

**Input:** Key size $n$ and public exponent $e$.
**Output:** Public and private key pair.

```
1  begin
2      while gcd(p − 1, e) ≠ 1 do
3          p ← random n/2-bit prime          /* Generate p */
4      while gcd(q − 1, e) ≠ 1 do
5          q ← random n/2-bit prime          /* Generate q */
6      d ← e^−1 mod (p − 1)(q − 1)     /* Private exponent */
7      dp ← d mod (p − 1)
8      dq ← d mod (q − 1)                /* CRT parameters */
9      iq ← q^−1 mod p
10     return (N, e), (d, p, q, dp, dq, iq)
```

OpenSSL's RSA key generation closely resembles Algorithm 1. The first steps aim at generating random secret primes $p$ and $q$, during which two loops ensure that $(p-1)$ and $(q-1)$ are coprime with $e$. Steps 7-9 are not relevant to this work.

Algorithm 1 involves the computing of (at least) two GCDs and two modular inversions. Binary GCD algorithms are a common implementation choice for both of these operations—a description follows.

## 2.2 Binary GCD Algorithms

Stein [45] proposed the binary greatest common divisor algorithm (binary GCD) in 1967. This algorithm computes the GCD of two integers $a$ and $b$ employing only right-shift operations and subtractions (Algorithm 2). This approach is very attractive in cryptography as it performs very well specially with large inputs.

In OpenSSL, GCD computations use the function `BN_gcd`, a high level wrapper to the function `euclid` that is one implementation of the binary GCD. Note however it does not follow the *classic* algorithm structure (c.f. Algorithm 2). Regardless, its flow can be analyzed using the classic variant since all side-channel models for this algorithm have the same threat model. An adversary can distinguish a right-shift from a subtraction operation to recover algorithm state [2, 3, 38]. We expand on this concept later in this section.

**Algorithm 2:** Binary GCD

**Input:** Integers $a$ and $b$ such that $0 < a < b$.
**Output:** Greatest common divisor of $a$ and $b$.

```
1  begin
2      u ← a, v ← b, i ← 0
3      while even(u) and even(v) do      /* Powers of two */
4          u ← u/2, v ← v/2, i ← i + 1
5      while u ≠ 0 do                    /* Greatest odd divisor */
6          while even(u) do
7              u ← u/2                          /* u loop */
8          while even(v) do
9              v ← v/2                          /* v loop */
10         if u ≥ v then
11             u ← u − v                    /* sub-step */
12         else
13             v ← v − u
14     return v · 2^i
```

Finally, it is worth noting that with respect to RSA key generation, Step 4 never executes since (at least) one of the inputs is always odd.

## 2.3 Binary GCD: Side-Channel Analysis

The execution flow of Algorithm 2 is highly dependent of its inputs. In Algorithm 2 some execution flow relevant steps are highlighted. The *u-loop* and *v-loop* are the loops that remove all power-of-two divisors in variables $u$ and $v$ at each iteration. The *sub-step* executes when both variables are odd, consisting of a single subtraction.

Consistent with the existing literature [3, 38], we encode the execution flow sequence of this algorithm with two symbols 'L'

and 'S' representing right-shift and subtraction, respectively. Another representation uses two variables $Z_i$ and $X_i$ defined in [3] as follows: [3]

- $Z_i$ stores the number of right-shifts at iteration $i$.
- $X_i$ stores a binary value to represent the result of the condition (Step 10 in Algorithm 2) at iteration $i$. $X_i$='$u$' means the condition was true while $X_i$='$v$' the opposite.

Figure 1 shows an LS-sequence example of an execution flow. The sequence reads from left to right: the first four $Z_i$ are 3, 2, 1, and 5.

LLLSLLSLSLLLLLSL...LS

**Figure 1: LS-encoded binary GCD execution flow example.**

Regarding side-channel analysis, there are three different models for analyzing Algorithm 2 leakages. Each model originally targets the Binary Extended Euclidean Algorithm (BEEA) for computing modular inverses. However they also apply to Algorithm 2 because the models exploit the execution flow leakage w.r.t. variables $u$ and $v$, and said flow is the same for both algorithms when executed with the same input pair.

*All-or-nothing.* Acıiçmez et al. [2] and Aravamuthan and Thumparthy [6] independently proposed this model in 2007. It requires that the adversary knows *all* $Z_i$ and $X_i$ to recover algorithm inputs.

*Partial.* Aldaya et al. [3] recently proposed this model, an algebraic approach that relates the number of known $Z_i$, $X_i$ with the number of input bits that can be recovered. In comparison with the previous model, this approach is more flexible as it can extract information from partial knowledge of the execution flow. In this model, the number of recovered bits grows with the number of $Z_i$ and $X_i$ that an adversary knows. Our work employs this model (see Section 3.2 for this selection rationale).

*Look-up.* Pereida García and Brumley [38] proposed a model that also allows partial recovery, but instead of an algebraic approach it employs a table look-up. The adversary generates a table that relates every LS-sequence of a given length with the corresponding partial input bits. This model performs better than the previous when the number of bits to recover is small as it captures some algebraic equivalences not previously modeled. However, it becomes impractical for recovering a large number of bits (i.e. longer LS-sequences). The magnitude increases linearly whereas the computational complexity (time and storage) for creating a table containing all possible LS-sequences increases exponentially.

## 2.4 The Flush+Reload Attack

The Flush+Reload technique is a cache-based side-channel attack technique targeting the Last-Level Cache (LLC) and used during our attack. Flush+Reload is a high resolution, high accuracy and high signal-to-noise ratio technique that positively identifies accesses to specific memory lines. It relies on cache sharing between processes, typically achieved through the use of shared libraries or page deduplication.

[3]Equivalent to SHIFTS[i] and SUBS[i] definition by Acıiçmez et al. [2]

**Algorithm 3:** Flush+Reload Attack

**Input:** Memory Address *addr*.
**Result:** True if the victim accessed the address.

```
1 begin
2     flush(addr)
3     Wait for the victim.
4     time ← current_time()
5     tmp ← read(addr)
6     readTime ← current_time() - time
7     return readTime < threshold
```

A round of attack, depicted in Algorithm 3, consists of three phases: (1) The attacker evicts the target memory line from the cache. (2) The attacker waits some time so the victim has an opportunity to access the memory line. (3) The attacker measures the time it takes to reload the memory line. The latency measured in the last step tells whether or not the memory line was accessed by the victim during the second step of the attack, i.e. identifies cache-hits and cache-misses.

The Flush+Reload attack technique tries to achieve the best resolution possible while keeping the error rate low. Typically, an attacker encounters multiple challenges due to several processor optimizations and different architectures. See [5, 39, 50] for discussions of these challenges.

## 2.5 Related Work

*Attacks on RSA keys.* Over the years, cryptanalysis of RSA keys have been studied and targeted due to its widespread usage, its mathematical structure (i.e. CRT-based methods) and the ease of generating low entropy keys. One classification of attacks on RSA keys is dividing them between: (1) only public key knowledge; (2) partial private key knowledge.

The first category assumes an attacker only has knowledge of the public key $(N, e)$, attempting to use factoring methods such as Pollard $p-1$ [40], Pollard Rho [41] and sieving methods to recover the private factors $p$ and $q$. This type of attack is bound by the often sub-exponential, yet intractable, time complexity of the factoring methods, requiring massive computation time and resources. Current research achieves factorization of 768-bit RSA keys [27], therefore it has limited practical applicability and interest for an attacker.

The second category exploits partial knowledge about the private *and* public keys to perform attacks such as low exponent attacks [10, 48], side-channel attacks [7, 51] and Coppersmith related attacks [15, 16], considered a universal tool to attack RSA keys with poorly chosen parameters or keys generated with poor entropy, i.e. using a faulty implementation.

In 2012, two independent teams [21, 31] exploited poor entropy of RSA keys in SSL certificates, SSH host keys and PGP keys, thus allowing them to trivially factorize keys by carrying out pairwise GCD computations to recover shared prime factors among other RSA keys. Similarly in 2013, Bernstein et al. [9] analyzed the public record of RSA keys in the "Citizen Digital Certificate" database of Taiwanese citizens. The authors recovered 265 private keys by

running a batch GCD computation followed by Coppersmith's method.

In 2017, Nemec et al. [34] discovered a critical vulnerability in the library used to generate RSA keys for identity cards, passports and Trusted Platform Modules; allowing factorization of 1024 and 2048-bit keys. Once again, this exploit was possible due to poor entropy introduced by a special mathematical structure of the prime factors that not only allowed key recovery using Coppersmith's method but also detection of keys with this special structure. The impact was so vast that multiple countries were forced to recall and issue new identity cards for their citizens.

*Microarchitecture Attacks on RSA.* In his seminal work, Percival [37] demonstrated a cache-timing attack against RSA by identifying access to precomputed multipliers stored in memory when using the Sliding Window Exponentiation (SWE) algorithm implemented in OpenSSL version 0.9.7c. To mitigate this issue, the OpenSSL team added a "constant-time" implementation of the modular exponentiation algorithm. This implementation combines a fixed-window exponentiation algorithm with a scatter-gather method [11], allowing to mask table access to the multipliers. The scatter-gather method ensures the same cache lines are always accessed, irrespective of the multiplier used.

In 2016, Yarom et al. [51] showed the scatter-gather method implemented in OpenSSL derived from Percival's work, still leaks timing information and thus, is not "constant-time" as it was believed. In their work, the authors demonstrated that even if the same cache lines are always accessed, the offset accessed within the cache line depends on the multiplier used, which is decided based on the private key. To that end, the authors exploited cache-bank conflicts. A cache is often divided in cache banks that allow concurrent access to the cache, but only one request at a time. If multiple requests are made to the same cache bank, a conflict occurs and the conflicting requests are delayed, creating a timing variation. The timing variations allowed to perform an attack against OpenSSL 1.0.2f, leading to 4096-bit RSA key recovery after observing 16000 decryptions on a HyperThreading architecture.

More recently, Bernstein et al. [8] performed 1024 and 2048-bit key recovery in the Libgcrypt library when computing modular exponentiations using the left-to-right sliding window method. More precisely, the authors demonstrated the direction of the sliding window matters since it leaks more or less information depending on the encoding direction. Applying the Flush+Reload technique, paired with the algorithm by Heninger and Shacham [22], the authors are able to efficiently reconstruct the keys using a side-channel leak after recovering roughly 50 % of the secret bits.

Acıiçmez et al. [2] showed that exponentiation is not the only operation potentially leaking RSA secret information. The modular inversion operation is also vulnerable and they developed Simple Branch Prediction Analysis (SBPA) to this end. The authors showed information leakage in OpenSSL 0.9.8a during the modular inversion operation due to the use of the Binary Extended Euclidean Algorithm (BEEA). The BEEA is used during key generation, decryption, and blinding when employing the RSA-CRT variant, and these algorithms compute on secret values. The authors conjecture it is theoretically possible to deduce the outcome of branch statements using timings, recovering critical BEEA algorithm state and therefore leading to secret key recovery.

*Side-channel attacks on RSA key generation.* The research available on SCA against RSA key generation is limited and focuses on leakages in physical devices. Finke et al. [17] performed an attack on a custom implementation of a prime generation algorithm used for RSA key generation, analyzed using Simple Power Analysis (SPA). In 2012, Vuillaume et al. [46] presented a Differential Power Analysis (DPA), template attack and fault attack on the Fermat and Miller-Rabin tests on a secure microcontroller but give no additional information regarding their setup. Later on, [7] analyzed the security of prime generation algorithms and the sieving process. Targeting the divisibility phase, the authors obtained more than half the bits of the prime number generated in their own implementation and then using Coppersmith's technique they recovered a 1024-bit RSA key. More recently, Aldaya et al. [4] analyze the modular inversion operation used during RSA private key generation. The BEEA is commonly used to perform modular inversions, but due to its highly input-dependent flow, the authors demonstrate full key recovery using SPA. The attack differs from previous work because it focuses on alternative routines invoked during key generation, instead of primality tests or prime number generation.

Recently, independent work examines one of the three code paths considered in our work (`BN_gcd`). Weiser et al. [47] target RSA key generation within an Intel SGX enclave by a noiseless controlled-channel page-fault attack. Controlled-channel attacks [49] are privileged attacks originating from a malicious OS targeting SGX enclaves, aligned with the SGX threat model. The most important differences with our work are (1) cache-timing attacks are unprivileged and do not require escalation to kernel space (i.e. a malicious OS); and (2) controlled-channels are error-free, while cache-timing channels are far from that.

# 3 RSA KEY GENERATION: NEW VULNERABILITIES

## 3.1 Insecure Code Paths: A Methodology

After its introduction in 2005, two works exploit the insecure default behavior of OpenSSL's constant time flag. Pereida García et al. [39] exploit the fact that, by design, the flag does not propagate from the source to the destination during BIGNUM copy operations. As a result, modular exponentiations during DSA sign operations took a side-channel insecure modular exponentiation path. Pereida García and Brumley [38] exploit the failure to set the flag during ECDSA sign operations. In that case, the resulting scalar multiplication function is oblivious to the flag and always followed a side-channel secure path; the modular inversion function, however, requires this flag to follow its side-channel secure path.

While not explicitly stated, it seems both of these vulnerabilities were found by manual code review—locating critical locations where the flag should be set for individual cryptosystems, and tediously tracking the flow. This is not a particularly efficient and accurate way to assess and ensure the flag's proper usage.

In contrast, we approach this problem from the opposite perspective: collecting a set of known side-channel vulnerable functions

within OpenSSL, and using stock tooling to determine if these functions reach vulnerable code during security-critical operations. Roughly, our methodology consists of the following steps.

(1) From the existing literature, create a list of side-channel vulnerable functions within a library. (Here, OpenSSL.)
(2) In a debugger, set break points at lines of code which should not be reached during security-critical operations such as key generation, public key decryption, and digital signature generation.
(3) Run the security-critical command and analyze the call stack upon hitting said break points.

Applying this methodology, w.r.t. RSA key generation we identified the following subset of known side-channel vulnerable functions of interest.

(1) The function `BN_gcd` contains highly input-dependent branches that can potentially be used as a side-channel attack vector. Since the code has no early exit to a side-channel secure code path, i.e. does not check the constant time flag at all, we set a break point at the function's entry point.
(2) The function `BN_mod_inverse` executes a check for the constant time flag at the beginning of the function, and early exits to a side-channel secure path if it is set. If the code fails the check, it continues to a side-channel insecure path. We set a break point immediately following the early exit.
(3) The function `BN_mod_exp_mont` is analogous to the above, yet for modular exponentiation. Similarly, we set a break point immediately following the early exit.

Following a debug session in Figure 2, these three break points are seen in `bn_gcd.c` (Lines 120 and 238) and `bn_exp.c` (Line 418). Upon executing the `genpkey` command to generate an RSA key, each of the three break points are hit multiple times, and the output gives their corresponding call stacks. Naturally, hitting the break points does not guarantee a vulnerability—a deeper analysis follows.

*Insecure exponentiation code path.* The Miller-Rabin primality test [42] is the most common implementation of Algorithm 1, Lines 3 and 5. It involves choosing a random "witness" base $b$ then computing $b^x \mod p$ where $p$ is the candidate prime and the relation $2^k x = p - 1$ holds. Indeed, OpenSSL's is a straightforward implementation of these steps. Looking at the call stack for the `BN_mod_exp_mont` break point, the function `BN_is_prime_fasttest_ex` implements iterating this test for different $b$ values to obtain prime confidence after sufficient successful trials. It carries out each trial by calling the function `witness` that performs the modular exponentiation, unfortunately calling `BN_mod_exp_mont` without setting `BN_FLG_CONSTTIME`. The algorithm continues with a classical sliding window exponentiation, potentially leaking partial information on $x$ hence $p$.

*Insecure inversion code path.* Related to the previous code path, as the function name `BN_mod_exp_mont` suggests, the implementation uses Montgomery arithmetic for efficiency. The Montgomery setup phase occurs in `BN_MONT_CTX_set`, computing the inverse of $2^w$ modulo $p$ for $w$-bit architectures. Examining the call stack, the function calls `BN_mod_inverse` without setting `BN_FLG_CONSTTIME`, potentially leaking critical binary GCD algorithm state. However, in this case our terse analysis reveals the operands are not $\{2^w, p\}$

```
+--------rsa_gen.c-------------------------------------------------------------------------+
|103                                                                                       |
|104     static int rsa_builtin_keygen(RSA *rsa, int bits, BIGNUM *e_value,               |
|105                                   BN_GENCB *cb)                                       |
|106     {                                                                                 |
|...                                                                                       |
|148         /* generate p and q */                                                       |
|149         for (;;) {                                                                    |
|150             if (!BN_generate_prime_ex(rsa->p, bitsp, 0, NULL, NULL, cb))             |
|151                 goto err;                                                             |
|152             if (!BN_sub(r2, rsa->p, BN_value_one()))                                  |
|153                 goto err;                                                             |
|154             if (!BN_gcd(r1, r2, rsa->e, ctx))                                         |
|155                 goto err;                                                             |
|156             if (BN_is_one(r1))                                                        |
|157                 break;                                                                |
+------------------------------------------------------------------------------------------+

+--------bn_gcd.c---------------------------------------------------------------------------+
|116                                                                                       |
|117     int BN_gcd(BIGNUM *r, const BIGNUM *in_a, const BIGNUM *in_b, BN_CTX *ctx)       |
|118     {                                                                                 |
|119         BIGNUM *a, *b, *t;                                                            |
B+>|120         int ret = 0;                                                              |
|...                                                                                       |
|143         t = euclid(a, b);                                                             |
|...                                                                                       |
|156     static BIGNUM *euclid(BIGNUM *a, BIGNUM *b)                                       |
|157     {                                                                                 |
|158         BIGNUM *t;                                                                    |
|159         int shifts = 0;                                                               |
|160                                                                                       |
|161         bn_check_top(a);                                                              |
|162         bn_check_top(b);                                                              |
|163                                                                                       |
|164         /* 0 <= b <= a */                                                            |
|165         while (!BN_is_zero(b)) {                                                      |
|...                                                                                       |
|226     BIGNUM *BN_mod_inverse(BIGNUM *in,                                                |
|227                            const BIGNUM *a, const BIGNUM *n, BN_CTX *ctx)            |
|228     {                                                                                 |
|229         BIGNUM *A, *B, *X, *Y, *M, *D, *T, *R = NULL;                                 |
|230         BIGNUM *ret = NULL;                                                           |
|231         int sign;                                                                     |
|232                                                                                       |
|233         if ((BN_get_flags(a, BN_FLG_CONSTTIME) != 0)                                  |
|234             || (BN_get_flags(n, BN_FLG_CONSTTIME) != 0)) {                            |
|235             return BN_mod_inverse_no_branch(in, a, n, ctx);                           |
|236         }                                                                             |
|237                                                                                       |
B+>|238         bn_check_top(a);                                                          |
+------------------------------------------------------------------------------------------+

+--------bn_exp.c---------------------------------------------------------------------------+
|402                                                                                       |
|403     int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, const BIGNUM *p,                 |
|404                         const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *in_mont)           |
|405     {                                                                                 |
|406         int i, j, bits, ret = 0, wstart, wend, window, wvalue;                        |
|407         int start = 1;                                                                |
|408         BIGNUM *d, *r;                                                                |
|409         const BIGNUM *aa;                                                             |
|410         /* Table of variables obtained from 'ctx' */                                 |
|411         BIGNUM *val[TABLE_SIZE];                                                      |
|412         BN_MONT_CTX *mont = NULL;                                                     |
|413                                                                                       |
|414         if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0) {                                 |
|415             return BN_mod_exp_mont_consttime(rr, a, p, m, ctx, in_mont);              |
|416         }                                                                             |
|417                                                                                       |
B+>|418         bn_check_top(a);                                                          |
+------------------------------------------------------------------------------------------+

Breakpoint 1 at 0x5fd4aa: file bn_gcd.c, line 120.
Breakpoint 2 at 0x5fda66: file bn_gcd.c, line 238.
Breakpoint 3 at 0x4e97a9: file bn_exp.c, line 418.
Starting program: openssl genpkey -algorithm RSA -out key.pem -pkeyopt rsa_keygen_bits:2048
Breakpoint 2, BN_mod_inverse (...) at bn_gcd.c:238
#0  BN_mod_inverse (...) at bn_gcd.c:238
#1  ... in BN_MONT_CTX_set (...) at bn_mont.c:450
#2  ... in BN_is_prime_fasttest_ex (...) at bn_prime.c:319
#3  ... in BN_generate_prime_ex (...) at bn_prime.c:199
#4  ... in rsa_builtin_keygen (...) at rsa_gen.c:150
Continuing.
Breakpoint 3, BN_mod_exp_mont (...) at bn_exp.c:418
#0  BN_mod_exp_mont (...) at bn_exp.c:418
#1  ... in witness (...) at bn_prime.c:356
#2  ... in BN_is_prime_fasttest_ex (...) at bn_prime.c:329
#3  ... in BN_generate_prime_ex (...) at bn_prime.c:199
#4  ... in rsa_builtin_keygen (...) at rsa_gen.c:150
Continuing.
Breakpoint 1, BN_gcd (...) at bn_gcd.c:120
#0  BN_gcd (...) at bn_gcd.c:120
#1  ... in rsa_builtin_keygen (...) at rsa_gen.c:154
```

**Figure 2: Testing the proposed methodology.**

but $\{2^w, p \bmod 2^w\}$, implemented by copying the least significant word of $p$ to a temporary BIGNUM. While all leaks are bad, some are worse than others—this is a nominal leak on the least significant word of $p$.

*Insecure GCD code path.* The shallowest call stack is for `BN_gcd`, called directly by `rsa_builtin_keygen` (Line 154). The function computes the GCD of $e$ and $p-1$ to ensure that $e$ is invertible mod $(p-1)(q-1)$. The value $p-1$ should remain secret, hence hitting this break point represents a potential side-channel attack vector. This is the code path we target in the remainder of this paper.

*Root cause analysis.* From these results, we deduce modular inversions in OpenSSL's RSA key generation at Steps 6 and 9 of Algorithm 1 have side-channel mitigations in place, yet GCD computations in Steps 2 and 4 lack such protection, and likewise for primality testing. The work of Acıiçmez et al. [2] induced the secure path code change, yet the impact of the academic result did not fully propagate throughout the entirety of the RSA key generation implementation. We speculate this is a result of a simplification in [2, Sec. 2.1]: the pseudocode for key generation abstracts away the prime generation loop, and assumes a priori coprimality of $e$ with $p-1$ and $q-1$ to compute $d$ at Step 6. This allowed Acıiçmez et al. to focus theoretical analysis on the impact of modular inversion leaks across various cryptosystems, while undoubtedly being aware of GCD and modular inversion execution flows having essentially the same branching characteristics. Alas, typical engineers are less inclined to such cryptographic subtleties—evidenced by this code path remaining vulnerable to microarchitecture side-channel attacks.

## 3.2 Theoretical Leakage Analysis

To this end, Pereida García and Brumley [38] demonstrate it is possible to recover some $Z_i$ from OpenSSL modular inversion operations (BEEA) with cache timings during ECDSA signature generations. We are left with the following open question: *Is it possible to similarly recover binary GCD algorithm state?*

Aldaya et al. [4] analyzed RSA key generation with respect to SCA of GCD-based algorithms. The analysis focuses on the modular inversion at Step 6 of Algorithm 1, exploiting the fact that BEEA inputs have very different bit-lengths. The product $(p-1)(q-1)$ has 2048 bits for modern RSA key sizes, while the other input $e$ has only 17 bits commonly.

Our vulnerability similarly exploits a large bit-length difference between inputs: the same $e$, but instead $p-1$ and $q-1$ having 1024 bits. As processing $p$ and $q$ are very similar regarding GCD computation, we use the prime $p$ to present our analysis. Furthermore, we select the *partial* bit-recovery model (see Section 2.3) because (1) we will be working with noisy LS-sequences later in our full attack, thus partial recovery reduces noise influence; (2) covered later in this section, we will utilize a factoring method that inputs incomplete $p$; and (3) we need to recover hundreds of bits so *Look-up* model is intractable.

The large bit-length difference between $p-1$ and $e$ implies that during several binary GCD iterations the condition $u \geq v$ will be true, giving the adversary partial execution flow information a priori (i.e. $X_i$='$u$' for some iterations $i$). This situation holds until $u$ (initialized to $p-1$) stores a value of roughly the same bit-length as $v$ (initialized to $e$). Therefore it divides $u$ by two roughly $\lg(N)/2 - \lg e$ times.

According to the $Z_i$ definition, at each iteration $u$ loses $Z_i$ bits. Therefore the number of iterations $t$ that should execute before $u$ has roughly the same bit-length as $v$ is the minimum $t$ that satisfies (1).

$$n = \sum_{i=1}^{t} Z_i \geq \lg(N)/2 - \lg e \tag{1}$$

Hence, the following question arises: *how many bits can be recovered in this setting?*

*Partial recovery.* Applying the *partial* model, we obtain a bit-recovery equation as follows. Assume the adversary obtains all $Z_i$, and $t$ is the first iteration for which $u < v$ (i.e. $X_i$='$u$' ; $0 < i < t$). The values of $u$ and $v$ just before the *sub-step* for iterations $i < t$ are as follows.

$$u_1 = p - 1, \qquad v_1 = v_i = e, \qquad u_{i+1} = \frac{u_i - v_i}{2^{Z_{i+1}}}$$

The invariant $u_i - v_i \equiv 0 \bmod 2$ holds for all iterations, since both variables are odd just before the *sub-step*. Expanding for $i < t$:

$$u_t - v_t = \cfrac{\cfrac{\cfrac{\cfrac{p-1}{2^{Z_1}} - e}{2^{Z_2}} - e}{2^{Z_3}} - e}{\ddots} - e \equiv 0 \bmod 2$$

thus solving for $p$ yields (2) for bit recovery, where $n$ from (1) is the number of recovered bits of $p$.

$$p \equiv e(2^{Z_1} + 2^{Z_1+Z_2} + \cdots + 2^n) + 1 \bmod 2^{n+1} \tag{2}$$

In summary, for RSA-2048 $n$ is roughly $1024 - 17 = 1007$ bits. However, due to Coppersmith [15] an adversary only needs $\lg(N)/4 = 512$ bits of one prime to factor an RSA-2048 $N$, depicted in Figure 3. For either NIST compliant value of $e$, the number of bits recovered is far beyond the Coppersmith bound.

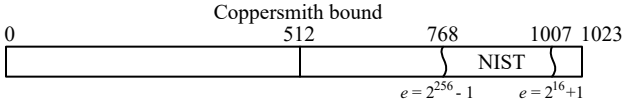


**Figure 3: RSA-2048 bit-recovery bounds of $n$ for different $e$.**

We now have our requirements for a successful attack—the adversary must obtain (at least) the first $t_c$ noise-free $Z_i$ to factor $N$:

$$n = \sum_{i=1}^{t_c} Z_i \geq \lg(N)/4$$

where $t_c$ is the minimum iteration for which $n$ reaches the Coppersmith bound.

### 3.3 A Single Trace Attack: Roadmap

Previously in this section, we uncovered three side-channel insecure code paths traversed during RSA key generation. Subsequently focusing on the `BN_gcd` code path, we then gave a theoretical analysis on the GCD algorithm as implemented in OpenSSL to describe what kind of side-channel information we can extract, and rough bounds for how much (noise-free) information we need to leverage that to recover the private key by factoring $N$. The remainder of this paper is dedicated to describing the methods, techniques and problems faced when trying to recover the necessary information from the side-channel leakage in order to achieve full private RSA key recovery from a single trace. The roadmap for our end-to-end attack is as follows.

(1) We capture cache-timing traces during `BN_gcd` execution during RSA key generation, then—leveraging signal processing techniques—extract the portions corresponding to $p-1$ and $q-1$, apply digital filters and extract their corresponding (noisy) LS-sequences (Section 3.4);
(2) Building upon previous work, we design and implement an error correction algorithm for these sequences—leveraging number theoretic constraints imposed by RSA—to extract partial bits of one factor of $N$ (Section 4);
(3) Said algorithm yields an ordered list of candidates for partial factors; we then derive lattice parameters for factoring with Coppersmith's method, and create lattice instances with said candidates, iteratively executing them until the result yields complete factorization of $N$ (Section 5).

### 3.4 From Timings to Sequence of Operations

The environment for our attack consists of an Intel Core i5-2400 Sandy Bridge 3.10 GHz (32 nm) with 8 GB of memory running 64-bit Ubuntu 16.04 LTS "Xenial" with hardware prefetching and Turbo Boost disabled. All the cores share a 12-way 6 MB unified LLC. The system does not feature HyperThreading.

We tested our attack against OpenSSL 1.0.2k with debugging symbols on the executable. We use the debugging symbols to map source code to memory addresses, this allow us to find the "hot" memory addresses for the degrading attack and probing accurately the sequence of operations mentioned previously. We passed the `shared` configuration option to compile OpenSSL as a shared object.

As discussed previously, the GCD algorithm execution as implemented in OpenSSL is highly dependent on its inputs. To that end, we use the well-known Flush+Reload technique to probe cache lines in code routines `BN_rshift1` and `BN_sub`. By probing these two routines, we are able to distinguish two branches executed by the GCD algorithm, namely right shifts and subtractions. Unfortunately this is not enough to recover meaningful data, since we need to know the exact $Z_i$ values (i.e. number of right shifts executed between subtractions) in order to identify bits, but due to the speed of the operations our probe misses some of the accesses. To get better resolution, we pair the Flush+Reload technique with the performance degradation attack [5] which targets different cache lines in the same previous routines to slow down the execution. Moreover, we apply the profiling approach [38] to easily identify the best memory addresses to probe and degrade. Adapted to our strategy, this provides a good starting point (see a trace excerpt in Figure 4) to recover a sequence of operations.

The contents of a typical trace (see Figure 5, top), from high to low abundance, is roughly: (1) noise and/or `BN_mod_exp_mont` executions during primality testing, not targeted by our probes but nonetheless consuming CPU cycles; (2) short `BN_mod_inverse` executions setting up Montgomery arithmetic, with the same underlying shifts and subtracts as `BN_gcd` that our probes target; (3) longer `BN_gcd` executions for testing coprimality. We are only interested in the latter, yet manually isolating the part of the trace that corresponds to the real `BN_gcd` executions for $p-1$ and $q-1$ is time consuming and not feasible at a large scale.

For other cache-timing attacks that instead target digital signature generation or public key decryption, this is mostly a non-issue solved by simple heuristics. The reason is that in those scenarios, attackers have oracle access to trigger the cryptographic operation. They then start side-channel signal acquisition (e.g. launch the spy) roughly at the same time as they start their query (e.g. initiate a protocol run or call an API), and can use rough estimates, e.g. based on profiled timing to extract and trim the trace, isolating the portion of interest.

In our case, this problem is very different since we have no control over when key generation takes place. Our traces are extremely long, leaving us with the challenging task of extracting a minuscule partial trace from abundant data—looking for a needle in a haystack. We turn to signal processing-based power analysis techniques to tackle this issue. Inspired by SPA and template attacks [13] and related (yet less statistical) cache-based techniques [12, 19], we create and use templates—exemplary partial traces—to trim full traces. Departing from these works and motivated by statistical methods such as Horizontal Correlation Analysis [14], we leverage the Pearson correlation coefficient to find the best match for these templates within full traces, automating the process.

*Templates.* We first create a template by manually inspecting and trimming a trace that looks visually correct, i.e. no clear preemptions of the probes or the victim process. It is worth mentioning the templates depend on the waiting period and additional parameters defined for the Flush+Reload technique, thus every time the parameters change during the probing stage, a new template needs to be created, otherwise the next step will fail to find the GCD iterations we are interested in. This fine tuning task can be tedious but it is a crucial step in a cache-timing attack and it is the difference between a failed and a successful attack. The middle plot in Figure 5 depicts one template we used in our experiments.

*Correlation.* Denote the length-$m$ trace by $t$ and length-$n$ template by $s$. The measurement at time $i$ is $t[i]$ and $t[i : i+n]$ the length-$n$ partial trace starting from point $t[i]$. The Pearson correlation coefficient of $t$ with $s$ at time $j$ is

$$r[j] = \frac{\sum_{i=1}^{n} s[i]t[i+j] - n\mu_s\mu_{t[j:j+n]}}{n\sigma_s\sigma_{t[j:j+n]}} \tag{3}$$

where $\mu_x$ are means and $\sigma_x$ standard deviations. The output lies in the interval $[-1\,..\,1]$ from perfect negative correlation, to no linear correlation, to perfect correlation.

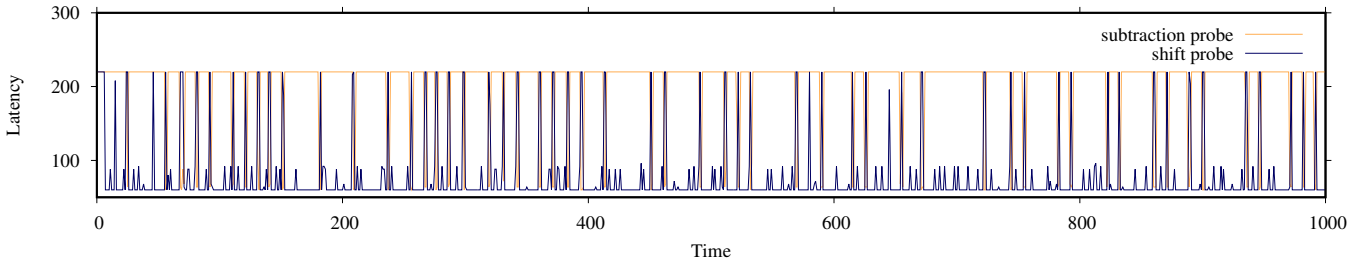


**Figure 4: A raw cache-timing trace obtained by only applying Flush+Reload and performance degradation techniques against the binary GCD.**

With $m$ large, care must be taken in computing the *moving* correlation $r[1 : m - n + 1]$. In (3), the summations are cross correlations, and the moving average $\mu_{t[j:j+n]}$ for all $j$ is a convolution. With a random variable $X$ and denoting $E$ expectation, the equality $\sigma = \sqrt{E[X^2] - E([X])^2}$ holds, so the moving standard deviation $\sigma_{t[j:j+n]}$ for all $j$ computes directly from the moving averages. To summarize, all of these are FFT-based methods to compute (3) efficiently for all $j$—even for our long traces.

*Matching.* Once we create our template and acquire our trace, we compute the moving Pearson correlation between the template and full trace, then extract the peaks, i.e. discrete indices that exhibit highest correlation between the template and the side-channel data. This automates the GCD identification step, proving to be accurate at finding the GCD executions of interest in our traces, allowing us to extract the sequences for $p$ and $q$.

Figure 5 illustrates our method in action using a real trace and template. The top plot is the filtered partial trace, containing two GCD runs for $p$ and $q$—note the narrower peaks corresponding to executions of `BN_mod_inverse` during the primality test, also leaking secret information on the prime values due to yet another flag not set (see Section 3.1).

The bottom plot is the moving Pearson correlation coefficient between the template (middle plot) and the trace, with two extremely distinguishable peaks that identify the locations of the two GCD operations. The middle plot aligns the template at maximum correlation for visualization purposes. As seen, our technique is remarkably effective.

*Horizontal analysis.* We need to extract the sequence of shift and subtraction operations from the $p$ and $q$ traces (i.e. $Z_i$). Due to the nature of the GCD algorithm, as the input values to the function are processed, their bit length decreases and this behavior is reflected in the trace. Typically, when a GCD operation starts execution, a single shift operation spans over several cache-hits in the trace, while at the end of the execution the same shift operation registers fewer points, sometimes even only one point. For this reason, we use a dynamic horizontal analysis approach to recover the sequence of operations executed during a GCD execution.

Our dynamic horizontal analysis works as follows: first we take as input the processed trace which has been aligned to the first subtraction operation and trimmed to a specific length, avoiding noise as much as possible. Then, take small chunks of the trace containing $n$ windows of subtraction plus shift operations, recall that each subtraction is followed by at least one shift operation. After that, we compute the Euclidean distance between the subtraction operations in those $n$ windows. We sort the resulting distances from shortest to longest and then we consider the shortest distance as a single shift operation (i.e. $Z_i = 1$), thus using it as basis to determine the number of shift operations in the rest of windows. After calculating all the shift counts in those $n$ windows, we proceed to the next chunk and repeat the process. We continue until all the operations in the trace have been calculated, resulting in a tentative *LS sequence* of operations (i.e. $Z_i$ candidates).

As mentioned previously, the accuracy of the operations in the trace decreases dramatically by the end of the GCD execution, therefore the trace is not perfect, in fact it contains errors introduced by several factors such as victim preemptions, spy preemptions as well as noise created by other processes and microarchitecture components. To defeat this, Section 4 details an error correction algorithm developed to find potential correct LS sequence candidates that later are converted to bits and used as input values to perform the lattice attack explained in Section 5.

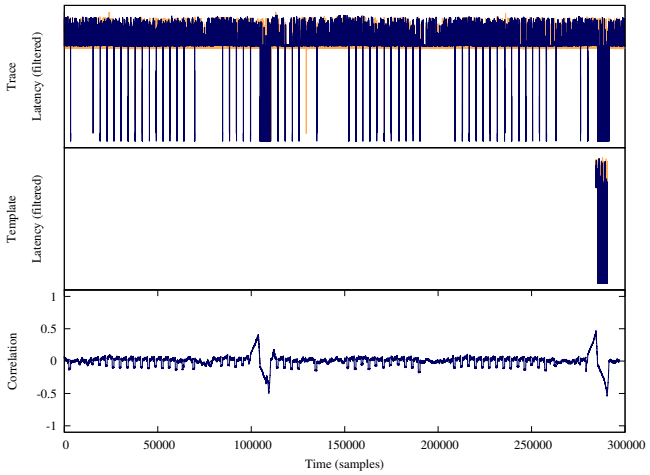


**Figure 5: Visualization of the moving Pearson correlation in action. From top to bottom: raw trace, aligned template trace, and Pearson correlation.**

## 4 ERROR CORRECTION IN RSA KEYS

Following Section 2.1, an RSA private key is composed by a six element tuple **sk** = ($p$, $q$, $d$, $dp$, $dq$, $iq$). The knowledge of any of these elements directly allows breaking the system. In addition, these elements contain redundancy and are related through public information. For example, as $N = pq$, the relation $N \equiv pq \bmod 2^n$ holds for every positive integer $n$. Therefore the knowledge of the first $n$ bits of $p$ directly reveals the same amount on $q$.

### 4.1 Noisy Keys: Previous Work

Motivated by some implementation attacks such as side-channel and cold-boot attacks, factoring $N$ from a noisy version of **sk** is an active research line since the pioneering works of Percival [37] and Heninger and Shacham [22].

The number of elements in a noisy version of **sk** depends on the data leak. For example, Percival [37] assumes a noisy version of ($dp$, $dq$) whereas Heninger and Shacham [22] consider different versions of noisy **sk**, such as ($p$, $q$, $d$, $dp$, $dq$) and ($p$, $q$). The number of elements in the noisy **sk** is often denoted in literature as $m$.

Regarding this work, the case of $m = 2$ is particularly interesting because from the Flush+Reload attack described in Section 2.4 we obtain a pair of noisy LS sequences related with $p$ and $q$. Therefore we focus further related work analysis on the $m = 2$ case.

Another implementation attack property is the nature of errors that produces the noisy **sk**—termed *noise model* by Henecka et al. [20]. They defined the noise model of Percival [37] and Heninger and Shacham [22] works as the *erasure model*. In these works, the adversary knows which bit positions contain valid data, and likewise exactly at which bit position data is missing.

On the other hand, Henecka et al. [20] proposed an algorithm for correcting errors in RSA keys from noisy **sk** where some bits are flipped. Correcting errors in this *bit-flip model* is more challenging than in the erasure model [20], however the authors showed that for $m = 2$ it is possible to achieve a success rate of 24 % if the probability of a single bit-flip is less than 0.084. Paterson et al. [36] also studied this error model but considering that a flip $0 \rightarrow 1$ has a different probability than $1 \rightarrow 0$.

Kunihiro [29] analyzed *erasure* and *bit-flip* models in a combined approach proposing an algorithm and a theoretical analysis of the bounds on erasure and bit-flip rates to succeed, improving the allowed bit-flip rate up to 0.11 in an scenario without erasures. See the survey by Kunihiro [30] for more details.

In addition to these error correction algorithms, some authors employs multiple traces to remove noise from **sk**. For example, in very different attack scenarios and completely independent research, Irazoqui et al. [26] and Schwarz et al. [44] obtained an error rate no more than 0.04 from a single trace attack. Later combining the same leakage data from five traces, they corrected all the errors in their respective setting. In contrast to previous approaches that exploit RSA private key element redundancy, correcting errors by using multiple traces is a general approach that applies to any cryptosystem.

While this approach is very attractive, our adversary model forces a single execution trace, so we should follow an error model based approach that exploits the relation $N = pq$ as our side-channel traces leak data about $p$ and $q$.

### 4.2 Noisy LS Sequences

In order to design an algorithm for correcting the errors in an LS sequence, we characterize the nature of them. As discussed in Section 3.3, cache-timing attacks like Flush+Reload provide noisy data due to variances in the execution environment, interruptions, preemptions, task scheduling, etc. Therefore LS sequence extraction from the raw traces is not error free and contains errors with overwhelming probability.

After analyzing many of the traces with known inputs, we identified the following classes of errors.

- Wrong number of 'L' symbols between two 'S' symbols (due to $Z_i$ estimation error).
- Missing 'S' symbols (less frequent).
- Extra 'S' symbols (much less frequent).
- Victim preemption: observed as a small gap (i.e. window of cache misses) in the middle of an operations but fixable by removing this window during trace processing.
- Spy preemptions: observed as high latency peaks in the trace. Unfortunately they are detectable but operations during the preemption window are completely lost.

### 4.3 Leakage Data: Error Modeling

From the Flush+Reload attack, we obtain pairs $(Z_i^p, Z_i^q)$ for $p$ and $q$ respectively. With this information, we obtain two recovery equations according to (2). Where, without losing generality $n_1$ and $n_2$ are greater than some $n$. We therefore have sufficient (noisy) $Z_i$ for both primes to recover their first $n$ bits.

$$\begin{aligned} p &\equiv ex_1 + 1 \mod 2^{n_1+1},\ x_1 = (2^{Z_1^p} + 2^{Z_1^p+Z_2^p} + \cdots + 2^{n_1}) \\ q &\equiv ex_2 + 1 \mod 2^{n_2+1},\ x_2 = (2^{Z_1^q} + 2^{Z_1^q+Z_2^q} + \cdots + 2^{n_2}) \end{aligned} \tag{4}$$

Therefore, the $Z_i$ for a prime $p$ (resp. $q$) defines the ones in the binary representation of $x_1$ (resp. $x_2$). Hence for every value of $n$ we can check if (5) holds.

$$N = (ex_1 + 1)(ex_2 + 1) \mod 2^n \tag{5}$$

This relation is very similar to that employed by Heninger and Shacham [22] for the $m = 2$ case. In their work, errors are in the bit positions of $p$ and $q$ directly while in our case they are instead in the bit positions of a multiple of $p - 1$ and $q - 1$. Irrespective of this difference, it leads to a similar error correction algorithm constructed for correcting some errors.

Regarding existing noise models, our data might have some form of erasures due to spy preemption. However, while *erasure model* considers missing data, at the same time it requires knowing where the missing bits are and they should be distributed at random. Spy preemption fulfills the first condition but it implies a consecutive missing bits/symbols instead of random ones. On the other hand this model does not consider insertions and deletions.

At the same time, the bit-flip model by Henecka et al. [20] does not apply to our case as the largest error source is due to insertions and deletions of zeros between consecutive ones in the binary representation of $x_1$ and $x_2$. It is worth noting that these insertions and deletions generate some bit-flips on $p$ and $q$. However, we verified that even a small insertion/deletion rate of 0.08 implies a bit-flip rate on $p$ and $q$ of about 0.5 due to an avalanche effect. Hence, obtained $p$ and $q$ from their noisy $Z_i$ look like random data.

In this regard, the solution to bit-flipping noise model by Henecka et al. [20] is tightly coupled to the Hamming distance as a metric for filtering out wrong solutions. The algorithm proposed in [20] could be an option to address our noise model, but requires selecting a proper distance metric. We identified the weighted Levenshtein distance as a possible good distance candidate, as it allows assigning different weights for each operations per symbol. Then, it is possible to assign operation per symbol weights to fit a specific noise model. While this is a plausible approach, changing the distance metric inhibits us from using the theoretical and experimental results from [20] to get an estimate on expected success rates. For this reason, we defer this approach to future work and implement an error correction algorithm that does not depend on an specific distance metric.

Inci et al. [25] use another interesting approach to correct errors in RSA keys. They developed an algorithm that recover some errors from a noisy version of $dp$ and $dq$ (i.e. $m = 2$ case). Their noise model has some similarities with ours, however they considered that $dp$ is almost error-free, while in our case we cannot make this assumption. In addition, not much is said about the error rate supported by this algorithm nor its success probability under any error rate.

However, despite that *erasure* and *bit-flip* noise models do not apply directly to our scenario, we borrow the core ideas from Heninger and Shacham [22] and Henecka et al. [20] to build an error correction algorithm that handles the most common errors in our noise model: *insertions and deletions of zeros in the binary representation of* $x_1$ *and* $x_2$.

This relaxed noise model selection is not arbitrary. Our intuition is to use this starting approach as a building block for fixing other error sources. Also, it allows getting a first lower bound on the success rate of the attack and then scaling it (if needed) to support other errors (for example errors in 'S'). In addition, avoiding some specific error handling like spy preemption allows using this correction algorithm in other scenarios for correcting these types of errors in other elements of **sk**.

## 4.4 Error Correction Algorithm

Our algorithm follows the Expand-and-Prune approach [20] as shown in Algorithm 4. The algorithm iterates over all bits from $i \leq n$. It processes a set of candidates $C_i$ at every iteration $i$, starting from a single candidate with the noisy $x_1$, $x_2$. At each iteration, each candidate resulting from the previous iteration expands to several candidates. Then, to avoid candidate space explosion, it prunes these candidates based on rules controlled by the algorithm parameters. Therefore, the configuration parameters of the **Prune** procedure manages the search space growth rate while aiming to increase the probability that the correct solution survives through iterations.

*Expand.* To expand a given candidate $c$ at some bit $i$ (i.e. $c[i]$), consider the selected bit as a branch in a tree. If we construct a search tree, then the possibilities for the bits at any level give rise to new branches in said tree. The tree at any level $i$ contains all the partial solutions $x_1$, $x_2$ up to the $i$-th LSB. At any level $i$ there are (at most) six possible branching candidates that can fulfill (5), listed in Table 1.

**Algorithm 4:** Error correction algorithm

**Input:** $N$, $e$, $Z_i^p$, $Z_i^q$, $n$, config parameters
**Result:** Set $C$ of candidates for corrected $n$ bits of $x_1$ and $x_2$

```
begin
    x1, x2 ← Using Z_i^p, Z_i^q according to (4)
    C_0 ← {(x1, x2)}
    for i = 1 to n − 1 do
        E_i ← ∅
        foreach c ∈ C_{i−1} do
            E_i = E_i ∪ Expand(c, i)
        C_i ← Prune(E_i, params)
    return C_{n−1}
```

**Table 1: Possible branching candidates.**

| Possibilities | Description |
|---|---|
| $(x_1, x_2)$ | (5) holds without changes to $x_1$ or $x_2$ |
| $(x_1 - 0, x_2)$ | Remove a zero at position $i$ from $x_1$; no changes to $x_2$ |
| $(x_1 + 0, x_2)$ | Insert a zero at position $i$ in $x_1$; no changes to $x_2$ |
| $(x_1, x_2 - 0)$ | Remove a zero at position $i$ from $x_2$; no changes to $x_1$ |
| $(x_1, x_2 + 0)$ | Insert a zero at position $i$ in $x_2$; no changes to $x_1$ |
| mult | A combination of changes in both $x_1$ and $x_2$ |

One interesting feature of this algorithm is that even if (5) holds without changing $x_1$ and $x_2$, it still tests the remaining possibilities, including errors that occur at the same index $i$ in $x_1$ and $x_2$—a situation not detected using (5).

Figure 6 shows the expansion and pruning procedure for three consecutive iterations of a candidate $c$ starting at bit $i$ showing the possible candidates. Here we used $\emptyset$ to represent the candidates that did not generate valid solutions. Note how in the first expansion, all possible branching candidates fulfilled Equation 5, however, some of them did not generate viable solutions afterwards or were pruned.

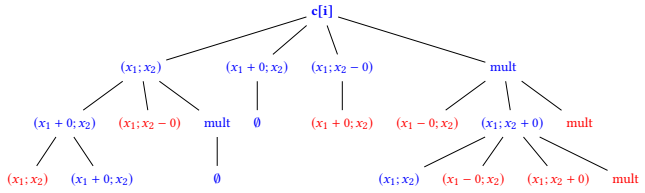


**Figure 6: Expansion process for candidate c[i]. Pruned solutions are in red.**

The candidate pool grows exponentially. We now turn to restricting the number of potential candidates (i.e. the partial solutions) at any level so that finding the correct one is possible through exhaustive search among all solutions within a certain feasible limit, examining situations that control the branching behavior of the tree.

*Prune.* The pruning process applies a set of filters to the expanded candidates $\mathcal{E}_i$. The filter spectrum is very wide and selecting the

best for a given noise model is a challenging task. In this regard, contrary to [20] we implemented a set of filters in a combined approach.

Candidates in $\mathcal{E}_i$ are grouped based on the total amount of errors present in both $x_1$ and $x_2$ at iteration $i$, allowing us to sort the potential candidates based on this parameter.

The most important filters relate to the number of groups ($m$) to keep and the maximum number of candidates in each group ($G$). Denote $e_{\min}$ as the minimum number of total errors found in all candidates in $\mathcal{E}_i$, $e_x$ the number of errors in $x_1$ or $x_2$, and $e_{\text{mult}}$ the number of multiple errors up to iteration $i$. We define the filters as follows:

**Max errors over minimum:** Keep the candidate if $e_{x_1} + e_{x_2} + e_{\text{mult}} \leq e_{\min} + m$ holds.

**Max candidates:** Sort each group based on $(e_{x_1} + e_{x_2} + e_{\text{mult}}, e_{\text{mult}})$, and keep the first $G$ candidates.

The other two filters are very intuitive. The former, defines a maximum number of consecutive errors. This follows the heuristic that higher error densities should be less probable than lower ones, therefore it should be more likely that the correct solution has a smaller error density. The latter helps detect very unlikely solutions—those with an extremely high number of errors.

**Consecutive errors:** Discard sequences having more that a fixed number of consecutive errors.

**Max errors (hard threshold):** Candidates that exceed a maximum number of errors threshold are discarded ($e_{x_1} + e_{x_2} + e_{\text{mult}} \geq e_{\text{th}}$).

We selected the parameters of these filters to keep the probability of pruning the correct solution low, while at the same time keeping the computational requirements affordable for the attacker. In general terms, the adversary can profile the target environment—generating a set of known RSA keys and collecting information about the number of errors, their distribution, etc. for selecting these parameters. We followed this approach for 100 independent RSA-2048 keys and tuned these parameters for our attack environment. We analyzed the number of groups between 5 and 10 and the number of candidates in each group between 5000 and 15000. We set the number of consecutive errors filter to three and based on the observed error rates it is very unlikely that a candidate has more than 150 errors at any iteration, therefore we set the hard threshold to this value.

After this characterization, we observed that 37 traces of 100 fit our reduced noise model: only errors in the number of zeros between ones of $x_1$ and $x_2$. We recovered at least 512 bits from 30 of the test traces, therefore our correction algorithm worked for 80 % of the traces it can handle (reduced noise model). However, as we used these same traces to tune the filter parameters, this value should not be taken as a measure of the success rate of our algorithm. Section 5.2 shows the experimental results for 10K independent traces, and from this large set we extracted the estimate that our error correction algorithm recovers at least 512 bits for 73 % of the traces that met our reduced noise model (see Section 5.2 for more details).

It is worth noting that these success rates and handled error rates are incompatible with other works (e.g. [20]) due to different noise models with respect to previous work. However, we point out that the multiple filter approach that we follow in our algorithm could be an interesting option for addressing other noise models.

*Candidates enumeration.* One important feature of our algorithm is the way it enumerates candidates for checking factors of $N$ (i.e. next stage of our end-to-end attack). As described above, each $C_i$ consists of a fixed number of groups $m$ with at most $G$ possible candidates in each group. Say we have 3 groups with 9 candidates at most in each group, the naïve approach would be to search all possible candidates in each group until finding the solution (Figure 7, top). Based on empirical data, we found that the real solution tends towards the first position of a group. In this case, it makes more sense to consume the candidates using a round robin approach (Figure 7, bottom), giving a higher priority to the highest ranked candidates in each group.

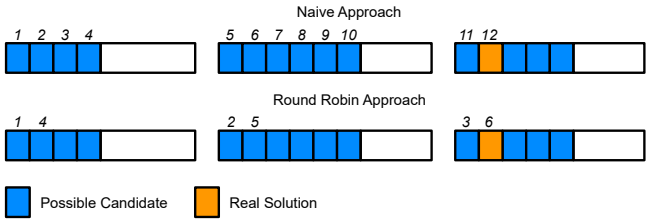


**Figure 7: Candidate enumeration proposals.**

## 5 FACTORING WITH PARTIAL INFORMATION: ENDGAME

In his groundbreaking work, Coppersmith [15] proposed a method to find small solutions of univariate modular equations with modulus having unknown factorization. This result finds many uses in cryptography (mainly in cryptanalysis) as several times in real-world applications an attacker has access to an oracle that gives partial information of a secret and the problem of recovering the remaining part is modeled as a univariate modular equation.

Side-channel attacks play very nicely the oracle role as they often only reveal a (minority) fraction of secret bits. Also, as in our scenario even if it is theoretically possible to fully recover the primes from side-channel traces, it is preferable to only partially recover them to reduce noise influence.

Coppersmith's method reduces the modular equation problem i.e. $f(x) = 0 \bmod p$ to an equation over the integers i.e. $g(x) = 0$ with roots easily found by algorithms like Berlekamp-Zassenhaus. This transformation employs lattice reduction algorithms such as LLL [32] assuming the original modular equation root is small [15].

Following Coppersmith's approach, Howgrave-Graham [24] revisited the lattice construction and proposed a new method to build a lattice that allows obtaining a $g(x) = 0$ from the original modular equation $f(x) = 0 \bmod p$. Howgrave-Graham lattice construction is often preferred due to its simplicity and numerous practical advantages [24, 35].

Coppersmith's result has several implications on RSA security. For an excellent surveys about its impact, we refer readers to [23, 35]. One of these applications is factoring $N$ when half the bits of one prime are known—either the most or the least significant

half. The rest of this section details the Coppersmith approach using the Howgrave-Graham lattice construction (i.e. Coppersmith-Howgrave-Graham method).

*Factoring N knowing LSBs of p.* Assume we known (without loss of generality) the $n$ LSBs of a prime $p$ that is a factor of $N$, i.e. $p$ is expressed as

$$p = \tilde{p}2^n + p_0$$

where $p_0$ is the known portion and $\tilde{p}$ the only unknown. Hence $\tilde{p}$ is a *small* root of the polynomial

$$f(x) = x2^n + p_0 \mod p \quad |\tilde{p}| \leq X$$

and in this case, *small* meaning that $\tilde{p}$ is bound by some known constant $X$. Coppersmith approach requires $f(x)$ to be monic. To achieve that, define $b = 2^{-n} \mod N$, where $b2^n = 1 + kN$ for some integer $k$, then express $f(x)$ as follows.

$$f(x) = x + bp_0 \mod p \quad |\tilde{p}| \leq X \tag{6}$$

Coppersmith-Howgrave-Graham approach aims to solve (6) by reducing this univariate modular equation to an equation over the integers, visualized below.

$$\underbrace{f(x_0) = 0 \bmod p \Rightarrow f_i(x_0) = 0 \bmod p^m \Rightarrow B \overset{\text{LLL}}{\Rightarrow} g(x_0) = 0}_{\text{Coppersmith-Howgrave-Graham}}$$

From the monic polynomial $f(x)$, build a set of $d = m + t$ polynomials $f_i(x)$ over $p^m$ according to the Howgrave-Graham approach, such that these $f_i(x)$ have the same root $x_0 = \tilde{p}$ modulo $p^m$ as $f(x)$ modulo $p$ [24]. Said polynomials are as follows.

$$f_i(x) = N^i f^{m-i}(x) \qquad i = 0, 1, \cdots, m-1$$
$$f_{m+i}(x) = x^i f^m(x) \qquad i = 0, 1, \cdots, t-1$$

The next step builds a lattice $B$ from the $f_i(x)$ for $0 \leq i < d$. Following Howgrave-Graham [24] the basis vectors of $B$ are the coefficient vectors of $f_i(xX)$. Then, lattice-reduced $B$ should yield a $g(x)$ over the integers if the Coppersmith-Howgrave-Graham conditions are respected or heuristically relaxed—we expand later. The small root bound $X$ defines these conditions and the lattice dimension $d = m + t$, therefore we select them such that we can factor $N$ knowing $n$ bits of $p$.

## 5.1 Lattice Parameterization

Three parameters control the effectiveness and efficiency of this method: $X$, $m$ and $t$, where the latter two control the lattice dimension ($d = m + t$) hence the amount of information in it. This dimension dictates the running time of LLL , therefore the goal is to minimize $m$ and $t$ considering that the attacker should run it for each candidate resulting from the error correction phase.

To validate the Coppersmith-Howgrave-Graham approach, we used a public SageMath implementation[4]. The objective of this validation is to obtain—for $n$, a given number of LSBs—which parameter set ($X$, $m$, $t$) yields the right solution with very high probability while minimizing the runtime as much as possible. This approach is very similar to that of Nemec et al. [34], where the authors fixed the bound $X$ and optimize $m$ and $t$—however we also tweak $X$ to get some runtime improvements.

[4]D. Wong, function `coppersmith_howgrave_univariate` [link]

One of the main tasks for using Coppersmith-Howgrave-Graham method is selecting the bound $X$ of the unknown root. Recalling Section 2.1, $N$ of an RSA-2048 key has exactly 2048 bits by forcing the two MSBs of $p$ and $q$ to be set, i.e. they have exactly 1024 bits. Hence for RSA-2048, the inequality (7) holds, where the ordering of $p$ and $q$ is arbitrary.

$$q < \sqrt{N} < p < 2^{1024} < 2\sqrt{N} < N \tag{7}$$

Considering that we known the $n$ LSBs of $p$, we divide (7) by $2^n$ to obtain bounds for $\tilde{p}$.

$$\frac{q}{2^n} < \frac{\sqrt{N}}{2^n} < \tilde{p} < \frac{2^{1024}}{2^n} < \frac{2\sqrt{N}}{2^n} < \frac{N}{2^n}$$

This results in the bound $X = \frac{2\sqrt{N}}{2^n}$ that should work for both primes. However, in practice the Coppersmith conditions are slightly pessimistic, hence in our analysis we also consider $X = \frac{\sqrt{N}}{2^n}$.

*Parameters optimization.* We aim at finding the parameters that solve the partial factorization problem given $n$ LSBs of a prime $p$. We are interested in obtaining this parametrization for different values of $n$, starting from Coppersmith bound for RSA-2048: 512 to 552 bits. The optimization process is as follows: (1) for each $n$, $X$, $m$ and $t$ we generate 100 RSA-2048 keys using OpenSSL and try to recover the remaining bits of both primes; (2) filter out those sets ($X$, $m$ $t$) that do not achieve 100 % success rate; (3) choose the set that minimizes $m + t$ for each $n$.

It is worth noting that each lattice test implies recovering the same key with $p$ and with $q$. This is due to the fact that in our attack scenario (see Section 3.3), the adversary is unaware if the known LSBs correspond to the larger or smaller prime, hence does not know if the bound is respected.

For all values of $n$, the bound $X = \frac{\sqrt{N}}{2^n}$ provides highly probable solutions and sometimes the lattice dimension shrinks by one. At first glance, this lattice reduction might be seen insignificant. But, for example, with $n = 522$ it implies a reduction runtime of roughly 40 s. Indeed this is quick for a single lattice run, but when the number of candidates to test is high (i.e. after error correction) every second counts.

Table 2 summarizes the results of this characterization process for $X = \frac{\sqrt{N}}{2^n}$. One important aspect is that, for either value of $X$, we could not achieve the Coppersmith bound ($n = 512$). However, as pointed out in Section 4 our preliminary simulations suggest that the probability of recovering/correcting 512 bits is roughly the same as 522 bits, i.e. $n = 522$ is adequate in our setting.

We executed this parametrization on Sage 8.1 running on Ubuntu 16.04 on an Intel i7-3770 3.4 GHz. The running times in Table 2 correspond to the average of 100 lattice runs, dominated by the execution of LLL (Sage 8.1 default).

**Table 2: Lattice attack characterization.**

| $n$ | $m$ | $t$ | time (s) |
|---|---|---|---|
| 522 | 26 | 27 | 133.0 |
| 532 | 13 | 14 | 3.0 |
| 542 | 9 | 10 | 0.7 |
| 552 | 6 | 7 | 0.2 |

After this characterization, we obtain a set of parameters for attacking RSA-2048 keys knowing at least $n = 522$ bits of a prime. As we obtained the parameters in Table 2 without considering the ordering of $p$ and $q$, the results are aligned with the error correction phase output that similarly does not contain this information.

## 5.2 Results: End-to-End Attack

To consolidate the attack and validate the successfulness of our techniques and the attack overall, we collected 10K traces from 10K different RSA key generation executions using OpenSSL. Using templates and the Pearson correlation coefficient, we extracted and aligned the GCD operations for $p - 1$ and $q - 1$. Out of those 10K traces, 566 traces were useless due to two main reasons: (1) key generation execution took more time than expected due to failed primality tests; or (2) spy/victim were preempted for a long period of time; thus the spy missed capturing one or both of the GCD operations in any of those cases. The remaining 9434 traces were almost noise free after processing, therefore we were able to perform a horizontal analysis on them to extract a tentative LS sequence of operations, i.e. $Z_i$ values, aiming to recover a minimum of 512 bits per prime value.

We then offloaded the data for these 9434 trials to a cluster for analysis, containing roughly 1500 nodes mixed between Intel E5-2680 (Sandy Bridge), E5-2670 (Sandy Bridge), and E5-2630 (Haswell) cores. In all of the stages that follow, we limited per-job execution times to 4 CPU hours.

The LS sequences contain errors that the subsequent lattice attack cannot tolerate—error correction is required to recover sufficient bits. Our lattice attack can factor RSA-2048 knowing 522 bits of a prime, therefore, we configured our error correction algorithm to recover the same number of bits. Following the algorithm tuning process (see Section 4.4) we selected the set of pruning filter parameters shown in Table 3.

**Table 3: Parameters for error correction algorithm.**

| Parameter | Value |
|---|---|
| Max errors over minimum | 10 |
| Max candidates per group | 15000 |
| Consecutive errors | 3 |
| Max errors (hard threshold) | 150 |

On the cluster, we launched the algorithm for the 9434 traces that contained tentative LS sequences using Table 3 parameters. From this set, 405 traces did not contain enough information to recover 522 bits, leaving 9029 processable traces. For each of the 9434 traces, we also analyzed the ground truth correct sequence to collect data about the probability of recovering a given amount of bits up to 552. Figure 8 shows the resulting survival probability curve.

This survival curve gives an idea about the error correcting algorithm behavior for our large data set. One of the most relevant results is the probability of recovering at least 522 bits: 27.89 % (2632 of 9434). At the same time, the probability is quite close to that of 512 bits: 29.17 %. This small difference confirms the estimation made during the lattice attack parameter optimization: correcting errors up to 522 bits is not significantly more challenging than the 512 bits case. Therefore in our setting, improving lattice attack parameters to Coppersmith's bound (512 bits) will not significantly increase the error correction success rate.

Of interest, Figure 8 shows an abrupt probability drop at the start of the curve. We analyzed it closely and confirmed that roughly 30 % of the traces have a spy preemption just at the beginning, resulting in incomplete traces. It seems there is a bias in our environment that increases the probability of spy preemption at the start of a GCD execution, yet not in the middle. We are investigating the reasons behind this bias, but the fact that the Figure 8 curve does not contain another abrupt drop confirms our bias hypothesis.

After this analysis, our error correction algorithm was able to recover 522 bits for 2632 traces. One interesting metric is the number of errors considering both LS sequences that it handled, and Figure 9 shows the boxplots of the number of errors for recovering various bit quantities. The data suggests the number of errors successfully handled is diverse—for example, recovering 522 bits (rightmost boxplot), this metric ranges from 24 to 108. It implies that our error correction algorithm with Table 3 parameters recovered 522 bits from 2632 traces with error rates that range from 0.02 to 0.10.

Of these 9434 instances, we successfully recovered 2285 private keys after 12875 lattice trials; Figure 10 "Computed" depicts these data points. This represents just over 45 days of CPU time. The remaining 7149 instances break down as follows, which we analyzed using the ground truth private keys. For 347 instances, the partial prime factor remained amongst the candidates, yet had a poor ranking, hence the number of lattice iterations needed exceeded our fixed alloted time (4 h); Figure 10 "Projected" depicts these data points. In these cases, we verified the lattice output at that future iteration indeed yields the intended factor. The remaining 6802 instances failed to retain the correct candidate. Regarding the number of lattice iterations to find a solution, the error correction algorithm with round-robin enumeration achieved an impressive median of one lattice iteration for successful instances.

*End-to-end attack summary.* From a large data set of 9434 independent traces, our end-to-end attack achieves a success rate of 27.89 %. The error correction algorithm was able to fix/recover 522 bits for 2632 traces. At the same time, the lattice attack succeeded for all these 2632 traces, showing the robustness of our lattice attack parameters for 522 bits (Table 2).

While our experiments represent a true proof-of-concept, it is worth noting the success rate is far from optimal. Regarding improvements, we identified several approaches: (1) employ different LS sequence extraction methods to obtain several tentative LS sequences per trace, subsequently executing our error correction algorithm and perform a combined analysis; (2) add support for handling errors in 'S' symbols in the LS sequences; (3) use the error correction algorithm to repair spy preemption. If the spy preemption window is small enough that exhaustive search on the missing LS sequence is feasible, then do so and utilize the error correction algorithm to recover limited bits to obtain a good guess for the most likely correct solution(s). Initial test results following this approach yield promising results to handle the abrupt drop in Figure 8.

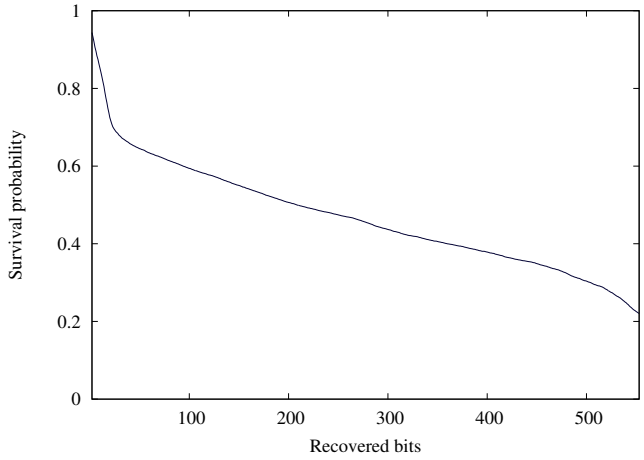


**Figure 8: Bit recovery survival plot.**

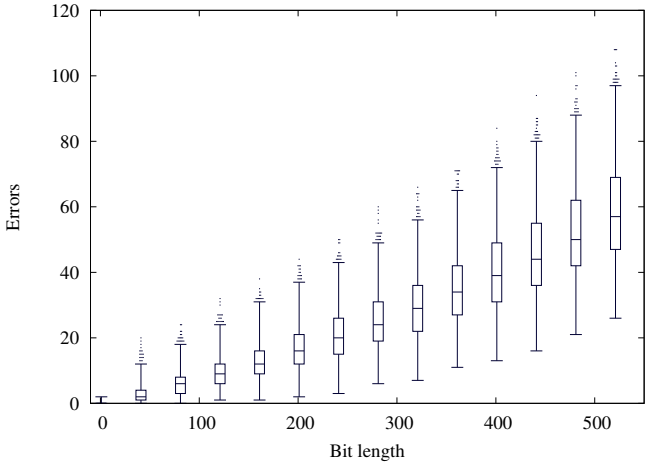


**Figure 9: Number of errors successfully handled by our error correction algorithm.**

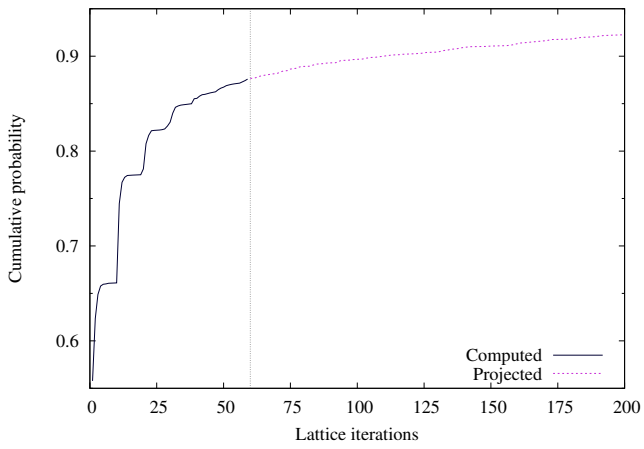


**Figure 10: Lattice iterations for successful instances.**

# 6 CONCLUSION

In this work, we proposed a methodology to analyze cryptographic software for side-channel insecure code path traversal. Applying our methodology to RSA key generation in OpenSSL uncovered three new vulnerabilities, one of which we designed an end-to-end cache-timing attack around, leading to key recovery with good probability and modest computational effort. The attack chain consisted of (1) gathering timings with a combination of Flush+Reload and performance degradation; (2) locating the trace segments of interest (two specific GCD executions) within abundant data; (3) transforming these traces into noisy LS-sequences representing GCD algorithm state; (4) executing our error correction algorithm, resulting in a ranked list of partial prime factor candidates; (5) formulating lattice problems for these candidates that recover the unknown portion; (6) testing if the result yields a prime factor of the RSA modulus $N$, hence the private key.

Executing 10K trials and moving the analysis to a cluster, we achieved roughly a 28 % success rate for full key recovery. We close with lessons learned from our work.

*Lesson 1: Secure by default.* Similar to two recent works [38, 39], two of the vulnerabilities our methodology uncovered are due to insecure default behavior—failure to set a particular flag that, by early exit, diverts the code through algorithms with SCA mitigations. Had the logic been inverted, taking the secure paths by default would have prevented these vulnerabilities. For OpenSSL, these new vulnerabilities continue an unfortunate trend of insecure by default failures that went undetected during unit testing.

*Lesson 2: Knowledge transfer.* Our end-to-end attack exploits only one of the three vulnerabilities our methodology uncovered. The function we targeted is oblivious to the constant-time flag, hence having it set or clear has no effect on our attack. Our root cause analysis (Section 3.1) suggests that the mitigations mainlined as a result of pioneering academic work [2] failed to consider RSA key generation as a whole, and the similarities between GCD computation (which we exploited) and modular inversion with respect to branching behavior went unnoticed when these mitigations were independently developed. This disconnect demonstrates the critical importance of engineers working side-by-side with cryptographers to ensure that academic results reach their intended impact on real-world products.

*Responsible disclosure.* Following responsible disclosure procedures, we reported these issues to OpenSSL and provided fixes, subsequently merged into the OpenSSL codebase after the embargo lifted. OpenSSL assigned CVE-2018-0737 based on our work.

## Acknowledgments

We would like to thank to Matúš Nemec for sharing his results.

We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources.

Supported in part by Academy of Finland grant 303814.

The second author was supported in part by a Nokia Foundation Scholarship and by the Pekka Ahonen Fund through the Industrial Research Fund of Tampere University of Technology.

This article is based in part upon work from COST Action IC1403 CRYPTACUS, supported by COST (European Cooperation in Science and Technology).

## REFERENCES


[1] 2013. *Digital Signature Standard (DSS).* FIPS PUB 186-4. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.186-4

[2] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings*. 185–203. http://dx.doi.org/10.1007/978-3-540-77272-9_12

[3] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. 2017. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic Engineering* 7, 4 (2017), 273–285. https://doi.org/10.1007/s13389-016-0135-4

[4] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. 2017. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *I. J. Circuit Theory and Applications* 45, 2 (2017), 199–213. https://doi.org/10.1002/cta.2283

[5] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. 2016. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, Stephen Schwab, William K. Robertson, and Davide Balzarotti (Eds.). ACM, 422–435. https://doi.org/10.1145/2991079.2991084

[6] Sarang Aravamuthan and Viswanatha Rao Thumparthy. 2007. A Parallelization of ECDSA Resistant to Simple Power Analysis Attacks. In *Proceedings of the Second International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2007), January 7-12, 2007, Bangalore, India*, Sanjoy Paul, Henning Schulzrinne, and G. Venkatesh (Eds.). IEEE. https://doi.org/10.1109/COMSWA.2007.382592

[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 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings (Lecture Notes in Computer Science)*, Lejla Batina and Matthew Robshaw (Eds.), Vol. 8731. Springer, 223–241. https://doi.org/10.1007/978-3-662-44709-3_13

[8] 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 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings (Lecture Notes in Computer Science)*, Wieland Fischer and Naofumi Homma (Eds.), Vol. 10529. Springer, 555–576. https://doi.org/10.1007/978-3-319-66787-4_27

[9] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren. 2013. Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild. In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II (Lecture Notes in Computer Science)*, Kazue Sako and Palash Sarkar (Eds.), Vol. 8270. Springer, 341–360. https://doi.org/10.1007/978-3-642-42045-0_18

[10] Johannes Blömer and Alexander May. 2003. New Partial Key Exposure Attacks on RSA. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings (Lecture Notes in Computer Science)*, Dan Boneh (Ed.), Vol. 2729. Springer, 27–43. https://doi.org/10.1007/978-3-540-45146-4_2

[11] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. 2006. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive* 2006 (2006), 52. http://eprint.iacr.org/2006/052

[12] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings (Lecture Notes in Computer Science)*, Mitsuru Matsui (Ed.), Vol. 5912. Springer, 667–684. https://doi.org/10.1007/978-3-642-10366-7_39

[13] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. 2002. Template Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers (Lecture Notes in Computer Science)*, Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar (Eds.), Vol. 2523. Springer, 13–28. https://doi.org/10.1007/3-540-36400-5_3

[14] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. 2010. Horizontal Correlation Analysis on Exponentiation. In *Information and Communications Security - 12th International Conference, ICICS 2010, Barcelona, Spain, December 15-17, 2010. Proceedings (Lecture Notes in Computer Science)*, Miguel Soriano, Sihan Qing, and Javier López (Eds.), Vol. 6476. Springer, 46–61. https://doi.org/10.1007/978-3-642-17650-0_5

[15] Don Coppersmith. 1996. Finding a Small Root of a Univariate Modular Equation. In *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding (Lecture Notes in Computer Science)*, Ueli M. Maurer (Ed.), Vol. 1070. Springer, 155–165. https://doi.org/10.1007/3-540-68339-9_14

[16] Don Coppersmith. 1997. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *J. Cryptology* 10, 4 (1997), 233–260. https://doi.org/10.1007/s001459900030

[17] 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, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings (Lecture Notes in Computer Science)*, Christophe Clavier and Kris Gaj (Eds.), Vol. 5747. Springer, 141–155. https://doi.org/10.1007/978-3-642-04138-9_11

[18] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 845–858. https://doi.org/10.1145/3133956.3134029

[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 897–912. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[20] Wilko Henecka, Alexander May, and Alexander Meurer. 2010. Correcting errors in RSA private keys. In *Annual Cryptology Conference*. Springer, 351–369.

[21] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 205–220. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger

[22] Nadia Heninger and Hovav Shacham. 2009. Reconstructing RSA private keys from random key bits. In *Advances in Cryptology-CRYPTO 2009*. Springer, 1–17.

[23] M. Jason Hinek. 2009. *Cryptanalysis of RSA and its variants.* CRC press.

[24] Nick Howgrave-Graham. 1997. Finding Small Roots of Univariate Modular Equations Revisited. In *Cryptography and Coding, 6th IMA International Conference, Cirencester, UK, December 17-19, 1997, Proceedings (Lecture Notes in Computer Science)*, Michael Darnell (Ed.), Vol. 1355. Springer, 131–142. https://doi.org/10.1007/BFb0024458

[25] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 368–388.

[26] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 353–364.

[27] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. 2010. Factorization of a 768-Bit RSA Modulus. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tal Rabin (Ed.), Vol. 6223. Springer, 333–350. https://doi.org/10.1007/978-3-642-14623-7_18

[28] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[29] Noburu Kunihiro. 2015. An Improved Attack for Recovering Noisy RSA Secret Keys and Its Countermeasure. https://doi.org/10.1007/978-3-319-26059-4

[30] Noburu Kunihiro. 2018. *Matematical Mdelling Next-Generation Cryptography*. Springer, Chapter Mathematical Approach for Recovering Secret Key from Its Noisy Version. https://doi.org/10.1007/978-981-10-5065-7

[31] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. 2012. Public Keys. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings (Lecture Notes in Computer Science)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.), Vol. 7417. Springer, 626–642. https://doi.org/10.1007/978-3-642-32009-5_37

[32] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. 1982. Factoring polynomials with rational coefficients. *Math. Ann.* 261, 4 (1982), 515–534.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *CoRR* abs/1801.01207 (2018). arXiv:1801.01207 http://arxiv.org/abs/1801.01207

[34] Matús Nemec, Marek Sýs, Petr Svenda, Dusan Klinec, and Vashek Matyas. 2017. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1631–1648. https://doi.org/10.1145/3133956.3133969

[35] Phong Q. Nguyen and Brigitte Vallée (Eds.). 2010. *The LLL Algorithm - Survey and Applications*. Springer. https://doi.org/10.1007/978-3-642-02295-1

[36] Kenneth G Paterson, Antigoni Polychroniadou, and Dale L Sibborn. 2012. A coding-theoretic approach to recovering noisy RSA keys. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 386–403.

[37] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*. http://www.daemonology.net/papers/cachemissing.pdf

[38] Cesar Pereida García and Billy Bob Brumley. 2017. Constant-Time Callees with Variable-Time Callers. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 83–98. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia

[39] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1639–1650. https://doi.org/10.1145/2976749.2978420

[40] J. M. Pollard. 1974. Theorems on factorization and primality testing. *Mathematical Proceedings of the Cambridge Philosophical Society* 76, 3 (1974), 521âĂŞ528. https://doi.org/10.1017/S0305004100049252

[41] J. M. Pollard. 1975. A Monte Carlo method for factorization. *BIT Numerical Mathematics* 15, 3 (01 Sep 1975), 331–334. https://doi.org/10.1007/BF01933667

[42] Michael O. Rabin. 1980. Probabilistic algorithm for testing primality. *J. Number Theory* 12, 1 (1980), 128–138. https://doi.org/10.1016/0022-314X(80)90084-0

[43] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM* 21, 2 (1978), 120–126. https://doi.org/10.1145/359340.359342

[44] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24.

[45] Josef Stein. 1967. Computational problems associated with Racah algebra. *J. Comput. Phys.* 1, 3 (1967), 397–405. https://doi.org/10.1016/0021-9991(67)90047-2

[46] Camille Vuillaume, Takashi Endo, and Paul Wooderson. 2012. RSA Key Generation: New Attacks. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings (Lecture Notes in Computer Science)*, Werner Schindler and Sorin A. Huss (Eds.), Vol. 7275. Springer, 105–119. https://doi.org/10.1007/978-3-642-29912-4_9

[47] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. 2018. Single trace attack against RSA key generation in Intel SGX SSL. In *Proceedings of the 2018 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Sungdo, Incheon, Korea, June 4-8, 2018*. ACM. https://rspreitzer.github.io/publications/proc/asiaccs-2018-paper-1.pdf to appear.

[48] Michael J. Wiener. 1990. Cryptanalysis of short RSA secret exponents. *IEEE Trans. Information Theory* 36, 3 (1990), 553–558. https://doi.org/10.1109/18.54902

[49] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 640–656. https://doi.org/10.1109/SP.2015.45

[50] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[51] 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 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings (Lecture Notes in Computer Science)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.), Vol. 9813. Springer, 346–367. https://doi.org/10.1007/978-3-662-53140-2_17