Constant-Time Callees with Variable-Time Callers

Cesar Pereida García Billy Bob Brumley
Tampere University of Technology
{*cesar.pereidagarcia,billy.brumley*}*@tut.fi*

Abstract

Side-channel attacks are a serious threat to security-critical software. To mitigate remote timing and cache-timing attacks, many ubiquitous cryptography software libraries feature constant-time implementations of cryptographic primitives. In this work, we disclose a vulnerability in OpenSSL 1.0.1u that recovers ECDSA private keys for the standardized elliptic curve P-256 despite the library featuring both constant-time curve operations and modular inversion with microarchitecture attack mitigations. Exploiting this defect, we target the errant modular inversion code path with a cache-timing and improved performance degradation attack, recovering the inversion state sequence. We propose a new approach of extracting a variable number of nonce bits from these sequences, and improve upon the best theoretical result to recover private keys in a lattice attack with as few as 50 signatures and corresponding traces. As far as we are aware, this is the first timing attack against OpenSSL ECDSA that does not target scalar multiplication, the first side-channel attack on cryptosystems leveraging P-256 constant-time scalar multiplication and furthermore, we extend our attack to TLS and SSH protocols, both linked to OpenSSL for P-256 ECDSA signing.

Keywords: applied cryptography; elliptic curve cryptography; digital signatures; side-channel analysis; timing attacks; cache-timing attacks; performance degradation; ECDSA; modular inversion; binary extended Euclidean algorithm; lattice attacks; constant-time software; OpenSSL; NIST P-256; CVE-2016-7056

1 Introduction

Being a widely-deployed open-source cryptographic library, OpenSSL is a popular target for different cryptanalytic attacks, including side-channel attacks that target cryptosystem implementation weaknesses that can leak critical algorithm state. As a software library, OpenSSL provides not only TLS functionality but also cryptographic functionality for applications such as SSH, IPSec, and VPNs.

Due to its ubiquitous usage, OpenSSL contains arguably one of the most popular software implementations of the Elliptic Curve Digital Signature Algorithm (ECDSA). OpenSSL's scalar multiplication algorithm was shown vulnerable to cache-timing attacks in 2009 [6], and attacks continue on the same code path to this date [2, 4, 10, 27]. Recognizing and responding to the threat cache-timing attacks pose to cryptosystem implementations, OpenSSL mainlined constant-time scalar multiplication for several popular standardized curves already in 2011 [16].

In this work, we disclose a software defect in the OpenSSL (1.0.1 branch) ECDSA implementation that allows us to design and implement a side-channel cache-timing attack to recover private keys. Different from previous work, our attack focuses on the modular inversion operation instead of the typical scalar multiplication, thus allowing us to target the standardized elliptic curve P-256, circumventing its constant-time scalar multiplication implementation. The root cause of the defect is failure to set a flag in ECDSA signing nonces that indicates only constant-time code paths should be followed.

We leverage the state-of-the-art FLUSH+RELOAD [28] technique to perform our cache-timing attack. We adapt the technique to OpenSSL's implementation of ECDSA and the *Binary Extended Euclidean Algorithm* (BEEA). Our spy program probes relevant memory addresses to create a timing signal trace, then the signal is processed and converted into a sequence of right-shift and subtraction (LS) operations corresponding to the BEEA execution state from which we extract bits of information to create a lattice problem. The solution to the lattice problem yields the ECDSA secret key.

We discover that observing as few as 5 operations

from the LS sequence allows us to use every single captured trace for our attack. This significantly reduces both the required amount of signatures and side-channel data compared to previous work [8], and maintains a good signature to lattice dimension ratio.

We build upon the performance degradation technique of Allan et al. [2] to efficiently find the memory addresses with the highest impact to the cache during the degrading attack. This new approach allows us to accurately find the best candidate memory addresses to slow the modular inversion by an average factor of 18, giving a high resolution trace and allowing us to extract the needed bits of information from all of the traces.

Unlike previous works targeting the *w*NAF scalar multiplication code path (for curves such as BitCoin's secp256k1) or performing theoretical side-channel analysis of the BEEA, we are the first to demonstrate a practical cache-timing attack against the BEEA modular inversion, and furthermore OpenSSL's ECDSA signing implementation with constant-time P-256 scalar multiplication.

Our contributions in this work include the following:

- We identify a bug in OpenSSL that allows a cache-timing attack on ECDSA signatures, despite constant-time P-256 scalar multiplication. (Section 3)
- We describe a new quantitative approach that accurately identifies the most accessed victim memory addresses w.r.t. data caching, then we use them for an improved performance degradation attack in combination with the FLUSH+RELOAD technique. (Section 4.1)
- We describe how to combine the FLUSH+RELOAD technique with the improved performance degradation attack to recover side-channel traces and algorithm state from the BEEA execution. (Section 4)
- We present an alternate approach to recovering nonce bits from the LS sequences, focused on minimizing required side-channel information. Using this approach, we recover bits of information from every trace, allowing us to use every signature query to construct and solve a lattice problem, revealing the secret key with as few as 50 signatures and corresponding traces. (Section 4.2)
- We perform a key-recovery cache-timing attack on the TLS and SSH protocols utilizing OpenSSL for ECDSA functionality. (Section 5)

2 Background

2.1 Elliptic Curve Cryptography

ECC. Developed in the mid 1980's, elliptic curves were introduced to cryptography by Miller [20] and Koblitz [17] independently. Elliptic Curve Cryptography (ECC) became popular mainly for two important reasons: no sub-exponential time algorithm to solve the elliptic curve discrete logarithm problem is known for well-chosen parameters and it operates in the group of points on an elliptic curve, compared to the classic multiplicative group of a finite field, thus allowing the use of smaller parameters to achieve the same security levels—consequently smaller keys and signatures.

Although there are more general forms of elliptic curves, for the purposes of this paper we restrict to short Weierstrass curves over prime fields. With prime $p > 3$, all of the $x, y \in GF(p)$ solutions the equation

$$E : y^2 = x^3 + ax + b$$

along with an identity element form an abelian group. Parameters of interest here are the NIST standard curves that set $a = -3$ and p a Mersenne-like prime, both chosen for their performance characteristics.

ECDSA. Throughout this paper, we use the following notation for the Elliptic Curve Digital Signature Algorithm (ECDSA).

Parameters: A generator $G \in E$ of an elliptic curve group of prime order n and an approved hash function h (e.g. SHA-1, SHA-256, SHA-512).

Private-Public key pairs: The private key α is an integer uniformly chosen from $\{1..n-1\}$ and the corresponding public key $D = [\alpha]G$ where $[i]G$ denotes scalar-by-point multiplication using additive group notation. Calculating the private key given the public key requires solving the elliptic curve discrete logarithm problem and for correctly chosen parameters, this is an intractable problem.

Signing: A given party, Alice, wants to send a signed message m to Bob. Using her private-public key pair (α_A, D_A), Alice performs the following steps:

1. Select uniformly at random a secret nonce k such that $0 < k < n$.
2. Compute $r = ([k]G)_x \bmod n$.
3. Compute $s = k^{-1}(h(m) + \alpha_A r) \bmod n$.
4. Alice sends (m, r, s) to Bob.

Verifying: Bob wants to be sure the message he received comes from Alice—a valid ECDSA signature gives strong evidence of authenticity. Bob performs the following steps to verify the signature:

1. Reject the signature if it does not satisfy $0 < r < n$ and $0 < s < n$.
2. Compute $w = s^{-1} \bmod n$ and $h(m)$.
3. Compute $u_1 = h(m)w \bmod n$ and $u_2 = rw \bmod n$.
4. Compute $(x, y) = [u_1]G + [u_2]D_A$.
5. Accept the signature if and only if $x = r \bmod n$ holds.

2.2 Side-Channel Attacks

Thanks to the adoption of ECC and the increasing use of digital signatures, ECDSA has become a popular algorithm choice for digital signatures. ECDSA's popularity makes it a good target for side-channel attacks.

At a high level, an established methodology for ECDSA is to query multiple signatures, then partially recover nonces k_i from the side-channel, leading to a bound on the value $\alpha t_i - u_i$ that is shorter than the interval $\{1..n-1\}$ for some known integers t_i and u_i. This leads to a version of the Hidden Number Problem (HNP) [5]: recover α given many (t_i, u_i) pairs. The HNP instances are then reduced to Closest Vector Problem (CVP) instances, solved with lattice methods.

Over the past decade, several authors have described practical side-channel attacks on ECDSA that exploit partial nonce disclosure by different microprocessor features to recover long-term private keys.

Brumley and Hakala [6] describe the first practical side-channel attack against OpenSSL's ECDSA implementation. They use the EVICT+RELOAD strategy and an L1 data cache-timing attack to recover the LSBs of ECDSA nonces from the library's *w*NAF (a popular low-weight signed-digit representation) scalar multiplication implementation in OpenSSL 0.9.8k. After collecting 2,600 signatures (8K with noise) from the `dgst` command line tool and using the Howgrave-Graham and Smart [15] lattice attack, the authors recover a 160-bit ECDSA private key from standardized curve secp160r1.

Brumley and Tuveri [7] attack ECDSA with binary curves in OpenSSL 0.9.8o. Mounting a remote timing attack, the authors show the library's Montgomery Ladder scalar multiplication implementation leaks timing information on the MSBs of the nonce used and after collecting that information over 8,000 TLS handshakes a 162-bit NIST B-163 private key can be recovered with lattice methods.

Benger et al. [4] target OpenSSL's *w*NAF implementation and 256-bit private keys for the standardized GLV curve [11] secp256k1 used in the BitCoin protocol. Using as few as 200 ECDSA signatures and the FLUSH+RELOAD technique [28], the authors find some LSBs of the nonces and extend the lattice technique of [21, 22] to use a varying amount of leaked bits rather than limiting to a fixed number.

van de Pol et al. [27] attack OpenSSL's 1.0.1e *w*NAF implementation for the curve secp256k1. Leveraging the structure of the modulus n, the authors use more information leaked in consecutive sequences of bits anywhere in the top half of the nonces, allowing them to recover the secret key after observing as few as 25 ECDSA signatures.

Allan et al. [2] improve on previous results by using a performance-degradation attack to amplify the side-channel. This amplification allows them to additionally observe the sign bit of digits in the *w*NAF representation used in OpenSSL 1.0.2a and to recover secp256k1 private keys after observing only 6 signatures.

Fan et al. [10] increase the information extracted from each signature by analyzing the *w*NAF implementation in OpenSSL. Using the curve secp256k1 as a target, they perform a successful attack after observing as few as 4 signatures.

Our work differs from previous ECDSA side-channel attacks in two important ways. (1) We focus on NIST standard curve P-256, featured in ubiquitous security standards such as TLS and SSH. Later in Section 2.4, we explain the reason previous works were unable to target this extremely relevant curve. (2) We do not target the scalar-by-point multiplication operation (i.e. the bottleneck of the signing algorithm), but instead Step 3 of the signing algorithm, the modular inversion operation.

2.3 Binary Extended Euclidean Algorithm

The modular inversion operation is one of the most basic and essential operations required in public key cryptography. Its correct implementation and constant-time execution has been a recurrent topic of research [1, 3, 8].

A well known algorithm used for modular inversion is the Euclidean Extended Algorithm and in practice is often substituted by a variant called the *Binary Extended Euclidean Algorithm (BEEA)* [18, Chap. 14.4.3]. This variant replaces costly division operations by simple right-shift operations, thus, achieving performance benefits over the regular version of the algorithm. BEEA is particularly efficient for very long integers—e.g. RSA, DSA, and ECDSA operands.

Figure 1 shows the BEEA. Note that in each iteration only one u or v while-loop is executed, but not both. Additionally, in the very first iteration only the u while-loop can be executed since v is a copy of p which is a large prime integer n for ECDSA.

In 2007, independent research done by Aciiçmez et al. [1], Aravamuthan and Thumparthy [3] demonstrated side-channel attacks against the BEEA. Aravamuthan and Thumparthy [3] attacked BEEA using Power Analysis attacks, whereas Aciiçmez et al. [1] attacked BEEA through Simple Branch Prediction Analysis (SBPA), demonstrating the fragility of this algorithm against side-channel attacks.

Both previous works reach the conclusion that in order to reveal the value of the nonce k, it is necessary to identify four critical input-dependent branches leaking information, namely:

1. Number of right-shift operations performed on v.

Input: Integers k and p such that $gcd(k,p) = 1$.
Output: $k^{-1} \bmod p$.

```
v ← p, u ← k, X ← 1, Y ← 0
while u ≠ 0 do
    while even(u) do
        u ← u/2                          /* u loop */
        if odd(X) then X ← X + p
        X ← X/2
    while even(v) do
        v ← v/2                          /* v loop */
        if odd(Y) then Y ← Y + p
        Y ← Y/2
    if u ≥ v then
        u ← u − v
        X ← X − Y
    else
        v ← v − u
        Y ← Y − X
return Y mod p
```

Figure 1: Binary Extended Euclidean Algorithm.

2. Number of right-shift operations performed on u.
3. Number and order of subtractions $u := u - v$.
4. Number and order of subtractions $v := v - u$.

Moreover, both works present a BEEA reconstruction algorithm that allows them to fully recover the nonce k—and therefore the secret signing key—given a perfect side-channel trace that distinguish the four critical branches.

Aravamuthan and Thumparthy [3] argue that a countermeasure to secure BEEA against side-channel attacks is to render u and v subtraction branches indistinguishable, thus the attack is computationally expensive to carry out. As a response, Cabrera Aldaya et al. [8] demonstrated a Simple Power Analysis (SPA) attack against a custom implementation of the BEEA. The authors' main contribution consists of demonstrating it is possible to partially determine the order of subtractions on branches u and v only by knowing the number of right-shift operations performed in every while-loop iteration. Under a perfect SPA trace, the authors use an algebraic algorithm to determine a short execution sequence of u and v subtraction branches.

They manage to recover various bits of information for several ECDSA key sizes. The authors are able to recover information only from some but not all of their SPA traces by using their algorithm and the partial information about right-shift and subtraction operations. Finally, using a lattice attack they recover the secret signing key.

As can be seen from the previous works, depending on the identifiable branches in the trace and quality of the trace it is possible to recover full or partial information about the nonce k. Unfortunately, the information leaked by most of the real world side-channels does not allow us to differentiate between subtraction branches u and v, therefore limiting the leaked information to three input-dependent branches:

1. Number of right-shift operations performed on v.
2. Number of right-shift operations performed on u.
3. Number of subtractions.

2.4 OpenSSL History

OpenSSL has a rich and storied history as a prime security attack target [19], a distinction ascribed to the library's ubiquitous real world application. One of the main contributions of our work is identifying a new OpenSSL vulnerability described later in Section 3. To understand the nature of this vulnerability and facilitate root cause analysis, in this section we give a brief overview of side-channel defenses in the OpenSSL library, along with some context and insight into what prompted these code changes. Table 1 summarizes the discussion.

0.9.7. Side-channel considerations started to induce code changes in OpenSSL starting with the 0.9.7 branch. The RSA cache-timing attack by Percival [23] recovered secret exponent bits used as lookup table indices in sliding window exponentiation using an EVICT+RELOAD strategy on HyperThreading architectures. His work prompted introduction of the `BN_FLG_CONSTTIME` flag, with the intention of allowing special security treatment of BIGNUMs having said flag set. At the time—and arguably still—the most important use case of the flag is modular exponentiation. Introduced alongside the flag, the `BN_mod_exp_mont_consttime` function is a fixed-window modular exponentiation algorithm featuring data cache-timing countermeasures. Recent research brings the security of this solution into question [29].

0.9.8. The work by Aciiçmez et al. [1] targeting BEEA prompted the introduction of the `BN_mod_inverse_no_branch` function, an implementation with more favorable side-channel properties than that of BEEA. The implementation computes modular inversions in a way that resembles the classical extended Euclidean algorithm, calculating quotients and remainders in each step by calling `BN_div` updated to respect the `BN_FLG_CONSTTIME` flag. Tracking callers to `BN_mod_inverse`, the commit[1] enables the `BN_FLG_CONSTTIME` across several cryptosystems where the modular inversion inputs were

[1] https://github.com/openssl/openssl/commit/bd31fb21454609b125ade1ad569ebcc2a2b9b73c

deemed security critical, notably the published attack targeting RSA.

1.0.1. Based on the work by Käsper [16], the 1.0.1 branch introduced constant-time scalar multiplication implementations for several popular elliptic curves. This code change was arguably motivated by the data cache-timing attack of Brumley and Hakala [6] against OpenSSL that recovered digits of many ECDSA nonces during scalar multiplication on HyperThreading architectures using the EVICT+RELOAD strategy. This information was then used to construct a lattice problem and calculate ECDSA private keys. The commit[2] included several new `EC_METHOD` implementations, of which arguably `EC_GFp_nistp256_method` has the most real world application to date. This new scalar multiplication implementation uses fixed-window combing combined with secure table lookups via software multiplexing (masking), and is enabled with the `ec_nistp_64_gcc_128` option at build time. For example, Debian 8.0 "Jessie" (current LTS, not EOL) and 7.0 "Wheezy" (previous LTS, not EOL) and Ubuntu 14.04 "Trusty" (previous LTS, not EOL) enable said option when possible for their OpenSSL 1.0.1 package builds. From the side-channel attack perspective, we note that this change is the reason academic research (see Section 2.2) shifted to the secp256k1 curve—NIST P-256 no longer takes the generic *w*NAF scalar multiplication code path like secp256k1.

1.0.2. Motivated by performance and the potential to utilize Intel AVX extensions, a contribution by Gueron and Krasnov [14] included fast and secure curve P-256 operations with their custom `EC_GFp_nistz256_method`. Here we focus on a cherry picked commit[3] that affected the ECDSA sign code path for all elliptic curves. While speed motivated the contribution, Möller observes[4]: "It seems that the `BN_MONT_CTX`-related code (used in `crypto/ecdsa` for constant-time signing) is entirely independent of the remainder of the patch, and should be considered separately." Gueron confirms: "The optimization made for the computation of the modular inverse in the ECDSA sign, is using const-time mod-exp. Indeed, this is independent of the rest of the patch, and it can be used independently (for other usages of the library). We included this addition in the patch for the particular usage in ECDSA." Hence following this code change, ECDSA signing for all curves now compute modular inversion via `BN_mod_exp_mont_consttime` and Fermat's Little Theorem (FLT).

[2] https://github.com/openssl/openssl/commit/3e00b4c9db42818c621f609e70569c7d9ae85717

[3] https://github.com/openssl/openssl/commit/8aed2a7548362e88e84a7feb795a3a97e8395008

[4] https://rt.openssl.org/Ticket/Display.html?id=3149&user=guest&pass=guest

Table 1: OpenSSL side-channel defenses across versions. Although `BN_mod_exp_mont_consttime` was introduced in the 0.9.7 branch, here we are referring to its use for modular inversion via FLT.

OpenSSL version	0.9.6	0.9.7	0.9.8	1.0.0	1.0.1	1.0.2
`BN_mod_inverse`	✓	✓	✓	✓	✓	✓
`BN_FLG_CONSTTIME`	—	✓	✓	✓	✓	✓
`BN_mod_inverse_no_branch`	—	—	✓	✓	✓	✓
`ec_nistp_64_gcc_128`	—	—	—	—	✓	✓
`BN_mod_exp_mont_consttime`	—	—	—	—	—	✓
`EC_GFp_nistz256_method`	—	—	—	—	—	✓

3 A New Vulnerability

From Table 1, starting with 1.0.1 the reasonable expectation is that cryptosystems utilizing P-256 resist timing attacks, whether they be remote, data cache, instruction cache, or branch predictor timings. We focus here on the combination of ECDSA and P-256 within the library. The reason this is a reasonable expectation is that `ec_nistp_64_gcc_128` provides constant-time scalar multiplication to protect secret scalar nonces, and `BN_mod_inverse_no_branch` provides microarchitecture attack defenses when inverting these nonces. For ECDSA, these are the two most critical locations where the secret nonce is an operand—to produce r and s, respectively.

The vulnerability we now disclose stems from the changes introduced in the 0.9.8 branch. The `BN_mod_inverse` function was modified to first check the `BN_FLG_CONSTTIME` flag of the BIGNUM operands—if set, the function then early exits to `BN_mod_inverse_no_branch` to protect the security-sensitive inputs. If the flag is not set, i.e. inputs are not secret, the control flow continues to the stock BEEA implementation.

Paired with this code change, the next task was to identify callers to `BN_mod_inverse` within the library, and enable the `BN_FLG_CONSTTIME` flag for BIGNUMs in cryptosystem implementations that are security-sensitive. Our analysis suggests this was done by searching the code base for uses of the `BN_FLG_EXP_CONSTTIME` flag that was replaced with `BN_FLG_CONSTTIME` as part of the changeset, given the evolution of constant-time as concept within OpenSSL and no longer limited to modular exponentiation. As a result, the code changes permeated RSA, DSA, and Diffie-Hellman implementations, but not ECC-based cryptosystems such as ECDH and ECDSA.

This leaves a gap for 1.0.1 with respect to ECDSA. While `ec_nistp_64_gcc_128` provides constant-time scalar multiplication to compute the r component of P-256 ECDSA signatures, the s component will compute modular inverses of security-critical nonces with the stock `BN_mod_inverse` function, not taking the `BN_mod_inverse_no_branch` code path. In the end, the root cause is that the ECDSA signing implementation does

```
   +--bn_gcd.c-------------------------------------------------------------------+
   |226     BIGNUM *BN_mod_inverse(BIGNUM *in,                                  |
   |227                            const BIGNUM *a, const BIGNUM *n, BN_CTX *ctx) |
   |228     {                                                                   |
B+ |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         }                                                               |
   +-----------------------------------------------------------------------------+
   |0x7ffff77da1c7 <BN_mod_inverse+56>  mov   -0x90(%rbp),%rax                  |
   |0x7ffff77da1ce <BN_mod_inverse+63>  mov   0x14(%rax),%eax                   |
   |0x7ffff77da1d1 <BN_mod_inverse+66>  and   $0x4,%eax                         |
   |0x7ffff77da1d4 <BN_mod_inverse+69>  test  %eax,%eax                         |
   |0x7ffff77da1d6 <BN_mod_inverse+71>  jne   0x7ffff77da1e9 <BN_mod_inverse+90>  |
   |0x7ffff77da1d8 <BN_mod_inverse+73>  mov   -0x98(%rbp),%rax                  |
   |0x7ffff77da1df <BN_mod_inverse+80>  mov   0x14(%rax),%eax                   |
   |0x7ffff77da1e2 <BN_mod_inverse+83>  and   $0x4,%eax                         |
   |0x7ffff77da1e5 <BN_mod_inverse+86>  test  %eax,%eax                         |
  >|0x7ffff77da1e7 <BN_mod_inverse+88>  je    0x7ffff77da212 <BN_mod_inverse+131> |
   +-----------------------------------------------------------------------------+
native process 3399 In: BN_mod_inverse                    L234  PC: 0x7ffff77da1e7
(gdb) run dgst -sha256 -sign prime256v1.pem -out lsb-release.sig /etc/lsb-release
Starting program: /usr/local/ssl/bin/openssl dgst -sha256 -sign prime256v1.pem ...
Breakpoint 1, BN_mod_inverse (...) at bn_gcd.c:229
(gdb) backtrace
#0  BN_mod_inverse (...) at bn_gcd.c:229
#1  0x00007ffff782aed9 in ecdsa_sign_setup (...) at ecs_ossl.c:182
#2  0x00007ffff782bc35 in ECDSA_sign_setup (...) at ecs_sign.c:105
#3  0x00007ffff782b29a in ecdsa_do_sign (...) at ecs_ossl.c:269
#4  0x00007ffff782bafd in ECDSA_do_sign_ex (...) at ecs_sign.c:74
#5  0x00007ffff782bb97 in ECDSA_sign_ex (...) at ecs_sign.c:89
#6  0x00007ffff782bb44 in ECDSA_sign (...) at ecs_sign.c:80 ...
(gdb) stepi
(gdb) macro expand BN_get_flags(a, BN_FLG_CONSTTIME)
expands to: ((a)->flags&(0x04))
(gdb) print BN_get_flags(a, BN_FLG_CONSTTIME)
$1 = 0
(gdb) print BN_get_flags(n, BN_FLG_CONSTTIME)
$2 = 0
```

Figure 2: Modular inversion within OpenSSL 1.0.1u (built with `ec_nistp_64_gcc_128` enabled) for P-256 ECDSA signing. Operands `a` and `n` are the nonce and generator order, respectively. The early exit to `BN_mod_inverse_no_branch` never takes place, since the caller `ecdsa_sign_setup` fails to set the `BN_FLG_CONSTTIME` flag on the operands. Control flow continues to the stock, classical BEEA implementation.

not set the `BN_FLG_CONSTTIME` flag for nonces. Scalar multiplication with `ec_nistp_64_gcc_128` is oblivious to this flag and always treats single scalar inputs as security-sensitive, yet `BN_mod_inverse` requires said flag to take the new secure code path.

Figure 2 illustrates this vulnerability running in OpenSSL 1.0.1u. The caller function `ecdsa_sign_setup` contains the bulk of the ECDSA signing cryptosystem—generating a nonce, computing the scalar multiple, inverting the nonce, computing r, and so on. When control flow reaches callee `BN_mod_inverse`, inputs `a` and `n` are the nonce and generator order, respectively. Stepping by instruction, it shows that the call to `BN_mod_inverse_no_branch` never takes place, since the `BN_FLG_CONSTTIME` flag is not set for either of these operands. Failing this security critical branch, the control flow continues to the stock, classical BEEA implementation.

3.1 Forks

OpenSSL is not the only software library affected by this vulnerability. Following HeartBleed, OpenBSD forked OpenSSL to LibreSSL in July 2014, and Google forked OpenSSL to BoringSSL in June 2014. We now discuss this vulnerability within the context of these two forks.

LibreSSL. An 04 Nov 2016 commit[5] cherry picked the `EC_GFp_nistz256_method` for LibreSSL. Interestingly, LibreSSL is the library most severely affected by this vulnerability. The reason is they did not cherry pick the `BN_mod_exp_mont_consttime` ECDSA nonce inversion. That is, as of this writing (fixed during disclosure) the current LibreSSL master branch can feature constant-time P-256 scalar multiplication with either `EC_GFp_nistz256_method` or `EC_GFp_nistp256_method` callees depending on compile-time options and minor code changes, but inverts all ECDSA nonces with the `BN_mod_inverse` callee that fails the same security critical branch as OpenSSL, due to the caller `ecdsa_sign_setup` not setting the `BN_FLG_CONSTTIME` flag for ECDSA signing nonces. We confirmed the vulnerability using a LibreSSL build with debug symbols, checking the inversion code path with a debugger.

BoringSSL. An 03 Nov 2015 commit[6] picked up the `EC_GFp_nistz256_method` implementation for BoringSSL. That commit also included the `BN_mod_exp_mont_consttime` ECDSA nonce inversion callee, which OpenSSL cherry picked. The parent tree[7] is slightly older on the same day. Said tree features constant-time P-256 scalar multiplication with callee `EC_GFp_nistp256_method`, but inverts ECDSA signing nonces with callee `BN_mod_inverse` that fails the same security critical branch, again due to the `BN_FLG_CONSTTIME` flag not being set by the caller—i.e. it follows essentially the same code path as OpenSSL. We verified the vulnerability affects said tree using a debugger.

4 Exploiting the Vulnerability

Our attack setup consists of an Intel Core i5-2400 Sandy Bridge 3.10GHz (32 nm) with 8GB of memory running 64-bit Ubuntu 16.04 LTS "Xenial". Each CPU core has an 8-way 32KB L1 data cache, an 8-way 32KB L1 instruction cache, an 8-way 256KB L2 unified cache, and all the cores share a 12-way 6MB unified LLC (all with 64B cache lines). It does not feature HyperThreading.

We built OpenSSL 1.0.1u with debugging symbols on the executable. Debugging symbols facilitate mapping source code to memory addresses, serving a double purpose to us: (1) Improving our degrading attack (see Section 4.1); (2) Probing the sequence of opera-

[5] https://github.com/libressl-portable/openbsd/commit/85b48e7c232e1dd18292a78a266c95dd317e23d3

[6] https://boringssl.googlesource.com/boringssl/+/18954938684e269ccd59152027d2244040e2b819%5E%21/

[7] https://boringssl.googlesource.com/boringssl/+/27a0d086f7bbf7076270dbeee5e65552eb2eab3a

tions accurately. Note that debugging symbols are not loaded during run time, thus not affecting victim's performance. Attackers can map source code to memory addresses by using reverse engineering techniques [9] if debugging symbols are not available. We set `enable-ec_nistp_64_gcc_128` and `shared` as configuration options at build time to ensure faster execution, constant-time scalar multiplication and compile OpenSSL as a shared object.

As seen in the Figure 2 backtrace, when performing an ECDSA digital signature, OpenSSL calls `ecdsa_sign_setup` to prepare the required parameters and compute most of the actual signature. The random nonce k is created and to avoid possible timing attacks [7] an equivalent fixed bit-length nonce is computed. The length of the equivalent nonce $\hat{k}$ is fixed to one bit more than that of the group's prime order n, thus the equivalent nonce satisfies $\hat{k} = k + \gamma \cdot n$ where $\gamma \in \{1,2\}$.

Additionally, `ecdsa_sign_setup` computes the signature's r using a scalar multiplication function pointer wrapper (i.e. for P-256, traversing the constant-time code path instead of generic wNAF) followed by the modular inverse k^{-1}, needed for the s component of the signature. To compute the inversion, it calls `BN_mod_inverse`, where the `BN_FLG_CONSTTIME` flag is checked but due to the vulnerability discussed in Section 3 the condition fails, therefore proceeding to compute k^{-1} using the classical BEEA.

Note that before executing the BEEA, the equivalent nonce $\hat{k}$ is unpadded through a modular reduction operation, resulting in the original nonce k and voiding the fixed bit-length countermeasure applied shortly before by `ecdsa_sign_setup`.

The goal of our attack is to accurately trace and recover side-channel information leaked from the BEEA execution, allowing us to construct the sequence of right-shift and subtraction operations. To that end, we identify the routines used in the `BN_mod_inverse` method leaking side-channel information.

The `BN_mod_inverse` method operates with very large integers, therefore it uses several specific routines to perform basic operations with BIGNUMs. Addition operations call the routine `BN_uadd`, which is a wrapper for `bn_add_words`—assembly code performing the actual addition. Two different routines are called to perform right-shift operations. The `BN_rshift1` routine performs a single right-shift by one bit position, used on X and Y in their respective loops. The `BN_rshift` routine receives the number of bit positions to shift right as an argument, used on u and v at the end of their respective loops. OpenSSL keeps a counter for the shift count, and the loop conditions test u and v bit values at this offset. This is an optimization allowing u and v to be right-shifted all at once in a single call instead of iteratively. Additionally, subtraction is achieved through the use of the `BN_usub` routine, which is a pure C implementation.

Similar in spirit to previous works [4, 24, 27] that instead target other functionality within OpenSSL, we use the FLUSH+RELOAD technique to attack OpenSSL's BEEA implementation. As mentioned before in Section 2.3, unfortunately the side-channel and the algorithm implementation do not allow us to efficiently probe and distinguish the four critical input-dependent branches, therefore we are limited to knowing only the execution of addition, right-shift and subtraction operations.

After identifying the input-dependent branches in OpenSSL's implementation of the BEEA, using the FLUSH+RELOAD technique we place probes in code routines `BN_rshift1` and `BN_usub`. These two routines provide the best resolution and combination of probes, allowing us to identify the critical input-dependent branches.

The modular inversion is an extremely fast operation and only a small fraction of the entire digital signature. It is challenging to get good resolution and enough granularity with the FLUSH+RELOAD technique due to the speed of the technique itself, therefore, we apply a variation of the performance degradation attack to slow down the modular inversion operation by a factor of ~18. (See Section 4.1.)

Maximizing performance degradation by identifying the best candidate memory lines gives us the granularity required for the attack. Combining the FLUSH+RELOAD technique with a performance degradation attack allows us to determine the number of right-shift operations executed between subtraction calls by the BEEA. From the trace, we reconstruct the sequence of right-shift and subtraction operations (*LS sequence*) executed by the BEEA.

Our attack scenario exploits three CPU cores by running a malicious process in every core and the victim process in the fourth core. The attack consists of a spy process probing the right-shift and subtraction operations running in parallel with the victim application. Additionally, two degrading processes slow down victim's execution, allowing us to capture the LS sequence almost perfectly. Unfortunately there is not always a reliable indicator in the signal for transitions from one right-shift operation to the next, therefore we estimate the number of adjacent right-shift operations by taking into account the latency and the horizontal distance between subtractions. Figure 3 contains sample raw traces captured in our test environment.

Our spy process accurately captures all the subtraction operations but duplicates some right-shift operations, therefore we focus on the first part of the sequence to recover a variable amount of bits of information from every trace. (See Section 4.2.)

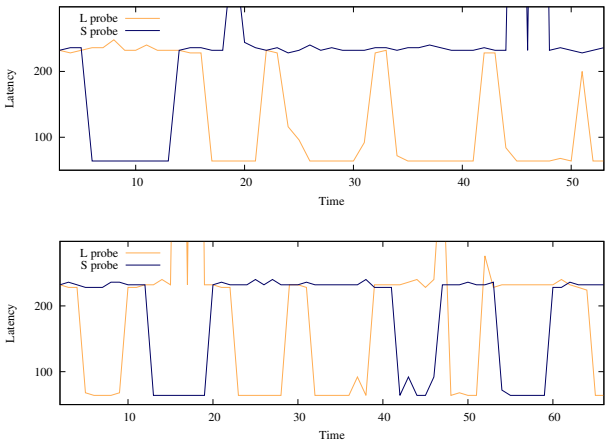

Figure 3: Raw traces for the beginning of two BEEA executions. The L probe tracks right-shift latencies and the S probe tracks subtraction. Latency is in CPU clock cycles. For visualization, focus on the amplitude valleys, i.e. low latency. Top: LS sequence starting SLLLL corresponds to $j = 5$, $\ell_i = 4$, $a_i = 1$. Bottom: LS sequence starting LSLLSLS corresponds to $j = 7$, $\ell_i = 5$, $a_i = 10$. See Section 4.2 for notation.

4.1 Improving Performance Degradation

Performance degradation attacks amplify side-channel signals, improving the quality and the amount of information leaked. Performance degradation attacks have been used previously in conjunction with other side-channel attacks (see e.g. [24]). It can be difficult and time consuming to identify the "hot" memory addresses to degrade that result in the best information leak.

Allan et al. [2] suggest two approaches to find suitable memory lines to degrade. The first approach is to read and understand the victim code in order to identify frequently accessed code sections such as tight loops. This approach requires understanding the code, a task that might not always be possible, takes time and it is prone to errors [26], therefore the authors propose another option.

The second and novel approach they propose is to automate code analysis by collecting code coverage information using the `gcov` tool. The code coverage tool outputs accessed code lines and then using this information it is possible for an attacker to locate the memory lines corresponding to the code lines. Some caveats of this approach are that source lines can be replicated due to compiler optimizations, thus the `gcov` tool might misreport the number of memory accesses. Moreover, code lines containing function calls can be twice as effective compared to the `gcov` output.

In addition to the caveats mentioned previously, we note that the `gcov` profiling tool adds instrumentation to the code. The instrumentation skews the performance of the program, therefore this approach is suboptimal since it requires building the target code twice, one with instrumentation to identify code lines and other only with debugging symbols to measure the real performance.

To that end, we follow a similar but faster and more quantitative approach, potentially more accurate since it leverages additional metrics. Similar to [2] we test the efficiency of the attack for several candidate memory lines. We compare cache-misses between a regular modular inversion and a degraded modular inversion execution, resulting in a list of the "hottest" memory lines, building the code only once with debugging symbols and using hardware register counters.

The `perf` command in Linux offers access to performance counters—CPU hardware registers counting hardware events (e.g. CPU cycles, instructions executed, cache-misses and branch mispredictions). We execute calls to OpenSSL's modular inverse operation, counting the number of cache-misses during a regular execution of the operation. Next, we degrade—by flushing in a loop from the cache—one memory line at a time from the caller `BN_mod_inverse` and callees `BN_rshift1`, `BN_rshift`, `BN_uadd`, `bn_add_words`, `BN_usub`.

The `perf` command output gives us the real count of cache-misses during the regular execution of `BN_mod_inverse`, then under degradation of each candidate memory line. This effectively identifies the "hottest" addresses during a modular inverse operation with respect to both the cache and the actual malicious processes we will use during the attack.

Table 2 summarizes the results over 1,000 iterations of a regular modular inversion execution versus the degradation of different candidate memory lines identified using our technique. The table shows cache-miss rates ranging from ~35% (`BN_rshift` and `BN_usub`) to ~172% (`BN_rshift1`) for one degrading address. Degrading the overall 6 "hottest" addresses accessed by the `BN_mod_inverse` function results in an impressive cache-miss rate of ~1,146%.

Interestingly, the last column of Table 2 reveals the real impact of cache-misses in the execution time of the modular inversion operation. Despite the impressive cache-miss rates, the clock cycle slow down is more modest with a maximum slow down of ~18. These results suggest that in order to get a quality trace, the goal is to achieve an increased rate of cache-misses rather than a CPU clock cycle slow down because whereas the cache-misses suggest a CPU clock cycle slow down, it is not the case for the opposite direction.

The effectiveness of the attack varies for each use case and for each routine called. Some of the routines iterate over internal loops several times (e.g. `BN_rshift1`) whereas in some other routines, iteration over internal loops happens few times (e.g. `BN_usub`) or none at all.

Table 2: `perf` cache-misses and CPU clock cycle statistics over 1,000 iterations for relevant routines called by the `BN_mod_inverse` method.

Target	Cache misses (CM)	Clock cycles (CC)	$\frac{CM}{CM_{BL}}$	$\frac{CC}{CC_{BL}}$
Baseline (BL)	13	211,324	1.0	1.0
`BN_rshift1`	2,396	947,925	172.6	4.4
`BN_usub`	489	364,399	35.2	1.7
`BN_mod_inverse`	956	540,357	68.9	2.5
`BN_uadd`	855	485,088	61.6	2.2
`bn_add_words`	1,124	558,839	81.0	2.6
`BN_rshift`	514	367,929	37.0	1.7
Previous "hot"	10,280	2,576,360	740.5	12.1
Overall "hottest"	15,910	3,817,748	1,146.2	18.0

Take for example previous "hot" addresses from Table 2, by degrading the most used address from each routine does not necessarily gives the best result. Overall "hottest" addresses in Table 2 shows the result of choosing the best strategy for our use case, where the addresses degraded in every routine varies from multiple addresses per routine to no addresses at all.

For our use case, we observe the best results with 6 degrading addresses across two degrading processes executing in different CPU cores. Additional addresses do not provide any additional slow down, instead they impact negatively the FLUSH+RELOAD technique.

4.2 Improving Key Recovery

Arguably the most significant contribution of [8] is they show the LS sequence is sufficient to extract a certain number of LSBs from nonces, even when it is not known whether branch u or v gets taken. They give an algebraic method to recover these LSBs, and utilize these partial nonce bits in a lattice attack, using the formalization in [21, 22]. The disadvantage of that approach is that it fixes the number of known LSBs (denoted ℓ) per equation [8, Sec. 5]: "when a set of signatures are collected such that, for each of them, [ℓ] bits of the nonce are known, a set of equations ... can be obtained and the problem of finding the private key can be reduced to an instance of the [HNP]." Fixing ℓ impacts their results in two important ways. First, since their lattice utilizes a fixed ℓ, they focus on the ability to algebraically recover only a fixed number of bits from the LS sequence. From [8, Tbl. 1], our target implementation is similar to their "Standard-M0" target, and they focus on $\ell \in \{8, 12, 16, 20\}$. For example, to extract $\ell = 8$ LSBs they need to query on average 4 signatures, discarding all remaining signatures that do not satisfy $\ell \geq 8$. Second, this directly influences the number of signatures needed in the lattice phase. From [8, Tbl. 2-3], for 256-bit n and $\ell = 8$, they require 168 signatures. This is because they are discarding three out of four signatures on average where $\ell < 8$, then go on to construct a $d+1$-dimension lattice where $d = 168/4 = 42$ from the signatures that meet the $\ell \geq 8$ restriction. The metric of interest from the attacker perspective is the number of required signatures.

In this section, we improve with respect to both points—extracting a varying number of bits from every nonce, subsequently allowing our lattice problem to utilize every signature queried, resulting in a significantly reduced number of required signatures.

Extracting nonce bits. Rather than focusing on the average number of required signatures as a function of a number of target LSBs, our approach is to examine the average number of bits extracted as a function of LS sequence length. We empirically measured this quantity by generating β_i uniformly at random from $\{1..n-1\}$ for P-256 n, running the BEEA on β_i and n to obtain the ground truth LS sequence, and taking the first j operations from this sequence. We then grouped the β_i by these length-j subsequence values, and finally determined the maximal shared LSBs value of each group. Intuitively, this maps any length-j subsequence to a known LSBs value. For example, a sequence beginning LLS has $j = 3$, $\ell = 3$, $a = 4$ interpreted as a length-3 subsequence that leaks 3 LSBs with a value of 4.

We performed 2^{26} trials (i.e. $1 \leq i \leq 2^{26}$) for each length $1 \leq j \leq 16$ independently and Figure 4 contains the results (see Table 5 in the appendix for the raw data). Naturally as the length of the sequence grows, we are able to extract more bits. But at the same time, in reality for practical side-channels longer sequences are more likely to contain trace errors (i.e. incorrectly inferred LS sequences), ultimately leading to nonsensical lattice problems for key recovery. So we are looking for the right balance between these two factors. Figure 4 allows us to draw several conclusions, including but not limited to: (1) Sequences of length 5 or more allow us to extract a minimum of 3 nonce bits per signature; (2) Similarly length 7 or more for a minimum of 4 nonce bits; (3) The average number of bits extracted grows rapidly at first, then the growth slows as the sequence length increases. This observation pairs nicely with the nature of side-channels: attempting to target longer sequences (risking trace errors) only marginally increases the average number of bits extracted. From the lattice perspective, $\ell \geq 3$ is a practical requirement [21, Sec. 4.2] so in that respect sequences of length 5 is the minimum to guarantee that every signature can be used as an equation for the lattice problem.

To summarize, the data used to produce Figure 4 allows us to essentially build a dictionary that maps LS sequences of a given length to an (ℓ_i, a_i) pair, which we now define and utilize.

Recovering private keys. We follow the formalization of [21, 22] with the use of per-equation ℓ_i due to [4,

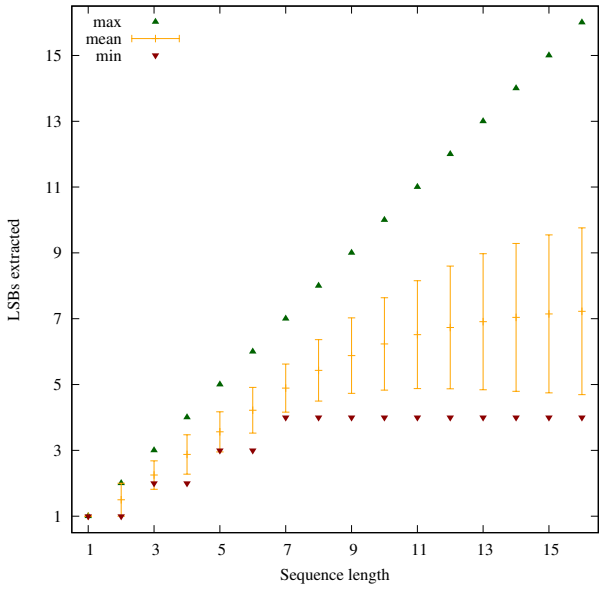

Figure 4: Empirical number of extracted bits for various sequence lengths. Each sequence length consisted of 2^{26} trials, over which we calculated the mean (with deviation), maximum, and minimum number of recovered LSBs. Error bars are one standard deviation on each side. See Table 5 in the appendix for the raw data.

Sec. 4]. Extracted from our side-channel, we are left with equations $k_i = 2^{\ell_i} b_i + a_i$ where ℓ_i and a_i are known, and since $0 < k_i < n$ it follows that $0 \le b_i \le n/2^{\ell_i}$. Denote $\lfloor x \rfloor_n$ modular reduction of x to the interval $\{0..n-1\}$ and $|x|_n$ to the interval $\{-(n-1)/2..(n-1)/2\}$. Define the following (attacker-known) values.

$$t_i = \lfloor r_i/(2^{\ell_i} s_i) \rfloor_n$$
$$\hat{u}_i = \lfloor (a_i - h_i/s_i)/2^{\ell_i} \rfloor_n$$

It now follows that $0 \le \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/2^{\ell_i}$. Setting

$$u_i = \hat{u}_i + n/2^{\ell_i+1}, \text{ we obtain}$$
$$v_i = |\alpha t_i - u_i|_n \le n/2^{\ell_i+1},$$

i.e. integers λ_i exist such that $|v_i - \lambda_i n| \le n/2^{\ell_i+1}$ holds. The u_i approximate αt_i since they are closer than a uniformly random value from $\{1..n-1\}$, leading to an instance of the HNP [5]: recover α given many (t_i, u_i) pairs.

Consider the rational $d+1$-dimension lattice generated by the rows of the following matrix.

$$B = \begin{bmatrix} 2^{\ell_1+1} n & 0 & \dots & \dots & 0 \\ 0 & 2^{\ell_2+1} n & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 2^{\ell_d+1} n & 0 \\ 2^{\ell_1+1} t_1 & \dots & \dots & 2^{\ell_d+1} t_d & 1 \end{bmatrix}$$

Setting

$$\vec{x} = (\lambda_1, \dots, \lambda_d, \alpha)$$
$$\vec{y} = (2^{\ell_1+1} v_1, \dots, 2^{\ell_d+1} v_d, \alpha)$$
$$\vec{u} = (2^{\ell_1+1} u_1, \dots, 2^{\ell_d+1} u_d, 0)$$

establishes the relationship $\vec{x}B - \vec{u} = \vec{y}$. Solving the CVP with inputs B and $\vec{u}$ yields $\vec{x}$ and hence α. We use the embedding strategy [13, Sec. 3.4] to heuristically reduce CVP approximations to Shortest Vector Problem (SVP) approximations. Consider the rational $d+2$-dimension lattice generated by the rows of the following matrix.

$$\hat{B} = \begin{bmatrix} B & 0 \\ \vec{u} & n \end{bmatrix}$$

There is a reasonable chance that lattice-reduced $\hat{B}$ will contain the short lattice basis vector $(\vec{x}, -1)\hat{B} = (\vec{y}, -n)$, revealing α. To extend the search space, we use the randomization technique inspired by Gama et al. [12, Sec. 5], shuffling the order of t_i and u_i and multiplying by a random sparse unimodular matrix between lattice reductions.

Empirical results. Table 3 contains our empirical results for various lattice parameters targeting P-256. As part of our experiments, we were able to successfully reproduce and verify the $\ell \in \{8, 12\}$, $\lg n \approx 256$ lattice results of Cabrera Aldaya et al. [8] in our environment for comparison. While the goal is to minimize the number of required signatures, this should be weighed with observed HNP success probability, affecting search duration. From Figure 4 we focus on LS subsequence lengths $j \in \{5, 7\}$ that yield ℓ_i nonce LSBs from ranges $\{3..5\}$ and $\{4..7\}$, respectively. Again this is in contrast to [8] that fixes ℓ and discards signatures—this is the reason their signature count is much higher than the $d+2$ lattice dimension in their case, but equal in ours.

A relevant metric affecting success probability is the total number of known nonce bits for each HNP instance. Naturally as this sum approaches $\lg n$ one expects correct solutions to start emerging. On the other hand, increasing this sum demands querying more signatures, at the same time increasing d and lattice methods become less precise. For a given HNP instance, denote $l = \sum_{i=1}^{d} \ell_i$, i.e. the total number of known nonce bits over all the equations for the particular HNP instance. Table 3 denotes μ_l the mean value of l over all successful HNP instances—intuitively tracking how many known nonce bits needed in total to reasonably expect success.

We ran 200 independent trials for each set of parameters on a computing cluster with Intel Xeon X5650 nodes. We allowed each trial to execute at most four hours, and we say successful trials are those HNP instances recovering the private key within this allotted

Table 3: P-256 ECDSA lattice attack improvements for BEEA leakage. Empirical values are over 200 trials (4hr max trial duration). Lattice dimension is $d+2$. The number of leaked LSBs per nonce is ℓ. LS subsequence length is j. The average total number of leaked nonce bits per successful HNP instance is μ_l. CPU time is the median.

Source	Signatures	d	ℓ	j	μ_l	Success Rate (%)	CPU Minutes
Prev. [8]	168	42	8	—	336.0	100.0	0.7
Prev. [8]	312	24	12	—	288.0	100.0	0.6
This work	50	50	{4..7}	7	249.7	14.0	79.5
This work	55	55	{4..7}	7	268.8	98.0	1.7
This work	60	60	{4..7}	7	293.4	100.0	0.7
This work	70	70	{3..5}	5	258.2	5.0	130.8
This work	80	80	{3..5}	5	286.1	94.5	13.2
This work	90	90	{3..5}	5	321.2	100.0	4.0

time. Our lattice implementation uses Sage software with BKZ [25] reduction, block size 30.

To summarize, utilizing every signature in our HNP instances leads to a significant improvement over previous work with respect to both the number of required signatures and amount of side-channel data required.

5 Attacking Applications

OpenSSL is a shared library and therefore any vulnerability present in it can potentially be exploited from any application linked against it. This is the case for the present work and to demonstrate the feasibility of our attack in a concrete real-life scenario, we focus on two applications implementing two ubiquitous security protocols: TLS within stunnel and SSH within OpenSSH.

OpenSSL provides ECDSA functionality for both applications and therefore we mount our attack against OpenSSL's ECDSA running within them. More precisely, this section describes the tools and the setup followed to successfully exploit the vulnerability within these applications. In addition, we explain the relevant messages collected for each application, later used for private key recovery together with the trace data and the signatures.

5.1 TLS

Stunnel[8] is a popular portable open source software application that forwards network connections from one port to another and provides a TLS wrapper. Network applications that do not natively support TLS communication benefit from the use of stunnel. More precisely, stunnel can be used to provide a TLS connection between a public port exposing a TLS-enabled network service and a localhost port providing a non-TLS network service. It links against OpenSSL to provide TLS functionality.

For our experiments, we used stunnel 5.39 compiled from stock source and linked against OpenSSL 1.0.1u. We generated a P-256 ECDSA certificate for the stunnel service and chose the `ECDHE-ECDSA-AES128-SHA` TLS 1.2 cipher suite.

In order to collect digital signature and digest tuples, we wrote a custom TLS client that connects to the stunnel service. Our TLS client initiates TLS connections, collects the protocol messages and continues the handshake until it receives the `ServerHelloDone` message, then it drops the connection. The protocol messages contain relevant information for the attack. The `ClientHello` and `ServeHello` messages contain each a 32-byte `random` field, in practice these bytes represent a 4-byte UNIX timestamp concatenated with a 28-byte nonce. The `Certificate` message contains the P-256 ECDSA certificate generated for the stunnel service. The `ServerKeyExchange` message contains ECDH key exchange parameters including the curve type (`named_curve`), the curve name (`secp256r1`) and the `SignatureHashAlgorithm`. Finally, the digital signature itself is sent as part of the `ServerKeyExchange` message. The ECDSA signature is over the concatenated string

```
ClientHello.random + ServerHello.random +
ServerKeyExchange.params
```

and the hash function is SHA-512, proposed by the client in the `ClientHello` message and accepted by the server in the `SignatureHashAlgorithm` field (explicit values `0x06`, `0x03`). Our TLS client saves the hash of the concatenated string and the DER-encoded ECDSA signature sent by the server. Our spy process is launched prior to the TLS handshakes, therefore it collects the trace for each ECDSA signature performed during the handshakes. The process is repeated as needed to build up a set of distinct trace, digital signature, and digest tuples. Section 5.3 shows accuracy results for several LS subsequence patterns for an stunnel victim.

5.2 SSH

OpenSSH[9] is a widely used open source software suite to provide secure communication over an insecure channel. OpenSSH is a set of tools implementing the SSH network protocol and it is typically linked against OpenSSL to perform several cryptographic operations, including digital signatures (excluding `ed25519` signatures) and key exchange.

For our experiments, we used OpenSSH 7.4p1 compiled from stock source and linked against OpenSSL

[8] https://www.stunnel.org

[9] http://www.openssh.com/

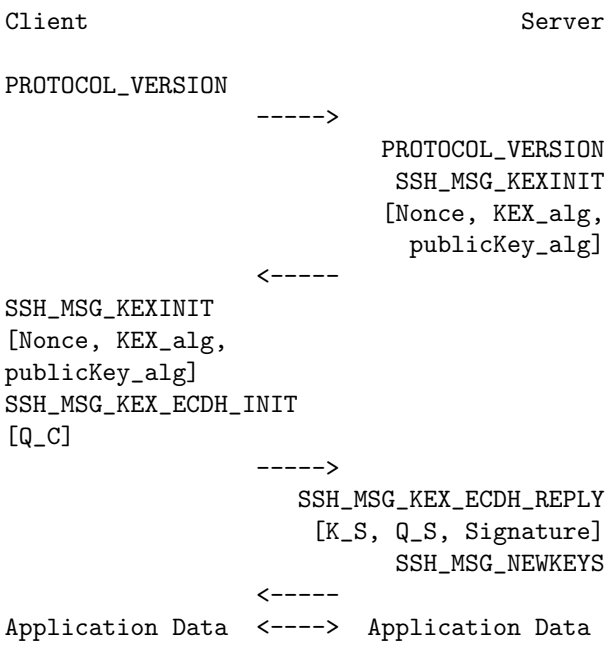

Figure 5: ECC SSH handshake flow with corresponding parameters from all the messages to construct the digest. Our spy process collects timing traces in parallel to the server's ECDSA sign operation, said digital signature being included in a SSH_MSG_KEX_ECDH_REPLY field and collected by our client.

1.0.1u. The ECDSA key pair used by the server and targeted by our attack is the default P-256 key pair generated during installation of OpenSSH.

Following a similar approach to Section 5.1, we wrote a custom SSH client that connects to the OpenSSH server to collect digital signatures and digest tuples. At the same time, our spy process running on the server side collects the timing signals leaked by the server during the handshake.

Relevant to this work, the OpenSSH server was configured with the ecdsa-sha2-nistp256 host key algorithm and the default P-256 key pair. After the initial ClientVersion and ServerVersion messages, the protocol defines the Diffie-Hellman key exchange parameters, the signature algorithm and the hash function identifiers in the SSH_MSG_KEXINIT message. To provide host authentication by the client and the server, a 16-byte random nonce is included in the SSH_MSG_KEXINIT message. The SSH_MSG_KEX_ECDH_REPLY[10] message contains the server's public host key K_S (used to create and verify the signature), server's ECDH ephemeral public key Q_S (used to compute the shared secret K in combination with the client's ECDH ephemeral public key Q_C) and the signature itself. The ECDSA signature is over the hash of the concatenated string

```
ClientVersion + ServerVersion +
Client.SSH_MSG_KEXINIT +
Server.SSH_MSG_KEXINIT +
K_S + Q_C + Q_S + K
```

Our SSH client was configured to use ecdh-sha2-nistp256 and ecdsa-sha2-nistp256 as key exchange and public key algorithms, respectively.

Similar to the previous case, our SSH client saves the hash of the concatenated string and the raw bytes of the ECDSA signature sent by the server. Our spy process is launched prior to the SSH handshakes, therefore it collects the trace for each ECDSA signature performed during the handshakes. All the protocol messages starting from SSH_MSG_NEWKEYS and any client responses are not required by our attack, therefore the client drops the connection and repeats the process as needed to build up a set of distinct trace, digital signature, and digest tuples. Section 5.3 shows accuracy results for several LS subsequence patterns for an SSH server victim.

5.3 Attack Results

Procurement accuracy. Table 4 shows the empirical accuracy results for patterns of length $j = 5$ at the beginning of the LS sequence in the context of OpenSSL ECDSA executing in real world applications (TLS via stunnel, SSH via OpenSSH). From our empirical results we note two trends: (1) Similar to previous works [4, 24, 27], the accuracy of the subsequence decreases as ℓ increases due to the deviation in the right-shift operation width. (2) SSH traces are slightly noisier than TLS traces; we speculate this is due to the computation of the ECDH shared secret prior to the ECDSA signature itself. Using our improved degradation technique (Section 4.1) we can recover a with very high probability, despite the speed of the modular inversion operation and the imperfect traces. See Table 6 in the appendix for analogous $j = 7$ results.

Key recovery. We close with a few data points for our end-to-end attack, here focusing on TLS. In this context, end-to-end means all steps from the attacker perspective—i.e. launching the degrade processes, launching the spy process, and launching our custom TLS client. Finally, repeating these steps to gather multiple trace and signature pairs, then running the lattice attack for key recovery. That is, no steps in the attack chain are abstracted away.

The experiments for Table 3 assume perfect traces. However, as seen in Table 4 and Table 6, while we observe quite high accuracy, in our environment we are unable to realize absolutely perfect traces. Trace errors will occur, and lattice methods have no recourse to compen-

[10] https://tools.ietf.org/html/rfc5656

Table 4: Accuracy for length $j = 5$ subsequences over 15,000 TLS/SSH handshakes.

			TLS	SSH
Pattern	ℓ_i	a_i	Accuracy (%)	Accuracy (%)
LLLLL	5	0	77.9	73.3
SLLLL	4	1	99.8	98.0
LSLLL	4	2	99.3	98.9
SLSLL	3	3	98.9	97.2
LLSLL	4	4	98.0	96.7
SLLSL	3	5	95.8	95.5
LSLSL	3	6	85.5	97.2
SLSLS	3	7	99.2	97.8
LLLSL	4	8	93.3	92.5
SLLLS	4	9	94.4	94.6
LSLLS	4	10	81.1	93.5
LLSLS	4	12	96.4	96.7
LLLLS	5	16	89.8	85.0

sate for them. We resort to oversampling and randomized brute force search to achieve key recovery in practice.

For the $j = 5$ case, we procured 150 signatures with (potentially imperfect) trace data. Consulting Table 3, we took 400 random subsets of size 80 from this set and ran lattice attack instances on a computing cluster. The first instance to succeed in recovering the private key did so in roughly 8 minutes. Checking the ground truth afterwards, 142 of these original 150 traces were correct, i.e. ~0.18% of all possible subsets are error-free. This successful attack is consistent with the probability $1-(1-0.0018)^{400} \approx 51.4\%$.

Similarly for the $j = 7$ case, we procured 150 signatures with (potentially imperfect) trace data. Consulting Table 3, we took 400 random subsets of size 55 from this set and ran lattice attack instances on a computing cluster. The first instance to succeed in recovering the private key did so in under a minute. Checking the ground truth afterwards, 137 of these original 150 traces were correct, i.e. ~0.19% of all possible subsets are error-free. This successful attack is also consistent with the probability $1-(1-0.0019)^{400} \approx 53.3\%$.

It is worth noting that with this naïve strategy, it is always possible to trade signatures for more offline search effort. Moreover, it is possible to traverse the search space by weighting trace data subsets according to known pattern accuracy, e.g. explore patterns with accuracy $\geq 95\%$ sooner.

6 Conclusion

In this work, we disclose a new vulnerability in widely-deployed software libraries that causes ECDSA nonce inversions to be computed with the BEEA instead of a code path with microarchitecture attack mitigations. We design and demonstrate a practical cache-timing attack against this insecure code path, leveraging our new performance degradation metric. Combined with our improved nonce bits recovery approach and lattice parameterization, this enable us to recover P-256 ECDSA private keys from OpenSSL despite constant-time scalar multiplication. As far as we are aware, this is the first cache-timing attack targeting nonce inversion in OpenSSL, and furthermore the first side-channel attack against cryptosystems leveraging its constant-time P-256 scalar multiplication methods. Our contributions traverse both practice and theory, recovering keys with as few as 50 signatures and corresponding traces.

Stepping back from the concrete side-channel attack we realized here, our improved nonce bit recovery approach coupled with tuned lattice parameters demonstrates that even small leaks of BEEA execution can have disastrous consequences. Observing as few as the first 5 operations in the LS sequence allows every signature to be used as an equation for the lattice problem. Moreover, our work highlights the fact that constant-time considerations are ultimately about the software stack, and not necessarily a single component in isolation.

The rapid development of cache-timing attacks paired with the need for fast solutions and mitigations led to the inclusion of the `BN_FLG_CONSTTIME` flag in OpenSSL. Over the years, the flag proved to be useful when introducing new constant-time implementations, but unfortunately its usage is now beyond OpenSSL's original design. As new cache-timing attacks emerged, the usage of the flag increased throughout the library. At the same time the programming error probability increased, and many of those errors permeated to forks such as LibreSSL and BoringSSL. The recent exploitation surrounding the flag's usage, this work included, highlights it as a prime example of why failing securely is a fundamental concept in security by design. For example, P-256 takes the constant-time scalar multiplication code path by default, oblivious to the flag, while in stark contrast modular inversion relies critically on this flag being set to follow the code path with microarchitecture attack mitigations.

Following responsible disclosure procedures, we reported the issue to the developers of the affected products after our findings. We lifted the embargo in December 2016. Despite OpenSSL's 1.0.1 branch being a standard package shipped with popular Linux distributions such as Ubuntu (12.04 LTS and 14.04 LTS), Debian (7.0 and 8.0), and SUSE, it reached EOL in January 2017. Backporting security fixes to EOL packages is a necessary and challenging task, and to contribute we provide a patch to mitigate our attack. OpenSSL assigned CVE-2016-7056 based on our work. See the appendix for the patch.

Acknowledgments

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

This research was supported in part by COST Action IC1306.

References

[1] ACIIÇMEZ, O., GUERON, S., AND SEIFERT, J. 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.

[2] ALLAN, T., BRUMLEY, B. B., FALKNER, K. E., VAN DE POL, J., AND YAROM, Y. 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*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 422–435.

[3] ARAVAMUTHAN, S. AND THUMPARTHY, V. R. 2007. A parallelization of ECDSA resistant to simple power analysis attacks. In *2007 2nd International Conference on Communication Systems Software and Middleware*. 1–7.

[4] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. 2014. "Ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, L. Batina and M. Robshaw, Eds. Lecture Notes in Computer Science, vol. 8731. Springer, 75–92.

[5] BONEH, D. AND VENKATESAN, R. 1996. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 129–142.

[6] BRUMLEY, B. B. AND HAKALA, R. M. 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*, M. Matsui, Ed. Lecture Notes in Computer Science, vol. 5912. Springer, 667–684.

[7] BRUMLEY, B. B. AND TUVERI, N. 2011. Remote timing attacks are still practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*. 355–371.

[8] CABRERA ALDAYA, A., CABRERA SARMIENTO, A. J., AND SÁNCHEZ-SOLANO, S. 2016. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic Engineering*.

[9] CIPRESSO, T. AND STAMP, M. 2010. Software reverse engineering. In *Handbook of Information and Communication Security*. 659–696.

[10] FAN, S., WANG, W., AND CHENG, Q. 2016. Attacking OpenSSL implementation of ECDSA with a few signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1505–1515.

[11] GALLANT, R. P., LAMBERT, R. J., AND VANSTONE, S. A. 2001. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, J. Kilian, Ed. Lecture Notes in Computer Science, vol. 2139. Springer, 190–200.

[12] GAMA, N., NGUYEN, P. Q., AND REGEV, O. 2010. Lattice enumeration using extreme pruning. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, H. Gilbert, Ed. Lecture Notes in Computer Science, vol. 6110. Springer, 257–278.

[13] GOLDREICH, O., GOLDWASSER, S., AND HALEVI, S. 1997. Public-key cryptosystems from lattice reduction problems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, B. S. K. Jr., Ed. Lecture Notes in Computer Science, vol. 1294. Springer, 112–131.

[14] GUERON, S. AND KRASNOV, V. 2015. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering 5,* 2, 141–151.

[15] HOWGRAVE-GRAHAM, N. AND SMART, N. P. 2001. Lattice attacks on digital signature schemes. *Des. Codes Cryptography 23,* 3, 283–290.

[16] KÄSPER, E. 2011. Fast elliptic curve cryptography in OpenSSL. In *Financial Cryptography and Data Security - FC 2011 Workshops, RLCPS and WECSR 2011, Rodney Bay, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers*, G. Danezis, S. Dietrich, and K. Sako, Eds. Lecture Notes in Computer Science, vol. 7126. Springer, 27–39.

[17] KOBLITZ, N. 1987. Elliptic curve cryptosystems. *Mathematics of Computation 48,* 177, 203–209.

[18] MENEZES, A., VAN OORSCHOT, P. C., AND VANSTONE, S. A. 1996. *Handbook of Applied Cryptography*. CRC Press.

[19] MEYER, C. AND SCHWENK, J. 2013. SoK: Lessons learned from SSL/TLS attacks. In *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, Y. Kim, H. Lee, and A. Perrig, Eds. Lecture Notes in Computer Science, vol. 8267. Springer, 189–209.

[20] MILLER, V. S. 1985. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*. 417–426.

[21] NGUYEN, P. Q. AND SHPARLINSKI, I. E. 2002. The insecurity of the Digital Signature Algorithm with partially known nonces. *J. Cryptology 15,* 3, 151–176.

[22] NGUYEN, P. Q. AND SHPARLINSKI, I. E. 2003. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Des. Codes Cryptography 30,* 2, 201–217.

[23] PERCIVAL, C. 2005. Cache missing for fun and profit. In *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*.

[24] PEREIDA GARCÍA, C., BRUMLEY, B. B., AND YAROM, Y. 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*. 1639–1650.

[25] SCHNORR, C. AND EUCHNER, M. 1994. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program. 66*, 181–199.

[26] SINHA, V., KARGER, D. R., AND MILLER, R. 2006. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), 4-8 September 2006, Brighton, UK*. 187–194.

[27] VAN DE POL, J., SMART, N. P., AND YAROM, Y. 2015. Just a little bit more. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*, K. Nyberg, Ed. Lecture Notes in Computer Science, vol. 9048. Springer, 3–21.

[28] YAROM, Y. AND FALKNER, K. 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.*, K. Fu and J. Jung, Eds. USENIX Association, 719–732.

[29] YAROM, Y., GENKIN, D., AND HENINGER, N. 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*, B. Gierlichs and A. Y. Poschmann, Eds. Lecture Notes in Computer Science, vol. 9813. Springer, 346–367.

A Mitigation

Below is the fix for CVE-2016-7056 in uuencode format.


```
begin-base64 664 fix_CVE-2016-7056.patch
RnJvbSAyNDliY2YzMTQwNWUxNjIyZDA1ZWY2MGRjNWU3M2M1NGVmYTY0ZjNj
IE1vbiBTZXAgMTcgMDA6MDA6MDAgMjAwMQpGcm9tOiA9P1VURi04P3E/Q2Vz
YXI9MjBQZXJlaWRhPTIwR2FyYz1DMz1BRGE/PSA8Y2VzYXIucGVyZWlkYWdh
cmNpYUB0dXQuZmk+CkRhdGU6IEZyaSwgMTYgRGVjIDIwMTYgMTI6MDI6MTkg
KzAyMDAKU3ViamVjdDogW1BBVENIXSBFQ0RTQSB2dWxuZXJhYmxlIHRvIGNh
Y2hlLXRpbWluZyBhdHRhY2suIEJOX21vZF9pbnZlcnNlIGZhaWxzCiB0byB0
YWtlIGNvbnN0YW50LXRpbWUgcGF0aCwgdGh1cyBsZWFraW5nIG5vbmNlJ3Mg
aW5mb3JtYXRpb24uCgotLS0KIGNyeXB0by9lY2RzYS9lY3Nfb3NzbC5jIHwg
MiArKwogMSBmaWxlIGNoYW5nZWQsIDIgaW5zZXJ0aW9ucygrKQoKZGlmZiAt
LWdpdCBhL2NyeXB0by9lY2RzYS9lY3Nfb3NzbC5jIGIvY3J5cHRvL2VjZHNh
L2Vjc19vc3NsLmMKaW5kZXggNGM1ZmE2Yi4uNzJlN2MwNSAxMDA2NDQKLS0t
IGEvY3J5cHRvL2VjZHNhL2Vjc19vc3NsLmMKKysrIGIvY3J5cHRvL2VjZHNh
L2Vjc19vc3NsLmMKQEAgLTE0Nyw2ICsxNDcsOCBAQCBzdGF0aWMgaW50IGVj
ZHNhX3NpZ25fc2V0dXAoRUNfS0VZICplY2tleSwgQk5fQ1RYICpjdHhfaW4s
IEJJR05VTSAqKmtpbnZwLAogICAgICAgICAgICAgICAgaWYgKCFCTl9hZGQoaywg
aywgb3JkZXIpKQogICAgICAgICAgICAgICAgIGdvdG8gZXJyOwogCisgICAg
ICAgIEJOX3NldF9mbGFncyhrLCBCTl9GTEdfQ09OU1RUSU1FKTsKKwogICAg
ICAgICAvKiBjb21wdXRlIHIgdGhlIHgtY29vcmRpbmF0ZSBvZiBnZW5lcmF0
b3IgKiBrICovCiAgICAgICAgIGlmICghRUNfUE9JTlRfbXVsKGdyb3VwLCB0
bXBfcG9pbnQsIGssIE5VTEwsIE5VTEwsIGN0eCkpIHsKICAgICAgICAgICAg
IEVDRFNBZXJyKEVDRFNBX0ZfRUNEU0FfU0lHTl9TRVRVUCwgRVJSX1JfRUNf
TElCKTsKLS0gCjIuNy40Cgo=
====
```


B Supplementary Empirical Data

Table 5 contains the raw data used to produce Figure 4. Table 6 with $j = 7$ is analogous to Table 4 with $j = 5$.

Table 5: Empirical number of extracted bits for various sequence lengths. Each sequence length consisted of 2^{26} trials, over which we calculated the mean (with deviation), maximum, and minimum number of recovered LSBs. See Figure 4 for an illustration.

j	Mean	St. Dev.	Min	Max
1	1.00	0.00	1	1
2	1.50	0.50	1	2
3	2.25	0.43	2	3
4	2.87	0.60	2	4
5	3.56	0.61	3	5
6	4.22	0.70	3	6
7	4.89	0.73	4	7
8	5.43	0.93	4	8
9	5.88	1.15	4	9
10	6.23	1.40	4	10
11	6.52	1.64	4	11
12	6.73	1.87	4	12
13	6.91	2.07	4	13
14	7.04	2.24	4	14
15	7.15	2.40	4	15
16	7.23	2.53	4	16

Table 6: Accuracy for length $j = 7$ subsequences over 15,000 TLS/SSH handshakes.

			TLS	SSH
Pattern	ℓ_i	a_i	Accuracy (%)	Accuracy (%)
LLLLLLL	7	0	43.8	30.1
SLLLLSL	5	1	93.4	93.1
LSLLLLS	6	2	82.6	88.0
SLSLLSL	4	3	94.8	93.4
LLSLLLL	6	4	92.9	86.4
SLLSLSL	4	5	95.2	94.1
LSLSLLS	5	6	79.2	92.3
SLSLSLL	4	7	98.8	96.6
LLLSLLL	6	8	84.8	80.5
SLLLSLL	5	9	80.0	81.1
LSLLSLS	5	10	80.8	90.9
SLSLLLS	5	11	91.7	85.4
LLSLSLL	5	12	94.3	94.5
SLLSLLS	5	13	90.9	90.6
LSLSLSL	4	14	83.5	95.1
SLSLSLS	4	15	97.8	97.1
LLLLSLL	6	16	87.7	83.8
SLLLLLL	6	17	92.0	92.4
LSLLLSL	5	18	81.8	90.7
LLSLLSL	5	20	94.3	94.7
LSLSLLL	5	22	80.0	91.5
LLLSLSL	5	24	94.4	91.1
SLLLSLS	5	25	94.3	94.3
LSLLSLL	5	26	74.7	86.1
SLSLLLL	5	27	92.9	89.7
LLSLSLS	5	28	94.6	93.6
SLLSLLL	5	29	85.4	84.8
LLLLLSL	6	32	65.7	61.1
LSLLLLL	6	34	91.5	91.5
LLSLLLS	6	36	93.0	89.3
LLLSLLS	6	40	89.0	88.5
LLLLSLS	6	48	87.2	82.7
SLLLLLS	6	49	86.8	85.5
LLLLLLS	7	64	25.6	33.0