“Make Sure DSA Signing Exponentiations Really are Constant-Time”

Cesar Pereida García
Department of Computer Science
Aalto University, Finland
cesar.pereida@aalto.fi

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

Yuval Yarom
The University of Adelaide and Data61, CSIRO, Australia
yval@cs.adelaide.edu.au

ABSTRACT

TLS and SSH are two of the most commonly used protocols for securing Internet traffic. Many of the implementations of these protocols rely on the cryptographic primitives provided in the OpenSSL library. In this work we disclose a vulnerability in OpenSSL, affecting all versions and forks (e.g. LibreSSL and BoringSSL) since roughly October 2005, which renders the implementation of the DSA signature scheme vulnerable to cache-based side-channel attacks. Exploiting the software defect, we demonstrate the first published cache-based key-recovery attack on these protocols: 260 SSH-2 handshakes to extract a 1024/160-bit DSA host key from an OpenSSH server, and 580 TLS 1.2 handshakes to extract a 2048/256-bit DSA key from an stunnel server.

Keywords

applied cryptography; digital signatures; side-channel analysis; timing attacks; cache-timing attacks; DSA; OpenSSL; CVE-2016-2178

1. INTRODUCTION

One of the contributing factors to the explosion of the Internet in the last decade is the security provided by the underlying cryptographic protocols. Two of those protocols are the Transport Layer Security (TLS) protocol, which provides security to network communication and the more specialized Secure Shell (SSH), which provides secure login to remote hosts.

Software implementations of these protocols often use the cryptographic primitives' implementations of the OpenSSL cryptographic library. Consequently, the security of these implementations depends on the security of OpenSSL.

In this paper we present a novel side-channel cache-timing attack against OpenSSL's DSA implementation. The attack exploits a vulnerability in OpenSSL, which fails to use a side-channel-secure implementation of modular exponentiation — the core mathematical operation used in DSA signatures.

Our attack builds upon several techniques to profile the cache memory and capture timing signals. The signals are processed and converted into a sequence of square and multiplication (SM) operations from which we extract information to create a lattice problem. The solution to the lattice problem yields the secret key of digital signatures.

Flush+Reload [40] is a powerful technique to perform cache-timing attacks. We adapt the Flush+Reload technique to OpenSSL's implementation of DSA and, exploiting properties of the Intel implementation of the x86 and x64 processor architectures, our spy program probes relevant memory addresses to create a signal trace.

We process the captured signal to get the SM sequence performed by the *sliding window exponentiation* (SWE) algorithm. Then we observe and analyze the number of bits that can be extracted and used from each of those sequences. Later, the variable amount of bits extracted from each trace is used as input to a lattice attack that recovers the private key.

To bridge the gap between the limited resolution of the Flush+Reload technique [4] and the high-performance of the OpenSSL code we apply the performance-degradation technique of Allan et al. [4]. This technique slows the exponentiation by an average factor of 20, giving a high resolution trace and allowing us to extract up to 8 bits of information from some of the traces.

Similar to previous works [9, 14, 21, 32], we perform a lattice attack to recover the secret key. We use the lattice construction of Benger et al. [9] and solve the resulting lattice problem using the lattice reduction technique of Nguyen and Shparlinski [28].

A unique feature of our work is that we target common cryptographic protocols. Previous works that demonstrate cache-timing key-recovery attacks only target the cryptographic primitives, ignoring potential cache noise from the protocol implementation. In contrast, we present end-to-end attacks on two common cryptographic protocols: SSH and TLS. We are, therefore, the first to demonstrate that cache-timing attacks are a threat not only when executing the cryptographic primitives but also in the presence of the cache activity of the whole protocol suite.

Our contributions in this work are the following:

- We identify a security weakness in OpenSSL which fails to use a side-channel safe implementation when performing DSA signatures. (Section 3)
- We describe how to use a combination of the Flush+

CCS'16 October 24-28, 2016, Vienna, Austria

ACM ISBN 978-1-4503-4139-4/16/10.
DOI: http://dx.doi.org/10.1145/2976749.2978420

Reload technique with a performance-degradation attack to leak information from the unsafe SWE algorithm. (Section 4)

- We present the first key-recovery cache-timing attack on the TLS and SSH cryptographic protocols. (Section 5)
- We construct and solve a lattice problem with the side-channel information and the digital signatures in order to recover the secret key. (Section 6)

2. BACKGROUND

2.1 Memory Hierarchy

Accessing data and instructions from main memory is a time consuming operation which delays the work of the fast processors, for that reason the memory hierarchy includes smaller and faster memories called *caches*. Caches improve the performance by exploiting the spatial and temporal locality of the memory access.

In modern processors the hierarchy of caches is structured as follows, higher-level caches, located closer to the processor core, are smaller and faster than low-level caches, which are located closer to main memory. Recent Intel architecture typically has three levels of cache: L1, L2 and Last-Level Cache (LLC).

Each core has two L1 caches, a data cache and an instruction cache, each 32 KiB in size with an access time of 4 cycles. L2 caches are also core-private and have an intermediate size (256 KiB) and latency (7 cycles). The LLC is shared among all of the cores and is a unified cache, containing both data and instructions. Typical LLC sizes are in megabytes and access time is in the order of 40 cycles.

The unit of memory and allocation in a cache is called *cache line*. Cache lines are of a fixed size B, which is typically 64 bytes. The $\lg(B)$ low-order bits of the address, called *line offset*, are used to locate the datum in the cache line.

When a memory address is accessed, the processor checks the availability of the address line in the top-level L1 cache. If the data is there then it is served to the processor, a situation referred to as a *cache hit*. In a *cache miss*, when the data is not found in the L1 cache, the processor repeats the search for the line in the next cache level and continues through all the caches. Once the line is found, the processor stores a copy in the cache for future use.

Most caches are *set-associative*. They are composed of S *cache sets* each containing a fixed number of cache lines. The number of cache lines in a set is the cache *associativity*, i.e., a cache with W lines in each set is a W-way *set-associative* cache.

Since the main memory is orders of magnitude larger than the cache, more than W memory lines may map to the same cache set. If a cache miss occurs and all the cache lines in the matching cache set are in use, one of the cached lines is evicted, freeing a slot for a new line to be fetched from a lower-level memory. Several *cache replacement policies* exist to determine the cache line to evict when a cache miss occurs but the typical policy in use is an approximation to the least-recently-used (LRU).

The last-level cache in modern Intel processors is *inclusive*. Inclusive caches contain a superset of the contents of the cache levels above them. In the case of Intel processors, the contents of the L1 and L2 caches is also stored in the last-level cache. A consequence of the inclusion property is that when data is evicted from the last-level cache it is also evicted from all of the other levels of cache in the processor.

Intel architecture implements several cache optimizations. The spatial pre-fetcher pairs cache lines and attempt to fetch the pair of a missed line [17]. Consecutive accesses to memory addresses are detected and pre-fetched when the processor anticipates they may be required [17]. Additionally, when the processor is presented with a conditional branch, *speculative execution* brings the data of both branches into the cache before the branch condition is evaluated [35].

Page [30] noted that tracing the sequence of cache hits and misses of software may leak information on the internal working of the software, including information that may lead to recovering cryptographic keys.

This idea was later extended and used for mounting several cache-based side-channel attacks [10, 29, 31]. Other attacks were shown against the L1-instruction cache [3], the branch prediction buffer [1, 2] and the last-level cache [20, 22, 25, 40].

2.2 The Flush+Reload Attack

Our LLC-based attack is based on the Flush+Reload [20, 40] attack, which is a cache-based side-channel attack technique.

Unlike the earlier Prime+Probe technique [29, 31] that detects activity in cache sets, the Flush+Reload technique identifies access to memory lines, giving it a higher resolution, a high accuracy and high signal-to-noise ratio.

Like Prime+Probe, Flush+Reload relies on cache sharing between processes. Additionally, it requires data sharing, which is typically achieved through the use of shared libraries or using page de-duplication [6, 36].

A round of the attack, which identifies victim access to a shared memory line, consists of three phases. (See Algorithm 1.) In the first phase the adversary evicts the monitored memory line from the cache. In the second phase, the adversary waits a period of time so the victim has an opportunity to access the memory line. In the third phase, the adversary measures the time it takes to reload the memory line. If during the second phase the victim accesses the memory line, the line will be available in the cache and the reload operation in the third phase will take a short time. If, on the other hand, the victim does not access the memory line then the third phase takes a longer time as the memory line is loaded from main memory.

Algorithm 1: Flush+Reload Attack

```
Input: Memory Address addr.
Result: True if the victim accessed the address.
begin
    flush(addr)
    Wait for the victim.
    time ← current_time()
    tmp ← read(addr)
    readTime ← current_time() - time
    return readTime < threshold
```

The execution of the victim and the adversary processes are independent of each other, thus synchronization of probing is important and several factors need to be considered when processing the side-channel data. Some of those factors are the waiting period for the adversary between probes, memory lines to be probed, size of the side-channel trace and *cache-hit* threshold. One important goal for this attack is to achieve the best resolution possible while keeping the error rate low and one of the ways to achieve this is by targeting memory lines that occur frequently during execution, such as loop bodies. Several processor optimizations are in place during a typical process execution and an attacker must be aware of these optimizations to filter them during the analysis of the attack results. See [4, 39, 40] for discussions of some of these parameters.

A typical implementation of the FLUSH+RELOAD attack makes use of the `clflush` instruction of the x86 and x64 instruction sets. The `clflush` instruction evicts a specific memory line from *all* the cache hierarchy and being an unprivileged instruction, it can be used by any process.

The inclusiveness of the LLC is essential for the FLUSH+RELOAD attack. Whenever a memory line is evicted from the LLC, the processor also evicts the line from all of the L1 and L2 caches. On processors that do not have an inclusive LLC, e.g., AMD processors, the attack does not work [40]. See, however, Lipp et al. [24] for a variant of the technique that does not require an inclusive LLC.

2.3 The Digital Signature Algorithm (DSA)

A variant of the ElGamal signature scheme, DSA was first proposed by the U.S. National Institute of Standards and Technology (NIST). DSA uses the multiplicative group of a finite field. We use the following notation for DSA.

Parameters: Primes p, q such that q divides $(p-1)$, a generator g of multiplicative order q in $GF(p)$ 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 such that $0 < \alpha < q$ and the corresponding public key y is given by $y = g^{\alpha} \bmod p$. Calculating the private key given the public key requires solving the 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—the message m is not necessarily encrypted. Using her private-public key pair (α_A, y_A), Alice performs the following steps:

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

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

1. Reject the signature if it does not satisfy $0 < r < q$ and $0 < s < q$.
2. Compute $w = s^{-1} \bmod q$ and $h(m)$.
3. Compute $u_1 = h(m)w \bmod q$ and $u_2 = rw \bmod q$.
4. Compute $v = (g^{u_1} y_A^{u_2} \bmod p) \bmod q$.
5. Accept the signature if and only if $v = r$ holds.

2.3.1 DSA in Practice

Putting it mildly, there is no consensus on key sizes, and furthermore keys seen in the wild and used in ubiquitous protocols have varying sizes—sometimes dictated by existing and deployed standards. For example, NIST defines 1024-bit p with 160-bit q as "legacy-use" and 2048-bit p with 256-bit q as "acceptable" [8]. We focus on these two parameter sets.

SSH's Transport Layer Protocol[1] lists DSA key type `ssh-dss` as "required" and defines r and s as 160-bit integers, implying 160-bit q. In fact the OpenSSH tool `ssh-keygen` defaults to 160-bit q and 1024-bit p for these key types, not allowing the user to override that option, and using the same parameters to generate the server's host key. It is worth noting that recently as of version 7.0, OpenSSH disables host server DSA keys by a configurable default option[2], but of course this does not affect already deployed solutions.

As a countermeasure to previous timing attacks, OpenSSL's DSA implementation pads nonces by adding either q or $2q$ to k—see details in Section 3.

For the DSA signing algorithm, Step 2 is the performance bottleneck and the exponentiation algorithm used will prove to be of extreme importance when we later collect our side-channel information in Section 4.

2.4 Sliding Window Exponentiation

Sliding window exponentiation (SWE) is a widely implemented software method to perform integer exponentiations, e.g. featured alongside other methods in the OpenSSL codebase. SWE is fairly popular due to its performance since it reduces the amount of pre-computation needed and, moreover, reduces the average amount of multiplications performed during the exponentiation.

An exponent e is represented and processed as a sequence of windows e_i, each of length $L(e_i)$ bits. Processing the exponent in windows reduces the amount of multiplications at the cost of increased memory utilization since a table of pre-computed values is used.

A window e_i can be a zero window represented as a string of "0"s or non-zero window represented as a string starting and ending with "1"s and such window is of width w (determined in OpenSSL by the size in bits of the exponent e). The length of non-zero windows satisfy $1 \leq L(e_i) \leq w$, thus the value of any given non-zero window is an odd number between 1 and $2^w - 1$.

As mentioned before, the algorithm pre-computes values and stores them in a table to be used later during multiplication operations. The multipliers computed are $b^v \bmod m$ for each odd value of v where $1 \leq v \leq 2^w - 1$ and these values are stored in table index $g[i]$ where $i = (v-1)/2$. For example, with the standard 160-bit q size, OpenSSL uses a window width $w = 4$, the algorithm pre-computes multipliers $b^1, b^3, b^5, ..., b^{15} \bmod m$ and stores them in $g[0], g[1], g[2]$, ..., $g[7]$, respectively.

Using the SWE representation of the exponent e, Algorithm 2 computes the corresponding exponentiation through a combination of squares and multiplications in a left-to-right approach. The algorithm scans every window e_i from

[1] https://tools.ietf.org/html/rfc4253
[2] http://www.openssh.com/legacy.html

Algorithm 2: Sliding window exponentiation.

```
Input: Window size $w$, base $b$, modulo $m$, N-bit
       exponent $e$ represented as $n$ windows $e_i$, each of
       length $L(e_i)$.
Output: $b^e$ mod $m$.
// Pre-computation
$g[0] \leftarrow b$ mod $m$;
$s \leftarrow$ MULT($g[0], g[0]$) mod $m$;
for $j \leftarrow 1$ to $2^{w-1} - 1$ do
    $g[j] \leftarrow$MULT($g[j-1], s$) mod $m$;
// Exponentiation
$r \leftarrow 1$;
for $i \leftarrow n$ to 1 do
    for $j \leftarrow 1$ to $L(e_i)$ do
        $r \leftarrow$MULT($r, r$) mod $m$;
    if $e_i \neq 0$ then $r \leftarrow$MULT($r, g[(e_i - 1)/2]$) mod $m$;
return $r$;
```

the most significant bit (MSB) to the least significant bit (LSB).

For any window, a square operation is executed for each bit and additionally for a non-zero window, the algorithm executes an extra multiplication when it reaches the LSB of the window.

For novel reasons explained later in Section 3, the side-channel part of our attack focuses on this algorithm. Specifically, in getting the sequence of squares and multiplies performed during its execution. Then we extract partial information from the sequence for later use in the lattice attack.

2.5 Partial key disclosure

Recall that the nonce k and the secret key α satisfy the following linear congruence.

$$s = k^{-1}(h(m) + \alpha r) \bmod q$$

The constants of the linear combination are specified by s, $h(m)$, and r, which, typically for a signed message, are all public. Hence, knowing the nonce k reveals the secret key α.

Typically, side-channel leakage from SWE only recovers partial information about the nonce. The adversary, therefore, has to use that partial information to recover the key. The usual technique for recovering the secret key from the partial information is to express the problem as a *hidden number problem* [12] which is solved using a lattice technique.

2.5.1 *The hidden number problem*

In the hidden number problem (HNP) the task is to find a hidden number given some of the MSBs of several modular linear combinations of the hidden number. More specifically, the problem is to find a secret number α given a number of triples (t_i, u_i, ℓ_i) such that for $v_i = |\alpha \cdot t_i - u_i|_q$ we have $|v_i| \leq q/2^{\ell_i+1}$, where $|\cdot|_q$ is the reduction modulo q into the range $(-q/2, \ldots, q/2)$.

Boneh and Venkatesan [12] initially investigate HNP with a constant $\ell_i = \ell$. They show that for $\ell < \log^{1/2} q + \log\log q$ and random t_i, the hidden number α can be recovered given a number of triples linear in $\log q$.

Howgrave-Graham and Smart [21] extend the work of Boneh and Venkatesan [12] showing how to construct an HNP instance from leaked LSBs and MSBs of DSA nonces. Nguyen and Shparlinski [27] prove that for a good enough hash function and for a linear number of randomly chosen nonces, knowing the ℓ LSBs of a certain number of nonces, the $\ell + 1$ MSBs or $2 \cdot \ell$ consecutive bits anywhere in the nonces is enough for recovering the long term key α. They further demonstrate that a DSA-160 key can be broken if only the 3 LSBs of a certain number of nonces are known. Nguyen and Shparlinski [28] extend the results to ECDSA, and Liu and Nguyen [26] demonstrate that only 2 LSBs are required for breaking a DSA-160 key. Benger et al. [9] extend the technique to use a different number of leaked LSBs for each signature.

2.5.2 *Lattice attack*

To find the hidden number from the triples we solve a lattice problem. The construction of the lattice problem presented here is due to Benger et al. [9], and is based on the constructions in earlier publications [12, 27].

Given d triples, we construct a $d+1$-dimensional lattice using the rows of the matrix

$$B = \begin{pmatrix} 2^{\ell_1+1} \cdot q & & & \\ & \ddots & & \\ & & 2^{\ell_d+1} \cdot q & \\ 2^{\ell_1+1} \cdot t_1 & \ldots & 2^{\ell_d+1} \cdot t_d & 1 \end{pmatrix}.$$

By the definition of v_i, there are integers λ_i such that $v_i = \lambda_i \cdot q + \alpha \cdot t_i - u_i$. Consequently, for the vectors $\mathbf{x} = (\lambda_1, \ldots, \lambda_d, \alpha)$, $\mathbf{y} = (2^{\ell_1+1} \cdot v_1, \ldots, 2^{\ell_d+1} \cdot v_d, \alpha)$ and $\mathbf{u} = (2^{\ell_1+1} \cdot u_1, \ldots, 2^{\ell_d+1} \cdot u_d, 0)$ we have

$$\mathbf{x} \cdot B - \mathbf{u} = \mathbf{y}.$$

The 2-norm of the vector $\mathbf{y}$ is about $\sqrt{d+1} \cdot q$ whereas the determinant of the lattice $L(B)$ is $2^{d+\sum l_i} \cdot q^d$. Hence $\mathbf{y}$ is a short vector in the lattice and the vector $\mathbf{u}$ is close to the lattice vector $\mathbf{x} \cdot B$. We can now solve the Closest Vector Problem (CVP) with inputs B and $\mathbf{u}$ to find $\mathbf{x}$, revealing the value of the hidden number α.

2.5.3 *Related Work*

Several authors describe attacks on cryptographic systems that exploit partial nonce disclosure to recover long-term private keys.

Brumley and Hakala [14] use an L1 data cache-timing attack to recover the LSBs of ECDSA nonces from the `dgst` command line tool in OpenSSL 0.9.8k. They collect 2,600 signatures (8K with noise) and use the Howgrave-Graham and Smart [21] attack to recover a 160-bit ECDSA private key. In a similar vein, Acıiçmez et al. [3] use an L1 instruction cache-timing attack to recover the LSBs of DSA nonces from the same tool in OpenSSL 0.9.8l, requiring 2,400 signatures (17K with noise) to recover a 160-bit DSA private key. Both attacks require HyperThreading architectures.

Brumley and Tuveri [15] mount a remote timing attack on the implementation of ECDSA with binary curves in OpenSSL 0.9.8o. They show that the timing leaks information on the MSBs of the nonce used and that after collecting that information over 8,000 TLS handshakes the private key can be recovered.

Benger et al. [9] recover the secret key of OpenSSL's ECDSA implementation for the curve *secp256k1* using less than

256 signatures. They use the FLUSH+RELOAD technique to find some LSBs of the nonces and extend the lattice technique of Howgrave-Graham and Smart [21] to use all of the leaked bits rather than limiting to a fixed number.

Van de Pol et al. [32] exploit the structure of the modulus in some elliptic curves to use all of the information leaked in consecutive sequences of bits anywhere in the top half of the nonces, allowing them to recover the secret key after observing only a handful of signatures. Allan et al. [4] improve on these results by using a performance-degradation attack to amplify the side-channel. The amplification allows them to observe the sign bit in the wNAF representation used in OpenSSL 1.0.2a and to recover a 256 bit key after observing only 6 signatures.

Genkin et al. [19] perform electromagnetic and power analysis attacks on mobile phones. They show how to construct HNP triples when the signature uses the low s-value [38].

3. A NEW SOFTWARE DEFECT

Percival [31] demonstrated that the SWE implementation of modular exponentiation in OpenSSL version 0.9.7g is vulnerable to cache-timing attacks, applied to recover RSA private keys. Following the issue, the OpenSSL team committed two code changes relevant to this work. The first[3] adds a "constant-time" implementation of modular exponentiation, with a fixed-window implementation and using the scatter-gather method [13, 41] of masking table access to the multipliers.

The new implementation is slower than the original SWE implementation. To avoid using the slower new code when the exponent is not secret, OpenSSL added a flag (`BN_FLG_CONSTTIME`) to its representation of big integers. When the exponent should remain secret (e.g. in decryption and signing) the flag is set (e.g. in the case of DSA nonces, Figure 1, Line 252) at runtime and the exponentiation code takes the "constant-time" execution path (Figure 2, Line 413). Otherwise, the original SWE implementation is used.

The execution time of the "constant-time" implementation still depends on the bit length of the exponent, which in the case of DSA should be kept secret [12, 15, 27]. The second commit[4] aims to "make sure DSA signing exponentiations really are constant-time" by ensuring that the bit length of the exponent is fixed. This safe default behavior can be disabled by applications enabling the `DSA_FLAG_NO_EXP_CONSTTIME` flag at runtime within the DSA structure, although we are not aware of any such cases.

To get a fixed bit length, the DSA implementation adds γq to the randomly chosen nonce, where $\gamma \in \{1, 2\}$, such that the bit length of the sum is one more than the bit length of q. More precisely, the implementation creates a copy of the nonce k (Figure 1, Line 264), adds q to it (Line 274), checks if the bit length of the sum is one more than that of q (Line 276), otherwise it adds q again to the sum (Line 277). If q is n bits, then $k + q$ is either n or $n + 1$ bits. In the former case, indeed $k + 2q$ is $n + 1$ bits. As an aside, we note the code in question is not constant-time and potentially leaks the value of γ. Such a leak would create a bias that can be exploited to mount the Bleichenbacher attack [5, 11, 18].

While the procedure in this commit ensures that the bit length of the sum kq is fixed, unfortunately it introduces a software defect. The function `BN_copy` is not designed to propagate flags from the source to the destination. In fact, OpenSSL exposes a distinct API `BN_with_flags` for that functionality—quoting the documentation:

> `BN_with_flags` creates a temporary shallow copy of `b` in `dest` ... Any flags provided in `flags` will be set in `dest` in addition to any flags already set in `b`. For example this might commonly be used to create a temporary copy of a BIGNUM with the `BN_FLG_CONSTTIME` flag set for constant time operations.

In contrast, with `BN_copy` the `BN_FLG_CONSTTIME` flag does not propagate to kq. Consequently, the sum is not treated as secret, reverting the change made in the first commit—when the exponentiation wrapper subsequently gets called (Figure 1, Line 285), it fails the security-critical branch. Following a debug session in Figure 2, indeed the flag (explicit value `0x4`) is not set, and the execution skips the call to `BN_mod_exp_mont_consttime` and instead continues with the insecure SWE code path for DSA exponentiation.

In addition to testing our attack against OpenSSL (1.0.2h), we reviewed the code of two popular OpenSSL forks: LibreSSL[5] and BoringSSL[6]. Using builds with debugging symbols, we confirm both LibreSSL[7] and BoringSSL[8] share the same defect. It is worth noting that BoringSSL stripped out TLS DSA cipher suites in late 2014[9].

4. EXPLOITING THE DEFECT

In this section we describe how we use and combine the FLUSH+RELOAD technique with a performance degradation technique [4] to attack the OpenSSL implementation of DSA.

We tested our attack on an Intel Core i5-4570 Haswell Quad-Core 3.2GHz (22nm) with 16GB of memory running 64-bit Ubuntu 14.04 LTS "Trusty". Each 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 lines). It does not feature HyperThreading.

We used our own build of OpenSSL 1.0.2h which is the same default build of OpenSSL but with debugging symbols on the executable. Debugging symbols facilitate mapping source code to memory addresses but they are not loaded during run time, thus the victim's performance is not affected. Debugging symbols are, typically, not available to attackers but using reverse engineering techniques [16] is possible to map source code to memory addresses.

As previously discussed in Section 2.5, for DSA-type signatures, knowing a few bits of sufficiently many signature nonces allows an attacker to recover the secret key. This is the goal of our attack: we trace and recover side-channel information of the SWE algorithm that reveals the sequence

[3] https://github.com/openssl/openssl/commit/46a643763de6d8e39ecf6f76fa79b4d04885aa59

[4] https://github.com/openssl/openssl/commit/0ebfcc8f92736c900bae4066040b67f6e5db8edb

[5] https://www.libressl.org

[6] https://boringssl.googlesource.com/boringssl

[7] https://github.com/libressl-portable/openbsd/blob/master/src/lib/libssl/src/crypto/dsa/dsa_ossl.c

[8] https://boringssl.googlesource.com/boringssl/+/master/crypto/dsa/dsa.c

[9] https://boringssl.googlesource.com/boringssl/+/ef2116d33c3c1b38005eb59caa2aaa6300a9b450

```
246     /* Get random k */
247     do
248         if (!BN_rand_range(&k, dsa->q))
249             goto err;
250     while (BN_is_zero(&k)) ;
251     if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
252         BN_set_flags(&k, BN_FLG_CONSTTIME);
253     }
...
263     if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
264         if (!BN_copy(&kq, &k))
265             goto err;
266
267         /*
268          * We do not want timing information to leak the length of k, so we
269          * compute g^k using an equivalent exponent of fixed length. (This
270          * is a kludge that we need because the BN_mod_exp_mont() does not
271          * let us specify the desired timing behaviour.)
272          */
273
274         if (!BN_add(&kq, &kq, dsa->q))
275             goto err;
276         if (BN_num_bits(&kq) <= BN_num_bits(dsa->q)) {
277             if (!BN_add(&kq, &kq, dsa->q))
278                 goto err;
279         }
280
281         K = &kq;
282     } else {
283         K = &k;
284     }
285     DSA_BN_MOD_EXP(goto err, dsa, r, dsa->g, K, dsa->p, ctx,
286                    dsa->method_mont_p);
```

Figure 1: Excerpt from OpenSSL's `dsa_sign_setup` in `crypto/dsa/dsa_ossl.c`. Line 252 sets the `BN_FLG_CONSTTIME` flag, yet `BN_copy` on Line 264 does not propagate it. The subsequent Line 285 exponentiation call will have pointer `K` with the flag clear.

```
   +--bn_exp.c---------------------------------------------------------------------+
   |402     int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, const BIGNUM *p,        |
   |403                         const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *in_mont)  |
   |404     {                                                                        |
B+ |405         int i, j, bits, ret = 0, wstart, wend, window, wvalue;               |
   |406         int start = 1;                                                       |
   |407         BIGNUM *d, *r;                                                       |
   |408         const BIGNUM *aa;                                                    |
   |409         /* Table of variables obtained from 'ctx' */                         |
   |410         BIGNUM *val[TABLE_SIZE];                                             |
   |411         BN_MONT_CTX *mont = NULL;                                            |
   |412                                                                              |
  >|413         if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0) {                        |
   |414             return BN_mod_exp_mont_consttime(rr, a, p, m, ctx, in_mont);     |
   |415         }                                                                    |
   |416                                                                              |
   |417         bn_check_top(a);                                                     |
   +------------------------------------------------------------------------------+
   |0x7ffff779db3e <BN_mod_exp_mont+92>     mov    0x14(%rax),%eax                |
   |0x7ffff779db41 <BN_mod_exp_mont+95>     and    $0x4,%eax                      |
   |0x7ffff779db44 <BN_mod_exp_mont+98>     test   %eax,%eax                      |
  >|0x7ffff779db46 <BN_mod_exp_mont+100>    je     0x7ffff779db85 <BN_mod_exp_mont+163> |
   |0x7ffff779db48 <BN_mod_exp_mont+102>    mov    -0x1b0(%rbp),%r8               |
   +------------------------------------------------------------------------------+
child process 29096 In: BN_mod_exp_mont                     Line: 413  PC: 0x7ffff779db46
(gdb) break BN_mod_exp_mont
Breakpoint 1 (BN_mod_exp_mont) pending.
(gdb) run dgst -dss1 -sign ~/dsa.pem -out ~/lsb-release.sig /etc/lsb-release
Starting program: /usr/local/ssl/bin/openssl \
dgst -dss1 -sign ~/dsa.pem -out ~/lsb-release.sig /etc/lsb-release
Breakpoint 1, BN_mod_exp_mont (...) at bn_exp.c:405
(gdb) backtrace
#0  BN_mod_exp_mont (...) at bn_exp.c:405
#1  0x00007ffff77eea62 in dsa_sign_setup (...) at dsa_ossl.c:285
#2  0x00007ffff77ee344 in DSA_sign_setup (...) at dsa_sign.c:87
#3  0x00007ffff77ee53d in dsa_do_sign (...) at dsa_ossl.c:159
#4  0x00007ffff77ee30c in DSA_do_sign (...) at dsa_sign.c:75
...
(gdb) stepi
(gdb) info register eax
eax            0x0      0
(gdb) print BN_get_flags(p, BN_FLG_CONSTTIME)
$1 = 0
(gdb) macro expand BN_get_flags(p, BN_FLG_CONSTTIME)
expands to: ((p)->flags&(0x04))
(gdb) print ((p)->flags&(0x04))
$2 = 0
(gdb)
```

Figure 2: Debugging OpenSSL DSA signing in `crypto/bn/bn_exp.c`. The Line 413 branch is not taken since `BN_FLG_CONSTTIME` is not set, as seen from the print command outputs. Hence `BN_mod_exp_mont_consttime` is not called— the control flow continues with classical SWE code.

of *squares* and *multiplications*, from that sequence we recover a few bits that we use for the lattice attack described in Section 6.

As seen in Figure 2, every time OpenSSL performs a DSA signature, the exponentiation method `BN_mod_exp_mont` in `crypto/bn/bn_exp.c` gets called. There, the `BN_FLG_CONSTTIME` flag is checked but due to the software defect discussed in Section 3 the condition fails and the routine continues with the SWE pre-computation and then the actual exponentiation. For the finite field operations, `BN_mod_exp_mont` calls `BN_mod_mul_montgomery` in `crypto/bn/bn_mont.c` and from there, the multiply wrapper `bn_mul_mont` is called, where, by default for x64 targets, assembly code is executed to perform low level operations using BIGNUMs for square and multiplication. OpenSSL uses Montgomery representation for efficiency. Note that for other platforms and/or non-default build configurations, the actual code executed ranges from pure C implementation to entirely different assembly. The attacker can easily adapt to these different execution paths, but the discussion that follows is geared towards our target platform.

The threshold set for the load time in the Flush+Reload technique (cache hit vs. cache miss) is system and software dependent. From our measurements we set this threshold accordingly since the load times from LLC and from memory were clearly defined. Figure 4 shows that loads from LLC take less that 100 cycles, while loads from main memory take more than 200 cycles.

As mentioned before, to get better resolution and granularity during the attack one effective strategy is to target body loops or routines that are invoked several times. For that reason we probe, using the Flush+Reload technique, inner routines used for square and multiply. Since squares can be computed more efficiently than multiplication, OpenSSL's multiply wrapper checks if the two pointer operands are the same and, if so, calls to assembly squaring code (`bn_sqr8x_mont`)—otherwise, to assembly multiply code (`bn_mul4x_mont`).

At the same time we run a performance degradation attack, flushing actively used memory addresses during these routines (e.g. assembly labels `Lsqr4x_inner` and `Linner4x`, respectively). We slow down the execution time to a safe, but not noticeable by the victim, threshold that ensures a good trace by our spy program. In our experiments, we observe slow down factors of roughly 16 and 26 for 1024-bit and 2048-bit DSA, respectively due to the degrade technique.

Using this strategy, our spy program collects data from two channels: one for square latencies and the other for multiply latencies. We then apply signal processing techniques to this raw channel data. A moving average filter on the data results in Figure 3 and Figure 4 for 1024-bit and 2048-bit DSA, respectively. There is a significant amount of information to extract from these signals on the SWE algorithm state transitions and hence exponent bit values. Generally, extracted multiplications yield a single bit of information and the squares yield the position for these bits. Some short examples follow.

Stepping through Figure 3, the initial low amplitude for the multiply signal is the multiplication for converting the base operand to Montgomery representation. The subsequent low amplitude for the square signal is the temporary square value used to build the odd powers for the SWE precomputation table (i.e. s in Algorithm 2). The subsequent long period of low multiply amplitude is the successive multiplications to build the pre-computation table itself. Then begins the main loop of SWE. As an upward sloping multiply amplitude intersects a downward sloping square amplitude, this marks the transition from a multiplication operation to a square operation (and vice versa). This naturally occurs several times as the main exponentiation loop iterates. The end of this particular signal shows a final transition from multiply to a single square, indicating that the exponent is even and the two LSBs are 1 and 0.

Stepping through Figure 4 is similar, yet the end of this particular signal shows a final transition from square to multiply—indicating that the exponent is odd, i.e. the LSB is 1.

Even when employing the degrade technique, it is important to observe the vast granularity difference between these two cryptographic settings. On average, a 2048-bit signal is roughly ten times the length of a 1024-bit signal, even when the exponent is only 60% longer (i.e. 256-bit vs. 160-bit). This generally suggests we should be able to extract more accurate information from 2048-bit signals than 1024-bit—i.e., the higher security cryptographic parameters are more vulnerable to side-channel attack in this case. See [37, 39, 40] for similar examples of this phenomenon.

Granularity is vital to determining the number of squares interleaved between multiplications. Since, in our environment, there appears to be no reliable indicator in the signal for transitions from one square to the next, we estimate the number of adjacent squares by the horizontal distance between multiplications. Since the channel is latency data, we also have reference clock cycle counter values so another estimate is based on the counter differences at these points. Our experiments showed no significant advantage of one approach over the other.

Extracting the multiplications from the signal and interleaving them with a number of consecutive squares proportional to the width of the corresponding gap gives us the square and multiplication sequence, or *SM sequence*, that the SWE algorithm passed through. Figure 7 shows an example of an SM sequence recorded by the spy program when OpenSSL signs using 2048-bit DSA.

Our spy program is able to capture most of the SM sequence accurately. It can miss or duplicate a few squares due to drift but is able to capture all of the multiplication operations. Closer to the LSBs, the information extracted from the SM sequence is more reliable since the bit position is lost if any square operation is missed during probing.

5. VICTIMIZING APPLICATIONS

The defect from the previous section is in a shared library. Potentially any application that links against OpenSSL for DSA functionality can be affected by this vulnerability. But to make our attack concrete, we focus on two ubiquitous protocols and applications: TLS within stunnel and SSH within OpenSSH.

As we discuss later in Section 6, the trace data alone is not enough for private key recovery—we also need the digital signatures themselves and (hashed) messages. To this end, the goal of this section is to describe the practical tooling we developed to exploit the defect within these applications, collecting both trace data and protocol messages.

5.1 Attacking TLS

To feature TLS support, one option for network applications that do not natively support TLS communication is to use stunnel[10], a popular portable open source software package that forwards network connections from one port to another and provides a TLS wrapper. A typical stunnel use case is listening on a public port to expose a TLS-enabled network service, then connecting to a localhost port where a non-TLS network service is listening—stunnel provides a TLS layer between the two ports. It links against the OpenSSL shared library to provide this functionality. For our experiments, we used stunnel 5.32 compiled from stock source and linked against OpenSSL 1.0.2h. We generated a 2048-bit DSA certificate for the stunnel service and chose the `DHE-DSS-AES128-SHA256` TLS 1.2 cipher suite.

We wrote a custom TLS client that connects to this stunnel service. It launches our spy to collect the timing signals, but its main purpose is to carry out the TLS handshake and collect the digital signatures and protocol messages. Figure 5 shows the TLS handshake. Relevant to this work, the initial `ClientHello` message contains a 32-byte `random` field, and similarly the server's `ServerHello` message. In practice, these are usually a 4-byte UNIX timestamp concatenated with a 28-byte nonce. The `Certificate` message contains the DSA certificate we generated for the stunnel service. The `ServerKeyExchange` message contains a number of critical fields for our attack: Diffie-Hellman key exchange parameters, the signature algorithm and hash function identifiers, and finally the digital signature itself in the `signed_params` field. Given our stunnel configuration and certificate, the 2048-bit DSA signature is over the concatenated string

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

and the hash function is SHA-512, both dictated by the `SignatureAndHashAlgorithm` field (explicit values `0x6`, `0x2`). Our client saves the hash of this string and the DER-encoded digital signature sent from the server. All subsequent messages, including `ServerHelloDone` and any client responses, are not required by our attack. Our client therefore drops the connection at this stage, and repeats this process several hundred times to build up a set of distinct trace, digital signature, and digest tuples. See Section 6 for our explicit attack parameters. Figure 4 is a typical signal extracted by our spy program in parallel to the handshake between our client and the victim stunnel service.

5.2 Attacking SSH

OpenSSH[11] is a suite of tools whose main goal is to provide secure communications over an insecure channel using the SSH network protocol.

OpenSSH is linked to the OpenSSL shared library to perform several cryptographic operations, including digital signatures (excluding `ed25519` signatures). For our experiments we used the stock OpenSSH 6.6.1p1 binary package from the Ubuntu repository, and pointed the run-time shared library loader at OpenSSL 1.0.2h. The DSA key pair used by the server and targeted by our attack is the default 1024-bit key pair generated during installation of OpenSSH.

Similar to Section 5.1, we wrote a custom SSH client that

[10] https://www.stunnel.org
[11] http://www.openssh.com

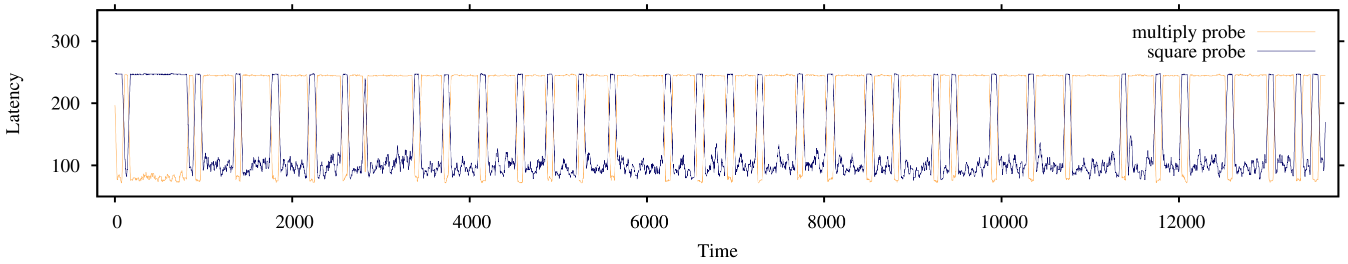

Figure 3: Complete filtered trace of a 1024-bit DSA sign operation during an OpenSSH SSH-2 handshake.

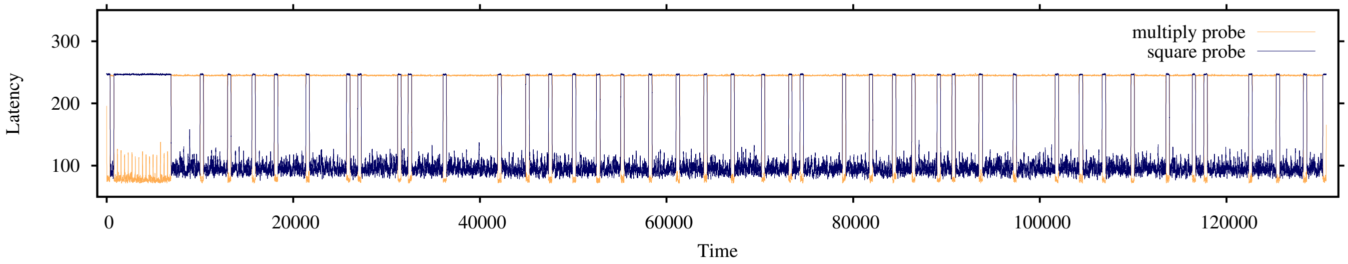

Figure 4: Complete filtered trace of a 2048-bit DSA sign operation during an stunnel TLS 1.2 handshake.

launches our spy program, the spy program collects the timing signals during the handshake. At the same time it performs an SSH handshake where the protocol messages and the digital signature are collected for our attack.

Relevant to this work, the SSH protocol defines the Diffie-Hellman key exchange parameters in the `SSH_MSG_KEXINIT` message, along with the signature algorithm and the hash function identifiers. Additionally a 16-byte `random` nonce is sent for host authentication by the client and the server.

The `SSH_MSG_KEXDH_REPLY` message contains the server's public key (used to create and verify the signature), server's DH public key `f` (used to compute the shared secret `K` in combination with the client's DH public key `e`) and the signature itself. Figure 6 shows the SSH handshake with the critical parameters sent in every message relevant for the attack. To be more precise, the signature is over the SHA-1 hash of the concatenated string

```
ClientVersion + ServerVersion +
Client.SSH_MSG_KEXINIT + Server.SSH_MSG_KEXINIT +
Server.publicKey + minSize + prefSize + maxSize +
p + g + e + f + K
```

As the key exchange[12] and public key parameters, our SSH client was configured to use `diffie-hellman-group-exchange-sha1` and `ssh-dss` respectively. Note that two different hashing functions may be used, one hash function for key derivation following Diffie-Hellman key exchange and another hash function for the signing algorithm, which for DSA is the SHA-1 hash function.

Similarly to the TLS case, our client saves the hash of the concatenated string and the digital signature raw bytes sent from the server. All subsequent messages, including `SSH_MSG_NEWKEYS` and any client responses, are not required by our attack. Our client therefore drops the connection at this stage, and repeats this process several hundred times to build up a set of distinct trace, digital signature, and digest tuples. See Section 6 for our explicit attack parameters. Figure 3 is a typical signal extracted by our spy program in parallel to the handshake between our client and the victim SSH server.

5.3 Observations

These two widely deployed protocols share many similarities in their handshakes regarding e.g. signaling, content of messages, and security context of messages. However, in the process of designing and implementing our attacker clients we observe a subtle difference in the threat model between the two. In TLS, all values that go into the hash function to compute the digital signature are public and can be observed (unencrypted) in various handshake messages. In SSH, *most* of the values are public—the exception is the last input to the hash function: the shared DH key. The consequence is side-channel attacks against TLS can be passive, listening to legitimate handshakes not initiated by the attacker yet collecting side-channel data as this occurs. In SSH, the attacker must be active and initiate its own handshakes—without knowing the shared DH key, a passive attacker cannot compute the corresponding digest needed later for the lattice stage of the attack. We find this innate protocol level side-channel property to be an intriguing feature, and a factor that should be carefully considered during protocol design.

6. RECOVERING THE PRIVATE KEY

In previous sections we showed how our attack can recover the sequence of square and multiply operations that the victim performs. We further showed how to get the signature information matching each sequence for both SSH and TLS. We now turn to recovering the private key from the information we collect.

The scheme we use is similar to past works. We first

[12] https://tools.ietf.org/html/rfc4419

```
Client                                     Server

ClientHello
[random,
CipherSuite]        ------->
                                      ServerHello
                                         [random,
                                    cipher_suite]
                                      Certificate
                                ServerKeyExchange
                                         [params,
                                   HashAlgorithm,
                              SignatureAlgorithm,
                                   signed_params]
                    <-------      ServerHelloDone
ClientKeyExchange
(ChangeCipherSpec)
Finished            ------->
                               (ChangeCipherSpec)
                    <-------             Finished
Application Data    <------->    Application Data
```

Figure 5: Our custom client carries out TLS handshakes, collecting certain fields from the `ClientHello`, `ServerHello`, and `SeverKeyExchange` messages to construct the digest. It collects timing traces in parallel to the server's DSA sign operation, said digital signature being included in a `SeverKeyExchange` field and collected by our client.

```
Client                                     Server

ClientVersion         ----->
                                    ServerVersion
                                  SSH_MSG_KEXINIT
                                 [Nonce, KEX_alg,
                      <-----       publicKey_alg]
SSH_MSG_KEXINIT
[Nonce, KEX_alg,
publicKey_alg]
SSH_KEXDH_GEX_REQUEST
[minSize, prefSize,
maxSize]              ----->
                              SSH_KEXDH_GEX_GROUP
                      <-----               [p, g]
SSH_KEXDH_GEX_INIT
[e]                   ----->
                              SSH_KEXDH_GEX_REPLY
                                  [publicKey, f,
                      <-----           Signature]
SSH_MSG_NEWKEYS
                      ----->
Application Data      <---->     Application Data
```

Figure 6: Our custom client carries out SSH handshakes, collecting parameters from all the messages to construct the digest. It collects timing traces in parallel to the server's DSA sign operation, said digital signature being included in a `SSH_KEXDH_GEX_REPLY` field and collected by our client.

Table 1: Empirical results of recovering various LSBs from the spy program traces and their corresponding SM sequences.

ℓ	a	Pattern	Accuracy (%) 1024-bit, SSH	Accuracy (%) 2048-bit, TLS
1	1	SSM	99.9	99.9
2	2	SMS	99.9	99.7
2	3	SMSM	98.2	97.2
3	4	SSMSS	99.7	99.7
3	6	SMSMS	99.4	98.2
4	8	SSMSSS	97.8	99.6
4	12	SMSMSS	98.4	97.8
5	16	SSMSSSS	96.7	99.1
5	24	SMSMSSS	95.0	97.6
6	32	SSMSSSSS	85.1	98.8
6	48	SMSMSSSS	90.4	95.0
7	64	SSMSSSSSS	87.5	97.5
7	96	SMSMSSSSS	84.6	95.1
8	128	SSMSSSSSSS	67.7	98.7
8	192	SMSMSSSSSS	75.0	94.8

```
SMMMMMMMMMMMMMMMMSSSMSSSSSSSMSSSSMSSSSSSSMSSSSSSM
SSSSSSMSSSSSSSSMSSSMSSSSSSSSSSMSSSSSSSSMSSSSSMSSS
SSMSSSSSSSSSMSSSSSMSSSSSSMSSSSSMSSSSSSSMSSSSMSSSS
SSSMSSSSSSMSSSMSSSSSSSSSSMSSSSSSSSMSSSMSSSSSSSMSSS
SSMSSSSSMSSSSMSSSSSSMSSSMSSSSSSMSSSSSSMSSMSSSSSSS
SSMSSSSMSSSSSSSSMSSSSSSSSSMSMSSSSSMSSSSSSSSSMSSSS
SSSSSSMSSSSM
```

Figure 7: Example of an extracted SM sequence, where `S` and `M` are square and multiply, respectively.

use the side-channel information we capture to collect information on the nonce used in each signature. We use the information to construct HNP instances and use a lattice technique to find the private key. Further details on each step are provided below.

6.1 Extracting the least significant bits

In Section 4 we showed how we collect the SM sequences of each exponentiation. From every SM sequence, we extract a few LSBs to be used later in the lattice attack. To that end, Table 1 contains our empirical accuracy statistics for various relevant patterns trailing the SM sequences, and furthermore not for the SWE in isolation but rather in the context of OpenSSL DSA executing in real world applications (TLS via stunnel, SSH via OpenSSH), as described above in Section 5. All of these patterns correspond to recovering $a = \bar{k} \bmod 2^{\ell}$ for an exponent $\bar{k}$. From these figures, we note two trends. (1) The accuracy decreases as ℓ increases due to deviation in the square operation width. Yet weighed with the exponentially decreasing probability of the longer patterns, the practical impact diminishes. (2) As expected, we generally obtain more accurate results with 2048-bit vs. 1024-bit due to granularity. These numbers show that, exploiting our new software defect and leveraging the techniques in Section 4, we can recover a with extremely high probability.

6.2 Lattice attack implementation

Recall that to protect against timing attacks OpenSSL uses an exponent $\bar{k}$ equivalent to the randomly selected nonce k. $\bar{k}$ is calculated by adding the modulus q once or twice to k to ensure that $\bar{k}$ is of a fixed length. That is, $\bar{k} = k + \gamma q$ such that $2^n \leq \bar{k} < 2^n + q$ where $n = \lceil \lg(q) \rceil$ and $\gamma \in \{1, 2\}$.

The side-channel leaks information on bits of the exponent $\bar{k}$ rather than directly on the nonce. To create HNP instances from the leak we need to handle the unknown value of γ. In previous works, due to ECC parameters the modulus is close to a power of two hence the value of γ is virtually constant [9]. For DSA, the modulus is not close to a power of two and the value of γ varies between signatures. The challenge is, therefore, to construct an HNP instance without the knowledge of γ. We now show how to address this challenge.

Recall that $s = k^{-1}(h(m) + \alpha r) \bmod q$. Equivalently, $k = s^{-1}(h(m) + \alpha r) \bmod q$. The side-channel information recovers the ℓ LSBs of $\bar{k}$. We, therefore, have $\bar{k} = b2^\ell + a$ where $a = \bar{k} \bmod 2^\ell$ is known, and

$$2^{n-\ell} \leq b < 2^{n-\ell} + \left\lceil q/2^\ell \right\rceil . \qquad (1)$$

Following previous works we use $\lfloor \cdot \rfloor_q$ to denote the reduction modulo q to the range $[0, q)$ and $| \cdot |_q$ for the reduction modulo q to the range $(-q/2, q/2)$. Within these expressions division operations are carried over the reals whereas all other operations are carried over $GF(q)$.

We now look at $\lfloor b - 2^{n-\ell} \rfloor_q$.

$$\begin{aligned}
& \lfloor b - 2^{n-\ell} \rfloor_q \\
= \; & \lfloor (\bar{k} - a) \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
= \; & \lfloor \bar{k} \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
= \; & \lfloor k \cdot 2^{-\ell} + \gamma \cdot q \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
= \; & \lfloor k \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
= \; & \lfloor (s^{-1} \cdot (h(m) + \alpha \cdot r) \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
= \; & \lfloor \alpha \cdot s^{-1} \cdot r \cdot 2^{-\ell} - (2^n + a - s^{-1} \cdot h(m)) \cdot 2^{-\ell} \rfloor_q
\end{aligned}$$

Hence, we can set:

$$\begin{aligned}
t &= \left\lfloor s^{-1} \cdot r \cdot 2^{-\ell} \right\rfloor_q \\
u &= \left\lfloor (2^n + a - s^{-1} \cdot h(m)) \cdot 2^{-\ell} + \left\lceil \frac{q}{2^{\ell+1}} \right\rceil \right\rfloor_q \\
v &= |\alpha \cdot t - u|_q
\end{aligned}$$

and by (1) we have $|v| \leq \lceil q/2^{\ell+1} \rceil$.

Out of the HNP instances we generate, we select at random 49 for the SSH attack, 130 for the TLS attack and construct a lattice as described in Section 2.5.2. We solve the CVP problem with a Sage script, performing lattice reduction using BKZ [34], and enumerate the lattice points using Babai's Nearest Plane (NP) algorithm [7]. We apply two different techniques to extend NP to a larger search space. First, we take multiple rounding values to explore 2^{10} different solutions in the tree paths [23, Sec. 4]. Second, we use a randomization technique [26, Sec. 3.5] and shuffle the rows of B between lattice reductions. We repeat with a different random selection of instances until we find the private key.

Table 2: Empirical lattice attack results over a thousand trials. Set size and errors are mean values. Iterations and CPU time are median values.

Victim	OpenSSH (SSH)	stunnel (TLS)
Key size	1024/160-bit	2048/256-bit
Handshakes	260	580
Lattice size	50	131
Set size	70.8	158.1
Errors	2.1	1.7
Iterations	13	22
CPU minutes	5.9	38.8
Success rate (%)	100.0	100.0

6.3 Results

We implemented the attack and evaluated it against the two protocols, SSH with 1024/160-bit DSA and TLS with 2048/256-bit DSA. Table 2 contains the results. For both protocols we only utilize traces with $\ell \geq 3$. With this value we experimentally found that we require 49 such signatures for SSH and 130 for TLS in order to achieve a reasonable probability of solving the resulting CVP.

Because the nonces are chosen uniformly at random, only about one in every four signatures has an ℓ that we can utilize. To gather enough signatures and to compensate for possible trace errors, we collect 580 SM sequences from TLS handshakes and 260 from SSH.

On average, these collected sequences yield 70.8 (SSH) and 158.1 (TLS) traces with $\ell \geq 3$. Comparing the traces to the ground truth, we know that on average less than 3 have trace errors. However, because an adversary cannot check against the ground truth, we leave these erroneous traces in the set and use them in the attack. We note that due to the smaller key size in SSH, trace errors are much more prevalent there.

We construct a lattice from a random selection of the collected traces and attempt to solve the resulting CVP. Due to the presence of the error traces there is a non-negligible probability that our selected set contains an error. Furthermore, even if all the chosen traces are correct, the algorithm may fail to find the target solution due to the heuristic nature of lattice techniques. In case of failure, we repeat the process with a new random selection from the same set. We need to execute a median value of 13 iterations for SSH and 22 for TLS until we find the target solution.

As seen from Table 2, repeating our experiment over a thousand trials on a cluster with hundreds of nodes, mixed between Intel X5660 and AMD Opteron 2435 cores, we find the private key in all cases requiring a median 5.9 CPU minutes for the SSH key and 38.8 CPU minutes for the TLS key. Although we executed each trial on a single core, in reality the iterations are independent of each other—the lattice attack is embarrassingly parallel.

7. CONCLUSION

In this work we disclose a programming error in OpenSSL that results in a security weakness. We show that as a result of the defect, the DSA implementation in OpenSSL is vulnerable to cache-timing attacks, and exploit the vulnerability to mount end-to-end attacks against SSH (via OpenSSH) and TLS (via stunnel).

It is all too easy to dismiss the bug as an innocent programming error. However, we believe that the core issue is a design problem. When designing the "constant-time" fix, the developers elected to use an insecure default behavior. From an engineering perspective the decision is justified—it is much easier to identify the handful of locations where we know that the exponent should be kept secret than to analyze the entire library identifying exponents that can be leaked. However, from a security perspective, this design decision breaches the principle of *fail-safe defaults*, which Saltzer and Schroeder [33] justify by saying: *a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use.*

It is hard not to appreciate the extraordinary prescience of Saltzer and Schroeder's justification. Had OpenSSL elected to use a better design, that defaults to the constant-time behavior, a similar bug could have resulted in a small performance loss for non-sensitive exponentiations, but the omission to preserve the flag in question would have been unlikely to jeopardize the security of the system. A more secure design would also improve the security of third-party products in the case that developers may not be aware of the intricacies of the constant-time flags.

The simplest software-based solution to mitigate our attack is to fix the software defect. During responsible disclosure, OpenSSL, LibreSSL, and BoringSSL merged patches for CVE-2016-2178[13], assigned as a result of this work.

Broader, the `clflush` instruction does not require elevated privileges to execute, hence we suggest access control mechanisms. We recommend that cache flush instructions be privileged-only execution, or at least restricted to memory pages to which the process has write access and to memory pages explicitly allowed by the kernel. Partially or fully disabling caching during sensitive code execution can prevent cache-timing attacks at the cost of performance [3]. Preventing page sharing between processes is a partial solution at the cost of increased memory requirements and avoiding sharing of sensitive code is possible by changing the program loader.

We close with some practical advice regarding this vulnerability. OpenSSH supports building without OpenSSL as a dependency. We recommend that OpenSSH package maintainers switch to this option. For OpenSSH administrators and users, we recommend migrating to `ssh-ed25519` key types, the implementation of which has many desirable side-channel properties. Furthermore, ensure that `ssh-dss` is absent from the `HostKeyAlgorithms` configuration field, and any such `HostKey` entries removed. On the TLS side, we recommend disabling cipher suites that have DSA functionality as a pre-requisite.

Acknowledgments

The first author is supported by the Erasmus Mundus NordSecMob Master's Programme and the European Commission.

The first and second authors are supported in part by TEKES grant 3772/31/2014 Cyber Trust.

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

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

References

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *2007 CT-RSA*, pages 225–242, 2007.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *2nd AsiaCCS*, Singapore, 2007.

[3] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, Santa Barbara, CA, US, 2010.

[4] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. IACR Cryptology ePrint Archive, Report 2015/1141, Nov 2015.

[5] Diego F. Aranha, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias. In *ASIACRYPT*, pages 262–281, Kaohsiung, TW, Dec 2014.

[6] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Linux symposium*, pages 19–28, 2009.

[7] László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1): 1–13, March 1986.

[8] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. NIST Special Publication 800-131A Revision 1, Nov 2015. URL http://dx.doi.org/10.6028/NIST.SP.800-131Ar1.

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

[10] Daniel J Bernstein. Cache-timing attacks on AES, 2005. Preprint available at http://cr.yp.to/papers.html#cachetiming.

[11] Daniel Bleichenbacher. On the generation of one-time keys in DL signature schemes. Presentation at IEEE P1363 Working Group meeting, Nov 2000.

[12] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *CRYPTO'96*, pages 129–142, Santa Barbara, CA, US, Aug 1996.

[13] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating cache/timing based side-channels in AES and RSA software implementations. RSA Conference 2006 session DEV-203, Feb 2006.

[13] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2178

[14] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *15th ASIACRYPT*, pages 667–684, Tokyo, JP, Dec 2009.

[15] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *16th ESORICS*, Leuven, BE, 2011.

[16] Teodoro Cipresso and Mark Stamp. Software reverse engineering. In *Handbook of Information and Communication Security*, pages 659–696. 2010.

[17] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual, Jan 2016.

[18] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In *CHES*, pages 435–452, Santa Barabara, CA, US, Aug 2013.

[19] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. IACR Cryptology ePrint Archive, Report 2016/230, Mar 2016.

[20] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *S&P*, pages 490–505, May 2011.

[21] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *DCC*, 23(3): 283–290, Aug 2001.

[22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *S&P*, San Jose, CA, US, May 2015.

[23] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In *2011 CT-RSA*, pages 319–339, 2011.

[24] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. ARMageddon: Last-level cache attacks on mobile devices. *arXiv preprint arXiv:1511.04897*, 2015.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, pages 605–622, May 2015.

[26] Mingjie Liu and Phong Q Nguyen. Solving BDD by enumeration: An update. In *Topics in Cryptology–CT-RSA 2013*, pages 293–309. 2013.

[27] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptology*, 15(2):151–176, Jun 2002.

[28] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *DCC*, 30(2): 201–217, Sep 2003.

[29] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *2006 CT-RSA*, 2006.

[30] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.

[31] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.

[32] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *2015 CT-RSA*, pages 3–21, San Francisco, CA, USA, Apr 2015.

[33] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, Sep 1975.

[34] C. P. Schnorr and M. Euchner. Lattic basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Prog.*, 66(1–3):181–199, Aug 1994.

[35] Augustus K. Uht, Vijay Sindagi, and Kelley Hall. Disjoint eager execution: An optimal form of speculative execution. MICRO 28, pages 313–325, 1995.

[36] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, pages 181–194, Dec 2002.

[37] Colin D. Walter. Longer keys may facilitate side channel attacks. In *SAC*, pages 42–57, Waterloo, ON, Canda, Aug 2004.

[38] Pieter Wuille. Dealling with malleability. https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki, March 2014.

[39] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. IACR Cryptology ePrint Archive, Report 2014/140, Feb 2014.

[40] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security*, pages 719–732, San Diego, CA, US, 2014.

[41] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *CHES*, 2016.