Big Numbers – Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations

Samuel Weiser
Graz University of Technology

David Schrammel
Graz University of Technology

Lukas Bodner
Graz University of Technology

Raphael Spreitzer
SGS Digital Trust Services

Abstract

Side-channel attacks exploiting (EC)DSA nonce leakage easily lead to full key recovery. Although (EC)DSA implementations have already been hardened against side-channel leakage using the constant-time paradigm, the long-standing cat-and-mouse-game of attacks and patches continues. In particular, current code review is prone to miss less obvious side channels hidden deeply in the call stack. To solve this problem, a systematic study of nonce leakage is necessary.

We present a systematic analysis of nonce leakage in cryptographic implementations. In particular, we expand DATA, an open-source side-channel analysis framework, to detect nonce leakage. Our analysis identified multiple unknown nonce leakage vulnerabilities across all essential computation steps involving nonces. Among others, we uncover inherent problems in Bignumber implementations that break claimed constant-time guarantees of (EC)DSA implementations if secrets are *close* to a word boundary. We found that lazy resizing of Bignumbers in OpenSSL and LibreSSL yields a highly accurate and easily exploitable side channel, which has been acknowledged with two CVEs. Surprisingly, we also found a tiny but expressive leakage in the constant-time scalar multiplication of OpenSSL and BoringSSL. Moreover, in the process of reporting and patching, we identified newly introduced leakage with the support of our tool, thus preventing another attack-patch cycle. We open-source our tool, together with an intuitive graphical user interface we developed.

1 Introduction

Digital signatures are an essential building block for encrypted communication channels, e.g., via Transport Layer Security (TLS) and the underlying public key infrastructures, SSH, as well as for cryptocurrencies. The extensive and ubiquitous usage of digital signature schemes demands good security arguments, not only from a cryptanalytic perspective but also regarding their implementation, as a single implementation vulnerability can completely break the scheme [14].

Most digital signature schemes used today are susceptible to attacks on their so-called nonces [40]. Even partial knowledge of nonces leads to full recovery of private keys, thus allowing an attacker to issue fake signatures, impersonate users, intercept communication channels, steal money, etc. In light of these threats, digital signature implementations need extensive hardening against nonce leakage. While biased random number generation [14] is a common implementation pitfall, also side channels [15] have been proven a powerful way of leaking nonce bits. Especially side-channel attacks constantly improve along several axes. This includes advanced side-channel observation methods, a reduction of required knowledge, faster key recovery attacks, and most importantly, the continued discovery of new side-channel leakage.

Modern cryptographic libraries already explicitly address nonce leakage by relying on constant-time code execution. Unfortunately, efforts to make implementations side-channel resistant are not being evaluated thoroughly enough, leading to a continuous cycle of vulnerability disclosure and patching. To break this cycle, a more systematic approach for nonce leakage analysis is required. However, this seems to be a challenging endeavor for the following reasons:

1. Although side-channel evaluation is actively researched, complex code bases such as OpenSSL are hard to evaluate.
2. Popular libraries use randomization, e.g., blinding, to avoid leakage in vulnerable non-constant-time code. However, analyzing blinded computations for side channels is non-trivial; and insufficient blinding is exploitable.
3. Cryptographic libraries use non-constant-time code when computing on public data. Although legitimate, this puts additional burden on code analysis to avoid false positives.
4. Although tool support for side-channel analysis is growing, existing tools do not address nonce leakage.

We address these challenges by extending the DATA framework [55]. In particular, we adapt DATA to recognize nonces as additional secrets in a backward manner and develop leakage models tailored for detecting nonce leakage. With our statistical tests, we filter leakage results with respect to nonce leakage. We also develop a graphical user interface for vi-

sualizing leakage results. This allows us to systematically analyze three popular cryptographic libraries for (EC)DSA nonce leakage, namely OpenSSL, LibreSSL, and BoringSSL.

We systematically analyze the whole lifetime of a nonce, *i.e.*, from its generation to its final use. Rather than proving code secure—which would typically require formal models and static analysis approaches—we focus on finding actual side-channel vulnerabilities. We uncovered numerous unknown vulnerabilities leaking nonce bits, and thereby highlight a fundamental problem in the Bignumber representation in OpenSSL and LibreSSL. In particular, if the nonce is close to a machine word boundary, the Bignumber implementations possibly leak whether the nonce crosses this boundary in either direction. We found that lazy resize operations involving the nonce leak several nonce bits via Flush+Reload [61], as documented under CVE-2018-0734 and CVE-2018-0735. Surprisingly, this leakage occurs due to a side-channel defense mechanism. We also found that small nonces can leak nine nonce bits at once for the secp521r1 curve. The Bignumber implementation of BoringSSL [7] prevents size-related Bignumber issues by design. Yet, we found a tiny but expressive leak in the constant-time scalar multiplication of BoringSSL and OpenSSL. During responsible disclosure, we identified a flaw in the OpenSSL patches that would have downgraded exponentiation to a vulnerable implementation (cf. [24]). We report residual leakage in the patched OpenSSL version, which we exploit via controlled-channel attacks [59] for full key recovery. Due to our findings, the OpenSSL team decided to rework Bignumber arithmetic, similar to BoringSSL [19].

This work provides a snapshot of the current situation of nonce leakage in popular cryptographic libraries. With the help of our GUI we analyzed known and unknown vulnerabilities and document their potential damage, exploitability, and patching state. We open-source both our tool and the GUI to facilitate reproducibility and future side-channel analysis. [1]

Contributions. Our contributions are as follows:

- We expand an analysis framework for automated nonce leakage detection, and present results in an intuitive GUI.
- We systematically analyze nonce leakage in three popular crypto libraries: OpenSSL, LibreSSL, and BoringSSL.
- We document several unknown leakage vulnerabilities resulting from fundamental flaws in the Bignumbers representation of OpenSSL and LibreSSL, among others.
- We responsibly disclosed vulnerabilities, proposed fixes, and document residual leakage that remains unfixed.

Outline. Section 2 gives background information. Section 3 discusses related work on nonce attacks and side-channel analysis tools. Section 4 presents our automated side-channel analysis tool. Section 5 outlines analysis results and Section 6 discusses the vulnerabilities in detail. Section 7 evaluates our leakage models. We discuss the implications of our work in Section 8 and conclude in Section 9.

[1] Our tool and the GUI is available under https://github.com/Fraunhofer-AISEC/DATA and https://github.com/IAIK/data-gui

2 Background

2.1 Digital Signatures

DSA. The Digital Signature Algorithm (DSA) [29] is based on prime fields. It relies on two primes p and q, where q divides $p-1$. Parameter g serves as generator over p such that $g^q \equiv 1 \mod p$. Keys are generated as follows:

$$x \overset{R}{\leftarrow} [1, q-1] \quad (1) \qquad y \leftarrow g^x \mod p \quad (2)$$

The private key x is sampled uniformly from $[1, q-1]$. The public key y is obtained by Equation (2). The signature (r, s) for message m involves a random value k denoted as nonce:

$$k \overset{R}{\leftarrow} [1, q-1] \quad (3) \qquad kinv \leftarrow k^{-1} \mod q \quad (5)$$

$$r \leftarrow g^k \mod q \quad (4) \qquad s \leftarrow kinv \cdot (m + xr) \mod q \quad (6)$$

Other DSA Constructions. Several DSA variants exist. Schnorr signatures [47] omit the inversion step in Equation (5). Deterministic schemes [28, 44] derive unique nonces from the message input instead of using random numbers in Equation (3). ECDSA [29] is one of the most widely used signature algorithms nowadays. It computes r in Equation (4) via scalar multiplication over an elliptic curve generator G as follows:

$$r = k \cdot G \quad (7)$$

Nonce Attacks. DSA-like cryptosystems strongly rely on the secrecy and the uniformity of the nonce k. It has been shown that even partial knowledge of the nonce suffices to break the scheme [40]. This knowledge can be obtained by weak nonce generation algorithms [5] or side channels [15]. By collecting enough "leaky" signatures, one can formulate a so-called Hidden Number Problem (HNP) [10] and recover the private key with lattice or Bleichenbacher attacks. Thus, an implementation needs to properly address both cases and protect nonces throughout their whole lifetime (cf. Equations (3) to (6)).

2.2 The Hidden Number Problem

Nonce leakage can be encoded as a Hidden Number Problem (HNP). Solving the HNP via lattice attacks or more generic Bleichenbacher attacks reveals the private key.

HNP. The HNP [10, 11] denotes the problem of finding a hidden number given partial information about multiples of the hidden number. Following [6, 46], we denote $\lfloor \cdot \rfloor_q$ as the value modulo q and $|\cdot|_q$ as reducing the argument modulo q into the range $[-q/2, q/2]$ and then taking the absolute value. $MSB_{L,q}(k)$ denotes knowledge about the L most significant bits of k, *i.e.*, an integer u satisfying $|k-u|_q < q/2^{L+1}$.

The HNP attempts to recover a hidden number $x \in [1, q-1]$, given knowledge of its multiples $t_1, \ldots, t_d \in \mathbb{F}_q$ for a known prime q as well as knowledge about $u_i = MSB_{L,q}(\lfloor t_i x \rfloor_q)$. This yields a system of d inequalities:

$$|\lfloor t_i x \rfloor_q - u_i|_q < q/2^{L_i+1} \text{ for all } i \in \{1, \ldots, d\} \quad (8)$$

(EC)DSA can be encoded as an instance of the HNP to recover the private key x from signatures (r,s) and known nonce bits $u = MSB_{L,q}(k)$. Using Equation (6) gives:

$$|k-u|_q < q/2^{L+1} \quad (9)$$

$$|\lfloor (m+xr)\cdot s^{-1}\rfloor_q - u|_q < q/2^{L+1} \quad (10)$$

$$|\lfloor \lfloor s^{-1}r\rfloor_q \cdot x\rfloor_q - \lfloor u - s^{-1}m\rfloor_q|_q < q/2^{L+1} \quad (11)$$

Applying Equation (11) to d signatures (r_i, s_i) and nonce bits u_i yields an HNP. The HNP can also be applied when leaking inverse nonces, least significant nonce bits, or a block of contiguous [27] or non-contiguous bits [26].

Lattice. Boneh et al. [10] mapped the HNP to a Closest Vector Problem (CVP). Let $\mathbf{t} = (t_1, ..., t_d, 1)$ and $\mathbf{t}x = (t_1x, ..., t_dx, x)$. According to the HNP, $\lfloor \mathbf{t}x\rfloor_q$ will be a close vector to $\mathbf{u} = (u_1, ..., u_d, 0)$ with a distance smaller than $q/2^{L_i+1}$ for the first d components, *i.e.*, $\lfloor \mathbf{t}x\rfloor_q - \mathbf{u}$ will be *small* multiples of q. By constructing a lattice basis B from $\mathbf{t}$ and solving the CVP, the closest vector $\mathbf{t}x$ reveals the private key x. Boneh et al. solved the CVP by using LLL [33] lattice reduction and Babai's nearest plane algorithm [4] to recover Diffie-Hellman keys.

Different representations of the lattice exist [6, 38, 39]. To ensure that the closest vector reveals the private key x, the first d components of $\mathbf{t}$ and $\mathbf{u}$ are scaled by 2^{L_i+1}. Following [6], this gives a $d+1$-dimensional row-wise lattice basis B:

$$B = \begin{bmatrix} 2^{L_1+1}q & & & 0 \\ & \ddots & & \vdots \\ & & 2^{L_d+1}q & 0 \\ 2^{L_1+1}t_1 & \dots & 2^{L_d+1}t_d & 1 \end{bmatrix} \quad (12)$$

Instead of using Babai's nearest plane algorithm, it is also possible to embed the CVP into a Shortest Vector Problem (SVP) and solve it directly via lattice reduction [22, 40, 57]. The idea is to include the scaled vector $\mathbf{u}'$ in the lattice basis:

$$B' = \begin{bmatrix} B & 0 \\ \mathbf{u}' & q \end{bmatrix} \quad (13)$$

Boneh et al. [10] showed that this requires at least $L = \log_2 \log_2 q$ bit leakage. Howegrave-Graham and Smart [27] recovered the private key for 160-bit DSA given 30 signatures and knowledge of 8 bits for each nonce. Naccache et al. [37] only required 27 signatures for the same leakage using the block Korkin-Zolotarev (BKZ) algorithm. Given 200 signatures and two shared LSBs of the nonce, Faugère et al. [22] recovered the private key using a lattice attack. Besides, they recovered the private key with a probability of 90% with just a single shared LSB and 400 signatures.

Bleichenbacher. Bleichenbacher [9] proposed an FFT-based attack using exponential sums to detect influences of small biases. Compared to lattice attacks, this requires more samples but is noise-tolerant and works with small and even fractional bit leaks [35, 36]. Aranha et al. [3] exploited a single-bit nonce bias for 160-bit ECDSA using 2^{33} signatures. De Mulder et al. [35] used a BKZ-based method to exploit a 5-bit leakage of 384-bit ECDSA using 4000 signatures.

2.3 Side-Channel Attacks

Side-channel attacks allow breaking cryptographic implementations via unintended information leakage. They range from observing the overall execution time [30] to more fine-grained microarchitectural effects. Cache attacks target code accesses on a cache-line granularity via Flush+Reload [61] or data accesses via the more generic but coarse-grained Prime+Probe technique [42, 50]. In an SGX setting, powerful controlled-channel attacks [59] leak page accesses with high accuracy. In this work we consider address leakage, as a generalization of above side channels. Physical side channels are out of scope.

3 Related Work

3.1 Side-Channel Attacks

Modular Exponentiation. Square-and-multiply is a common technique for computing modular exponentiations and was targeted by Yarom and Falkner [61] in GnuPG. They extracted 97% of an RSA key from a single sign operation observed with Flush+Reload. Similarly, Prime+Probe attacks have been launched against GnuPG [34] and libgcrypt [64].

A faster alternative is the sliding window approach [12]. Percival [42] attacked OpenSSL's sliding window implementation by a technique that became known as Prime+Probe [50]. Similarly, the sliding window implementations of libgcrypt RSA [8] and GnuPG ElGamal [34] have been attacked.

Using fixed windows eradicates leakage due to conditional code execution in the sliding window approach. However, an implementation flaw in an earlier version of OpenSSL allowed bypassing the fixed window implementation [24].

To prevent leakage of the window multipliers, the scatter-gather technique aligns multipliers in memory such that the same cache lines are accessed all the time. Yarom et al. [62] exploited cache-bank conflicts to attack OpenSSL's scatter-gather implementation.

In the SGX setting, Prime+Probe attacks have been launched from malicious operating systems against the fixed-window exponentiation during the RSA decryption in the Intel IPP library [13]. Besides, Prime+Probe attacks have also been launched from one SGX enclave against another SGX enclave in order to extract an RSA key from mbedTLS [48].

ECDSA Scalar Multiplication. Brumley et al. [16] targeted constant-time double-and-add in OpenSSL ECDSA by measuring the total number of iterations. Yarom et al. [60] exploited conditional code during double-and-add via Flush+Reload, bypassing the constant-time implementation.

Brumley et al. [15] attacked the windowed Non-Adjacent Form (wNAF) multiplication of OpenSSL on the secp160

curve via Flush+Reload. Similar attacks on the popular secp256k1 curve leverage better side-channel observations and better recovery methods [2, 6, 21, 51]. Dall et al. [20] attacked a fixed-window scatter-gather version of Intel EPID by exploiting a leak in the number of iterations.

Modular Inversion. García and Brumley [23] attacked the binary extended Euclidean algorithm (BEEA) of OpenSSL via Flush+Reload, which was used for the modular inversion of the nonce k during ECDSA signature computations. Weiser et al. [54] mounted a controlled-channel attack against RSA key generation in OpenSSL by exploiting conditional branches in the binary Euclidean algorithm (BEA) used for checking co-primality of RSA parameters. Concurrently, Aldaya et al. [1] mounted a Flush+Reload attack on the vulnerable BEA implementation with a success rate of 28%.

Modular Reduction. Ryan [46] discovered an early abort condition in OpenSSL's modular reduction and exploited it with a Flush+Reload attack to recover ECDSA private keys.

3.2 Side-channel Analysis Tools

Due to the significant number of side-channel attacks, side-channel analysis frameworks have been developed. CacheAudit [31] uses symbolic execution to compute upper bounds on the possible leakage. However, these upper bounds could become imprecise, and analyzing large code bases such as OpenSSL with many potential leaks demands more practical approaches with high precision and low overhead.

Reparaz et al. [45] identify timing leaks with a black-box approach, which does not capture fine-grained cache attacks. ctgrind [32] tracks unsafe usage of secrets with the Valgrind memory error detector on annotated secrets. CacheD [53] taint-tracks instructions accessing secret data and evaluates them symbolically to find potential data leaks. CacheS [52] improves CacheD by using abstract interpretation and by finding secret-dependent branches. Zankl et al. [63] base their analysis on concrete instead of symbolic execution, which gives more precise results and better performance. They use binary instrumentation to build a histogram of all executed instructions and correlate it against the Hamming weight of the private key. Stacco [58] uses binary instrumentation to record instruction traces rather than histograms only and reveals padding oracle vulnerabilities. DATA [55] introduces the notion of more generic address traces, capturing instruction and data addresses. By matching address traces, it finds potential control-flow and data leaks. DATA also provides methods for distinguishing secret-dependent leaks from unrelated ones due to non-determinism (e.g., blinding), and it supports dedicated leakage models. MicroWalk [56] also records all accessed addresses but collapses the execution context, losing, e.g., call stack information in favor of faster analysis.

None of these approaches was designed or used to detect addresses leakage of (EC)DSA nonces. In this work, we adapt the idea of leakage models [55, 63] to detect nonce leakage.

3.3 Research Gap

To sum up, nonce leakage can occur in several (EC)DSA steps and can be exploited via efficient lattice attacks and more generic Bleichenbacher attacks. Despite extensive research, a systematic study of nonce leakage is still missing, and side-channel tools have not been tailored for nonce leakage.

We bridge this gap and provide the first systematic analysis of nonce leakage in popular crypto libraries. We extended the automated side-channel analysis tool DATA to also identify nonce leakage and visualize it in a GUI. By using this tool, we identify vulnerabilities in several computations involving the secret nonce, including Equations (3) to (6).

4 Automated Nonce Leakage Detection

Tool support is essential for effective and accurate side-channel analysis. We first discuss the open-source DATA framework [55], and introduce our threat model. Next, we discuss our changes to DATA, define proper leakage models for nonces and develop an intuitive GUI for visualizing results.

Original DATA Framework. DATA identifies address-based side-channel vulnerabilities through dynamic analysis in three phases. In the first phase, DATA collects address traces by instrumenting the target binary. By comparing those traces, it identifies address-based differences at byte granularity that indicate potential leaks. However, analyzing randomized (blinded) algorithms yields various address differences that do not leak secret information. Also, many differences stem from public input and are also uncritical. To filter these false positives, DATA employs statistical tests. The second phase tests if the differences depend on the private key by comparing traces generated from a fixed key with traces generated from varying keys. This fixed-vs-random testing requires control over the secret variable. Since nonces are not controllable from the outside but generated randomly (internally), this phase cannot be used for detecting nonce leakage. The third phase classifies information leakage based on a leakage model and detects linear and non-linear relations between address traces and a secret.

Threat Model and Limitations. DATA operates on address traces at byte granularity. This models a powerful side-channel attacker probing memory pages [59], cache lines [61], cache banks [62], or even single byte addresses which are currently only partially exploitable in specific settings [17]. However, as with any dynamic analysis, DATA cannot guarantee absence of leakage (e.g., it cannot prove code secure). Nevertheless, by increasing the number of traces and tested configurations, coverage increases (cf. [55]).

Leakage models correlate the observed leakage (*i.e.*, the address traces) with the secret. However, a high correlation does not necessarily imply actual leakage but could also stem from public values (e.g., the modulus). This is a fundamental issue of statistical testing and implies that an analyst should

always carefully review potential leakage reported by DATA, as we do in this work.

Speculative execution attacks are out of scope for this work, as they leverage data leakage rather than address leakage only.

Detecting Nonce Leakage. To tailor DATA for detecting nonce leakage, we bypass the second phase and make the third phase run independently. The third phase correlates leakage to a secret value via leakage models. However, secret nonces are generated internally and are not exposed to the outside. To overcome this limitation, we adapt DATA to recognize nonces as an additional secret in a backward manner. That is, we recover the nonce from the private key, the message, and the signature using Equation (6). Furthermore, we significantly improve the performance of phase three via multiprocessing. Finally, we introduce appropriate leakage models.

Leakage Models. Definition of proper leakage models is essential for finding nonce leaks. This, however, demands knowledge of potential leaks to search for. Based on initial manual inspection of OpenSSL's source code, we developed leakage models tailored for detecting nonce leakage. This was no straightforward process but involved extending the leakage models the more issues we found. In particular, we searched for Bignumber issues by testing the bit length of the nonce k and its variants $kinv$, $k+q$ and $k+2q$. This leakage model is denoted as *num_bits* and finds leakage, e.g., due to lazy resizing of Bignumbers. Furthermore, we used the Hamming weight model denoted as *hw* to search for leaks in DSA modular exponentiation (square-and-multiply) and ECDSA scalar multiplication (double-and-add), respectively. With these models, we were able to greatly reduce the number of unrelated differences. E.g., the leakage models typically filter well above 90% of the differences.

Semi-automated Analysis with GUI. While tool support does not make thorough side-channel analysis obsolete, we found it to be essential. Especially constant-time code can be reviewed much easier with tool assistance. Also, side-channel patches can be easily tested for their efficacy, preventing reintroduction of previously known leaks. In particular, a high degree of automation and a proper representation of results is imperative for productive analysis. Due to the nature of statistical testing used in DATA, an analyst should always carefully review leakage reports of DATA to rule out potential false positives and assess actual exploitability.

Since analyzing DATA reports is cumbersome, we developed a graphical user interface called DATA GUI. DATA GUI allows to quickly navigate leakage reports together with the source code and disassembly, and rate or comment potential leaks. For this to work, we extended DATA to generate an accompanying file archive that contains all necessary object files, disassemblies and source code files, alongside the regular leakage report. This also decouples the test phases of DATA from GUI-aided analysis, which now may be done on a completely different computer. Since we need to repeatedly test different cryptographic libraries under different configurations, the DATA GUI was key to master the amount of data we collected. We open-source our tool, including the DATA GUI and provide examples to reproduce our results. Figure 5 in Appendix A depicts the DATA GUI, showing a discovered control-flow leak in BoringSSL.

Table 1: Handling of secret nonces is either secure ○ or vulnerable ● to side channels, according to our analysis.

	Generate				Exp.		S. mul.		Invert		Mod. mul.		
	minimal rep.	rej. sampling	truncation	+ private key	k-padding	fixed window	k-pad + blind.	fixed window	Ext. Euclid	Little Fermat	Unprotected	Blinding	Const-time
OpenSSL	●		●	○	●	○	●	●	●			○	
LibreSSL	●	●			●	○	●		●		●		
BoringSSL		○				○		●		○			○

5 Vulnerability Analysis Overview

In this work, we analyze OpenSSL, LibreSSL, and BoringSSL for (EC)DSA nonce leakage. We include the whole life cycle of nonces in the analysis, *i.e.*, nonce generation, modular exponentiation for DSA or scalar multiplication for ECDSA, modular inversion, and the final modular multiplication. Our findings are summarized in Table 1 and outlined in the following. As mentioned in Section 4, our analysis cannot prove an implementation secure in a mathematical sense.

Nonce representation is based on Bignumbers. OpenSSL and LibreSSL minimize memory usage, *i.e.*, small numbers use fewer memory words than larger ones. This *minimal representation* of Bignumbers leaks the length of small nonces in several subsequent computation steps. BoringSSL, on the other hand, does not shrink sensitive Bignumbers, avoiding all Bignumber-related vulnerabilities we found by design.

Generation of nonces is done via rejection sampling in LibreSSL and BoringSSL, which gives uniformly distributed nonces. In contrast, OpenSSL truncates a large random number to the target nonce, introducing a negligible bias. Only OpenSSL includes the private key in the nonce generation to address potential weaknesses in random number generators.

DSA modular exponentiation itself did not reveal any leaks, as the fixed-window implementations are constant time. However, for OpenSSL and LibreSSL, we found several critical leaks due to padding the nonce *prior* to exponentiation. This enables easy-to-mount cache attacks, leading to full key recovery. Although the patched OpenSSL version closes the cache-attack vulnerability, it is still vulnerable to more sophisticated attacks, which we demonstrate in Appendix B.

ECDSA scalar multiplication leaks in OpenSSL and LibreSSL in the same way as DSA exponentiation, namely when padding the nonce. On the other hand, the default multiplication uses blinding to make side-channel leakage independent of the nonce. Additionally, OpenSSL and BoringSSL provide

Table 2: Discovered vulnerabilities in OpenSSL, LibreSSL, and BoringSSL and whether they are patched ✓as of October 2019, currently being patched 🔧, or unpatched ✗. Exploiting the side channel can be easy ●, medium ◐, or hard ○. The number of leaked bits (Nonce Leakage) indicates the complexity of a full key recovery.

	Vulnerability	OpenSSL	LibreSSL	BoringSSL	Nonce Leakage	SC	Comments
Generate:	(V1) Small k (top)	EC✗	EC✗	–	Topmost 0-limbs of k	●	Leaks in several subsequent steps
Exp. / Scalar Mult.	(V2) k-padding resize	DSA✓EC✓	DSA✗EC✗	–	Topmost 0-bits of k	●	CVE-2018-0734 and CVE-2018-0735
	(V3) consttime-swap	DSA✓EC✓	DSA✗EC✗	–	same as (V2)	◐	Already known
	(V4) Downgrade	DSA✓	–	–	same as (V2) + [24]	●	Introduced while fixing (V2)
	(V5) k-padding (top)	DSA✗EC✗	DSA✗EC✗	–	same as (V2)	○	Leaks in BN_add and BN_is_bit_set. SGX attack shown in Appendix B.
	(V6) Buffer conversion	EC✓	–	–	Topmost 0-bytes of k	○	
	(V7) Point addition	EC🔧	–	EC✓	All 0-windows of k	○	
Invert	(V8) Euclid BN_div	DSA✓	DSA✗	–	Topmost bit of k	●	Leaks via resize, similar to (V2)
	(V9) Euclid negation	DSA✓	DSA✗	–	Topmost 0-bit of *kinv*	●	Leaks via conditional negation
Multiply:	(V10) Small k^{-1} (top)	–	EC✓	–	Topmost 0-limbs of *kinv*	◐	

optimized constant-time windowed multiplication routines for several NIST curves. We discovered a tiny but severe side-channel leakage in their constant-time point addition, which leaks whenever a nonce multiplication window is all zero. For OpenSSL, we identified additional nonce leakage due to Bignumber handling, which was partly known before.

Modular inversion in OpenSSL and LibreSSL is done via a variant of Euclid's algorithm, claiming some side-channel security. Nevertheless, we found an easy-to-exploit vulnerability leaking the topmost nonce bit during a division step. Moreover, Euclid's algorithm inherently leaks the number of iterations, which correlates to the nonce itself. While we could not find a way to exploit this non-constant time behavior, our tool reported another leak in a final negation step that helps an attacker again to learn the topmost nonce bit. BoringSSL employs Fermat's little theorem to invert nonces securely. Due to our findings, OpenSSL also switched to Fermat inversion.

Modular Multiplication. While OpenSSL uses blinding to alleviate non-constant time code, LibreSSL removes blinding too early, leaking the length of the inverse nonce.

6 Detailed Analysis

In the following, we present our analysis methodology and discuss results and discovered vulnerabilities in detail.

Analysis Methodology. The process of tool-aided side-channel analysis comprises a proper selection of algorithms to test, the actual analysis phase and an interpretation of the results. Since OpenSSL supports over 80 different elliptic curves and countless compiler options, exhaustive testing of each combination is impractical. We selected the default configuration as a basis for our analysis, and selectively enabled different implementations of popular NIST curves. We tested all three DSA parameter sets and focused on ECDSA curves operating close to a machine word boundary. For the actual analysis, we used our tool alongside manual code review to specifically test relevant portions in the code. While the tool helps uncover leakage, interpreting the results remains a manual task. In particular, leakage models might not trigger if they do not match the actual leakage. In this case, leakage might still show up in the *phase one* differences reported by DATA, and an extension of the leakage models is required. Also, leakage models might show a correlation without causation, e.g., via public values. Such cases can be eliminated by tracing the leakage back to its sources in our DATA GUI.

Following this methodology helped us uncover numerous vulnerabilities, as summarized in Table 2. To give an intuition about their exploitability, we rank them as easy to exploit ● if a Flush+Reload attack suffices for extracting nonce bits, medium ◐ for more elaborate attacks requiring performance degradation or Prime+Probe, or hard ○ for tiny leakage (e.g., few assembler instructions on a single cache line) which might be only exploitable in an SGX setting [17].

6.1 Nonce Representation

OpenSSL and LibreSSL represent cryptographic values such as nonces via Bignumbers. Each Bignumber is stored in a `BIGNUM` struct that contains a lazily allocated array of limbs (e.g., 64-bit words). The number of allocated limbs is tracked via the field `dmax`. Bignumbers are represented in their minimal form, *i.e.*, each `BIGNUM` tracks the actually used limbs in a separate `top` field. As seen in Figure 1, `top` can be smaller than `dmax`. Whenever space is exhausted, a `BIGNUM` is dynamically resized via a call to `bn_wexpand`.

To maintain the minimal representation, OpenSSL and LibreSSL constantly realign `top` via a call to `bn_fix_top` by excluding leading zero limbs. This has two advantages: First, it avoids unnecessary computations and increases performance. Second, the programmer does not need to know the maximum size of Bignumbers in advance. However, it is also a source for side-channel leakage, leading to various vulnerabilities.

BoringSSL, in contrast, has hardened their implementation against such leaks by abandoning the minimal representation invariant of Bignumbers. They introduced a `width` field,

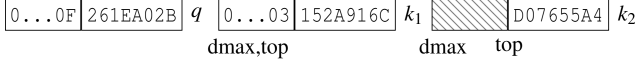

Figure 1: OpenSSL/LibreSSL (V1): some nonces (k_2) are smaller than the average (k_1) and the modulus q.

Table 3: OpenSSL/LibreSSL curves leaking L bits of small (inverse) nonces (V1),(V10) on 32/64-bit systems.

Curve	L_{32}	L_{64}	Curve	L_{32}	Curve	L_{32}
secp112r1	15.8	–	sect163r1	2.0	c2tnb359v1	0.8
secp112r2	13.8	–	sect163r2	2.0	c2tnb431r1	1.7
secp521r1	9.0	**9.0**	sect233k1	7.0	wap-wtls1	16.0
prime239v1	15.0	–	sect233r1	8.0	wap-wtls3	2.0
prime239v2	15.0	–	sect239k1	13.0	wap-wtls4	16.0
prime239v3	15.0	–	c2pnb163v1	2.0	wap-wtls5	2.0
sect113r1	16.0	–	c2pnb163v2	2.0	wap-wtls6	15.8
sect113r2	16.0	–	c2pnb163v3	2.0	wap-wtls8	16.0
sect131r1	2.0	**2.0**	c2tnb239v1	13.0	wap-wtls10	7.0
sect131r2	2.0	**2.0**	c2tnb239v2	12.4	wap-wtls11	8.0
sect163k1	2.0	–	c2tnb239v3	11.7		

which fixes top to the maximum width in advance.[2] Hence, it is immune to the Bignumber-related leaks we found.

Small Nonce Vulnerability (V1). Nonces are generated in the range $[1, q-1]$. If the length of the modulus q is slightly above a word boundary, it may happen that the generated nonce uses fewer limbs than q. In Figure 1, the first nonce k_1 uses two limbs, whereas the second nonce k_2 is represented in one limb, as indicated by top. A side-channel attacker learning the value of top can distinguish small nonces from large ones and mount a key recovery attack.

In this example, q uses only four bits (0xF) of the topmost limb. Thus, an attacker learns whether the four topmost bits of k are zero. Consider w as the word size, *i.e.*, the size of one limb. For i386, w=32 and for x86_64, w=64. Thus, a small nonce leaks $L = log_2(q) \mod w$ bits, which occurs every 2^Lth signature on average. By collecting enough leaky signatures, an attacker can recover the private key via lattice or Bleichenbacher attacks (see Section 2.2).

In general, both DSA and ECDSA are affected by small nonces. However, if L is too large, leaky signatures occur too rarely to be practically exploitable. Since DSA moduli are always (half)word-aligned, $L = 32$ or $L = 64$ and attacks are impractical. On the other hand, for ECDSA, several curves have a modulus (group order) that is slightly above a word boundary. Table 3 lists all affected curves with $L < 20$, and curves affected on 64-bit systems are marked bold. For example, the sect131 curves leak 2 bits approximately every 4th signature, while secp521r1 leaks 9 bits every 512th signature.

In order to exploit the small nonce vulnerability, an attacker needs to learn the nonce length (*i.e.*, the value of top). Since the nonce is involved in many different computation steps, there are plenty of opportunities for an attacker to observe its length. We found leakage in the nonce generation, scalar multiplication, and nonce inversion (Equations (3), (5) and (7)). Details for OpenSSL and LibreSSL can be looked up in Appendix C. In the following, we focus on the most critical leakage present in the OpenSSL version patched against (V8). The leaky code in Listing 1 converts the nonce stored in BIGNUM a into its Montgomery representation. BIGNUM b holds a Montgomery conversion factor. If both, a and b have the full word length of q, denoted as num, the if branch will execute an assembler-optimized multiplication (bn_mul_mont in line 4) and terminate in line 5. If, however, the nonce a is one limb smaller, OpenSSL falls back to the functions bn_mul_fixed_top and bn_from_montgomery_word. By probing any of those functions, e.g., with Flush+Reload, an attacker can distinguish small nonces from larger ones.

Unfortunately, this vulnerability is not only easy to exploit, but patching is hard as small nonces leak in several places. On June 25, 2019, we reported this issue to OpenSSL, who decided to target a fix in OpenSSL version 3.0, as it requires a major redesign of OpenSSL's Bignumber implementation.

6.2 Nonce Generation

In the following, we analyze nonce generation for different libraries under the default configuration. DSA and ECDSA nonces are generated both in the same way.

Rejection Sampling. To generate a nonce k uniformly at random in the interval $[1, q-1]$, LibreSSL and BoringSSL implement rejection sampling. They sample k in the interval $[1, 2^{qbits}-1]$, where $qbits = \lfloor \log_2 q \rfloor + 1$. If k exceeds $q-1$, it is rejected, and the procedure is repeated. The final k is uniformly distributed, assuming an unbiased random number generator. Although rejection sampling is inherently non-constant time, it only leaks information about rejected nonces. While we did not find issues for BoringSSL, small nonces leak for LibreSSL, as detailed in Appendix C.

Truncation. OpenSSL first generates a large number k' in the interval $[0, 2^{qbits+64}-1]$, as seen in Algorithm 1 lines 2–6. To compute the final nonce, k' is truncated to the target interval $[0, q-1]$ via modular reduction (line 8). As with LibreSSL, small nonces leak during truncation, as detailed in

```
1  if (a->top == num && b->top == num) {
2      if (bn_wexpand(r, num) == NULL)
3          return 0;
4      if (bn_mul_mont(...))
5          return 1;
6  }
7  ...
8  if (!bn_mul_fixed_top(tmp, a, b, ctx))
9      goto err;
10 if (!bn_from_montgomery_word(r, tmp, mont))
11     goto err;
```

Listing 1: Simplified OpenSSL Little Fermat inversion leaking small nonces (V1) via conditional branching.

[2] https://github.com/openssl/openssl/issues/6640

Algorithm 1: OpenSSL nonce generation by truncation

input : x, q `// Private key and modulus`
input : m `// Message digest`
output : k `// Nonce`

1. $k' \leftarrow []$
2. **while** $num_bits(k') < num_bits(q) + 64$ **do**
3. $rnd \xleftarrow{R} [0, 2^{512} - 1]$
4. $digest \leftarrow SHA512(x|m|rnd)$
5. $k'.append(digest)$ `// Up to` $num_bits(q) + 64$ `bits`
6. **end**
7. $k'' \leftarrow BN_bin2bn(k')$ `// Convert to BIGNUM`
8. $k \leftarrow k'' \mod q$ `// Reduce via BN_div`

Appendix C. Moreover, truncation introduces a tiny bias in k since q does not exactly divide $2^{qbits+64}$. However, since k' is 64 bits larger than q, this bias is impractical to exploit.

Before reducing k', OpenSSL converts it to a Bignumber representation via `BN_bin2bn` in line 7, which introduces a tiny side-channel leakage on k'. In particular, `BN_bin2bn` removes leading zeros, leaking the byte length of k' to a side-channel attacker. Our tool revealed another leakage in `BN_div` called in line 8, leaking the length of k'. Luckily, both issues are impractical to exploit due to the 64-bit margin of k'.

Private Key Inclusion. Biases in the nonce generation are fatal. For that reason, some variants of (EC)DSA [28,44] compute the nonce deterministically from the message via hash functions rather than using randomness. Similarly, OpenSSL uses the private key as additional input for nonce generation.[3] By applying a cryptographic hash function to the random number, the message m and the private key x (Algorithm 1 line 4), the resulting nonce is unpredictable to an attacker, even for biased random numbers. Moreover, this approach also protects against side-channel leaks. We found that OpenSSL uses a leaky AES[4] during random number generation when compiled with the `no-asm` flag. The hash in line 4 decorrelates these leaks from the nonce. BoringSSL and LibreSSL do not include the private key in the nonce computation, which makes them susceptible to biased random number generators. However, we did not analyze the uniformity or unpredictability of the random number generators themselves.

6.3 DSA Exponentiation

K-padding Vulnerabilities (V2)-(V5). Bignumber computation has been a source for nonce leakage in the past. For example, the fixed window exponentiation of OpenSSL leaks the bit length of the secret exponent k (Algorithm 2 line 5). This leakage was fixed by padding nonce k with q until it has a fixed length $num_bits(q) + 1$, as shown in Algorithm 2

[3] This change was introduced in OpenSSL commit `8a99cb2` in 2013.
[4] It leaks several intermediate values via lookup tables `Te0 - Te3`.

Algorithm 2: Exponentiation with k-padding

input : k `// Nonce`
output : r `// Signature part`

1. $k \leftarrow k + q$ `// Expand` k `to fixed` $num_bits(q) + 1$
2. **if** $num_bits(k) <= num_bits(q)$ **then**
3. $k \leftarrow k + q$
4. **end**
5. $r \leftarrow g^k \mod q$

```
1   q_bits = BN_num_bits(dsa->q);
2  -if (!BN_set_bit(k, q_bits)
3  -     || !BN_set_bit(l, q_bits)
4  -     || !BN_set_bit(m, q_bits))
5  + q_words = bn_get_top(dsa->q);
6  +if (!bn_wexpand(k, q_words + 2)
7  +     || !bn_wexpand(l, q_words + 2))
8        goto err;
9   ...
10  BN_set_flags(k, BN_FLG_CONSTTIME);
11 +BN_set_flags(l, BN_FLG_CONSTTIME);
12  ...
13  if (!BN_add(l, k, dsa->q)
14 -     || !BN_add(m, l, dsa->q)
15 -     || !BN_copy(k, BN_num_bits(l) > q_bits ? l : m))
16 +     || !BN_add(k, l, dsa->q)
17       goto err;
18 +BN_consttime_swap(BN_is_bit_set(l, q_bits), k, l,...);
```

Listing 2: Vulnerable k-padding in OpenSSL, with code added (+) and removed (-) during patching.

lines 1–3. The initial k-padding[5] executed the second addition in line 3 conditionally. To prevent attacking this conditional execution, it was made constant-time.[6] As shown in Listing 2, lines 13–14 unconditionally compute both additions inside `BIGNUMs` `l` and `m`, while line 15 copies the correct result to `k`.

By analyzing OpenSSL, we found that k-padding leaks in several ways. First, we discovered an easy-to-exploit vulnerability leaking the size of the nonce via `dmax` inside the second `BN_add` (Listing 2 line 14). This leakage denoted as (V2) allows full key recovery. Second, our tool also reported data leakage in line 15, already known before and denoted (V3). By distinguishing whether buffer `k` or `l` is copied, one learns the same information as before. Third, we found the same information leaking via the nonce's `top` variable, denoted (V5). This leakage exists in all patched versions and occurs when `k` is processed in lines 16 and 18. Although harder to exploit, we show an end-to-end attack in an SGX setting in Appendix B.

K-padding Resize Vulnerability (V2). As mentioned before, OpenSSL lazily resizes Bignumbers whenever their space is exhausted. E.g., when adding two `BIGNUMs` with `BN_add`, the result `BIGNUM` is expanded to the largest `top` value of the summands plus one limb for a potential carry. Unfortunately, lazy resizing happens during nonce padding in lines 13 and

[5] Nonce padding was introduced in OpenSSL commit `0ebfcc8` in 2005.
[6] Constant-time padding was introduced in OpenSSL commit `c0caa94`.

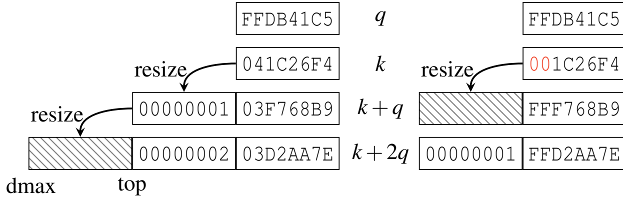

Figure 2: OpenSSL/LibreSSL k-padding causes Bignumber resize, depending on the topmost nonce bits (V2).

14 of Listing 2. Consider the example in Figure 2, where the BIGNUMs k and q contain one limb each. On the left side, the first addition $k+q$ resizes the result buffer to two limbs in order to hold the additional carry exceeding the first limb. The second addition $k+2q$ resizes to three limbs, although only two limbs are actually used since the carry is zero. In contrast, on the right-hand side, the first addition does not overflow, and the second addition only requests two limbs. Since the result BIGNUM already has two limbs, no actual resize happens.

By distinguishing whether one or two resize operations happen, a side-channel attacker can learn information about k. The second resize only happens if the first addition overflows into the carry limb. In practice, such an overflow can only happen if q is close to a word boundary, that is, the topmost bits are set. Again, consider w as word size. Then, $Q = \lfloor log_{2^w}(q) \rfloor + 1$ is the number of words needed to represent q, and $qbound = (2^w)^Q > q$ is the upper bound (exclusive) of q representable with Q words. No resize happens if $k+q < qbound$, which occurs with probability $(qbound - q)/q$. Thus, for each such situation, an attacker can learn L nonce bits at once:

$$L = log_2(q) - log_2(qbound - q) \tag{14}$$

Since k is chosen uniformly at random, this happens for approximately every 2^Lth signature. In the previous example, $qbound = 0x1\,0000\,0000$ and $q = 0xFFDB41C5$, hence an attacker can learn $L = 10.8$ nonce bits for one out of 1783 signatures on average. By collecting enough leaky signatures, an attacker can recover the private key, as shown in Section 2.2.

Only DSA moduli close to the word boundary are susceptible. OpenSSL supports DSA moduli in the ranges 160, 224 or 256 bits, respectively. Since these parameters are all at a 32-bit boundary, they are all susceptible on a 32-bit system. For 64-bit systems, only DSA with 256-bit is on a word boundary and, thus, susceptible. The modulus q is a prime generated randomly for each key with its topmost bit set. Hence, every 2^Lth key is susceptible to $L+1$-bit nonce leakage.

Exploitation of the vulnerability is straight forward. An attacker needs to monitor Bignumber resize operations during k-padding. Each Bignumber resize triggers several nested allocation routines of OpenSSL, which in turn invoke malloc/realloc from the standard library. Hence, a Flush+Reload attacker has plenty of opportunities to observe a resize with little noise. This attack is practical in terms of easy-to-obtain side-channel observations and low complexity for key recovery, which caused OpenSSL to issue CVE-2018-0734.

Consttime-swap Vulnerability (V3). Our tool showed another k-padding issue, which was already documented in the source code comments. After the two additions, copying the correct result to the target Bignumber `k` accesses different Bignumbers `l` or `m`, as shown in Listing 2 line 15. This leaks the same information as (V2) and could be exploited via a Prime+Probe attack on the Bignumber `l` or `m`, respectively.

Patching (V2) and (V3). Our reports triggered immediate discussion and patching[7] by the OpenSSL team. To avoid lazy reallocation, the patch enlarges the preallocation of the nonce buffers (lines 6–7). To hold the padded nonce, one additional limb would suffice. Since `BN_add` allocates an additional carry limb, this totals two additional limbs to preallocate. To fix the consttime issue, the patch replaces Bignumber `m` with `k` in line 16 and introduces `BN_consttime_swap` in line 18.

LibreSSL adopted similar patches for ECDSA, but insufficiently, as explained in Section 6.4. We contacted LibreSSL on May 17, 2019, but they did not apply these patches to DSA.

Downgrade Vulnerability (V4). By analyzing the OpenSSL patches for (V2) and (V3) with our tool, we immediately recognized a flaw bypassing constant-time exponentiation. While Bignumber `k` has the flag `BN_FLG_CONSTTIME` set, Bignumber `l` has not. The consttime-swap introduced in Listing 2 line 18 also swaps these flags between `l` and `k`, making `k` lose its flag. This causes every other subsequent exponentiation (Equation (4)) to downgrade to the unprotected variant. As shown in [24], this can be exploited to recover DSA keys from OpenSSH handshakes. Erroneous flag propagation has a long history, since manual detection within the complex code base of OpenSSL is non-trivial. Luckily, our systematic tool-aided approach uncovered this issue straight away, avoiding another exploit-patch cycle. The final patch[8] applies the `BN_FLG_CONSTTIME` flag also to the Bignumber `l` in line 11.

K-padding Top Vulnerability (V5). Fixing the resize vulnerability (V2) does not mitigate the Bignumber minimal representation issue. That is, even if the buffer size (`dmax`) is independent of `k`, the number of used limbs (`top`) still depends on the nonce (cf. Figure 2). In particular, the second addition `BN_add` in Listing 2 line 14 leaks the value of `l->top` via the number of limb-wise additions carried out. Also, `BN_is_bit_set` (line 18) leaks via an early abort, as detailed in Appendix B. This has the same implications as (V2).

Naturally, exploitation is harder than (V2), as the leaky code is only a few instructions. Nevertheless, we reported this residual leakage already back in October 2018. Since we could not observe any progress, we developed an end-to-end SGX attack, as outlined in Appendix B. Reporting our attack on May 8, 2019 triggered a pull request with our proposed patch [19]. However, the pull request was closed, since the OpenSSL team decided for a long-term mitigation

[7]See OpenSSL commit `a9cfb8c`.

[8]See OpenSSL commit `00496b6`.

Table 4: OpenSSL/LibreSSL curves leaking L nonce bits via k-padding (V2)–(V5) on 32-bit and 64-bit systems.

Curve	L_{32}	L_{64}	Curve	L_{32}	L_{64}
brainpoolP160	3.4	–	brainpoolP320	2.2	2.2
brainpoolP192	1.7	1.7	brainpoolP384	0.3	0.3
brainpoolP224	2.4	–	brainpoolP512	1.0	1.0
brainpoolP256	1.0	1.0			

abandoning the minimal representation invariant similar to BoringSSL [7]. While the decision for a complete fix is encouraging, this vulnerability remains unpatched until then.

6.4 ECDSA Scalar Multiplication

K-padding Resize Vulnerability (V2). Similar to DSA, our investigations revealed the same Bignumber resize vulnerability also in ECDSA, leading to CVE-2018-0735. Only curves with a word-aligned modulus (*i.e.*, the curve cardinality) are vulnerable. We found that all Brainpool curves are exploitable and leak up to 3.4 bits, as listed in Table 4. Luckily, other curves have a word-aligned modulus but are not practically exploitable. For example, the curve secp128r1 has cardinality 0xFFFFFFFD FFFFFFFE F80091C8 184ED68C. By using Equation (14), an attacker could learn $L = 31$ nonce bits at once. However, only every 2^{31}th signature will be vulnerable, which renders actual attacks impractical.

Fixing this issue for ECDSA is analogous to DSA.[9] Although LibreSSL adopted the patch,[10] our tool still reported leakage. Further analysis revealed that the patched LibreSSL version uses k-padding twice, once correctly during multiplication `ec_GFp_simple_mul_ct` and a second time inside `ecdsa_sign_setup`. The second k-padding was not only unpatched, leading to another instance of (V2), it even created additional leakage. In particular, the multiplication routine performs an additional leaky modular reduction if the nonce (the scalar) is larger than the group order. This again highlights the importance of tool-aided side-analysis during the patching process. Although we reported this issue to LibreSSL on May 20, 2019, it is still unpatched.

Issues (V3), (V5). As with DSA, the issues with consttime swap (V3) and k-paddding top (V5) as well as their patches equally apply to ECDSA for the curves listed in Table 4. Since the patched LibreSSL uses k-padding twice for ECDSA, it is still vulnerable not only to (V2) but also to (V3).

Buffer Conversion (V6). We uncovered distinct vulnerabilities in some ECDSA scalar multiplication routines of OpenSSL[11] leaking the byte length of the nonce. Before the actual scalar multiplication, the nonce is converted from a Bignumber to a byte array with `BN_bn2bin` and `flip_endian`. In contrast to Bignumber-related issues subject to word-granular leakage, those functions operate on bytes. By stripping leading zero bytes, they leak the byte length of a nonce. For secp224r1 and secp256k1, $L = 8$ bits leak every 256th signature, and $L = 16$ bits every 65536th signature. secp521r1 is not byte aligned and leaks $L = 1$ bit every 2nd signature, or $L = 9$ bits every 512th signature, etc. Since the side channel only comprises a few instructions and data bytes, we rate it as hard to exploit. Yet, an SGX attack similar to Appendix B could target the stripped nonce buffer. This issue was patched on August 3, 2019.[12]

```
1 if (x_equal && y_equal && !z1_is_zero && !z2_is_zero)
2     point_double(...)
```

Listing 3: Simplified excerpt from vulnerable point_add (V7) in OpenSSL/BoringSSL scalar multiplication.

Point Addition Vulnerability (V7). For ECDSA signatures, the nonce k is multiplied with the generator G in Equation (7). Analyzing OpenSSL and BoringSSL showed that the constant-time scalar multiplication uses a non-constant-time point addition. This leaks nonce windows consisting of zeros. We uncovered this leakage with our tool showing 100% correlation on the bit length of k, as shown in Appendix A Figure 5.

For the multiplication, the scalar is split into multiple fixed-size windows. Each window is used as an index into a pre-computed table to select the point to be added. If the window is all-zero, the first point is selected from the table. This first point represents infinity and has all-zero coordinates. Point addition has a special doubling case in Listing 3 line 2. Although doubling itself is never performed, the check in line 1 reveals whether the added point is infinity or not. Hence, an attacker can learn whether the current nonce window is zero. With a window size of w bits, roughly 2^{-w}th of the nonce is leaked per sign operation. E.g., for the common window size of 5, around 3.2% of the nonce is leaked.

The leak occurs due to the order in which the branching condition is evaluated. The `if` in line 1 consists of four separate conditions, which are compiled into multiple compare and jump instructions (cf. Figure 5 in Appendix A). This creates a tiny leakage because a different number of instructions are executed, depending on the secret scalar. When the added point is not infinity, already the first comparison (`x_equal`) fails, since the added points are unequal. If the added point is infinity, this causes the flags `x_equal` and `y_equal` to be `true`. This is because infinity is represented with all-zero projective (x,y,z) coordinates. Only the last flag `!z2_is_zero` fails, which results in a few more executed instructions. Exploiting this leakage with a cache attack seems infeasible due to the tiny difference in the executed code. However, in an SGX setting, [17] could be used to single-step instructions.

[9] See OpenSSL commit 99540ec.

[10] See LibreSSL commit 34b4fb9.

[11] This applies to the optimized NIST curve implementations, which are obtained via the `enable-ec_nistp_64_gcc_128` compilation flag.

[12] See https://github.com/openssl/openssl/pull/9511 as well as commits 8b44198b and 805315d3

Table 5: Curves vulnerable (●) to ECDSA point addition leak (V7) in constant-time scalar multiplication for base point (BP) or arbitrary point (AP).

	Curve	BP	AP	Compile configuration
OpenSSL	secp224r1	○	●	`enable-ec_nistp_64_gcc_128`
	secp256k1	○	●	
	secp256k1	●	●	`enable-ec_nistp_64_gcc_128 no-asm`
	secp521r1	●	●	`enable-ec_nistp_64_gcc_128`
BoringSSL	secp224r1	●	●	`OPENSSL_SMALL`
	secp256k1	○	●	
	secp384r1	●	●	
	secp521r1	●	●	

We systematically analyzed various point multiplication implementations and list affected ones in Table 5. Base point multiplication with precomputed lookup tables is used in ECDSA, whereas arbitrary point multiplication is used in ECDH. In OpenSSL, only optimized NIST implementations are affected. Other configurations and curve settings are unaffected because they use a blinded double-and-add implementation. In BoringSSL, all curves are vulnerable at least under one configuration. Since LibreSSL only uses blinded double-and-add for scalar multiplication, it is also unaffected.

Our report led to an immediate fix[13] by BoringSSL, which replaces the evaluation of the branching condition with bitwise operations, such that a short-circuit evaluation is no longer possible. OpenSSL is currently in the process of patching[14], since our responsible disclosure on May 31, 2019.

6.5 Modular Inversion

Euclid BN_div (V8). OpenSSL and LibreSSL implement modular inversion via the Extended Euclidean algorithm. In contrast to the binary extended Euclidean algorithm (BEEA), which is known to be vulnerable [1, 23, 54], the inversion used for DSA is denoted as constant-time in the source code. With our tool, we uncovered a leak hidden deeply in this constant-time modular inversion of OpenSSL. In particular, the first Euclidean iteration leaks the topmost nonce bit of every signature to a side-channel attacker.

Since DATA accumulates leakage not only over the first but over all Euclidean iterations, our leakage models did not show high correlation. Instead, we found this leak by carefully analyzing the differences reported by the first phase of DATA.

Algorithm 3 shows the leaky Extended Euclidean inversion. The division `BN_div` in line 3 is not constant time, although the `BN_FLG_CONSTTIME` flag is used. Note that `BN_div` computes both, the integer division D and the remainder M. In the first iteration, `A` holds the public modulus `q`, and `B` holds the secret nonce k. Inside `BN_div` the `BIGNUM`s are aligned before the actual division, as follows. The divisor (nonce k) is shifted to the left such that its highest word is filled, having no leading

[13] See BoringSSL commit `12d9ed6`.

[14] `https://github.com/openssl/openssl/pull/9239`

Algorithm 3: OpenSSL/LibreSSL leaky inversion

input : a, n
output: inv `// Inverse of a mod n`
1. $(A,B,X,Y,sign) \leftarrow (a,n,1,0,-1)$
2. **while** $B > 0$ **do**
3. $\quad (D,M) \leftarrow (A/B, A\%B)$ `// Leaky division (V8)`
4. $\quad (A,B) \leftarrow (B,M)$
5. $\quad (X,Y) \leftarrow (D \cdot X + Y, X)$
6. $\quad sign \leftarrow -sign$
7. **end**
8. **ensure** $A = 1$
9. **if** $sign < 0$ **then**
10. $\quad Y \leftarrow n - Y$ `// Leaky negation (V9)`
11. **end**
12. $inv \leftarrow Y \mod n$

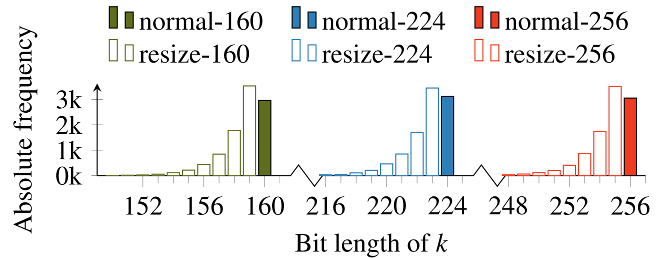

Figure 3: OpenSSL DSA leaks the topmost bit of the nonce during Euclidean inversion (V8).

zero bits. The numerator (modulus) is shifted left by the same amount of bits (modulo the word size). Normally, the nonce has the same bit length as the modulus, and the numerator also gets word-aligned. If the nonce, however, has fewer bits than the modulus, this shift operation causes the numerator `BIGNUM` to spill over to the next limb, and `top` is incremented. This will cause a `BIGNUM` resize operation. Observing such resize operations allows an attacker to distinguish nonces whose most significant bit is cleared.

To evaluate the leakage further, we generated 100 DSA keys and computed 100 DSA signatures per key. Figure 3 plots the resulting bit length of k for each of the common DSA settings $qbits \in \{160, 224, 256\}$. There is a clear separation between nonces with a zero MSB causing a resize, and "normal" nonces whose topmost bit is set. Since the modulus is chosen randomly per key, the probability of having the MSB of the nonce set is only around 30%, whereas the probability of a zero MSB is around 70%. Thus, an attacker can effectively learn approximately 0.88 bits[15] for each signature.

To exploit the vulnerability, an attacker probes for leaky resize operations in `BN_div` during the first Euclidean inversion. A simple Flush+Reload attack on the corresponding Bignum-

[15] Computed via the information entropy

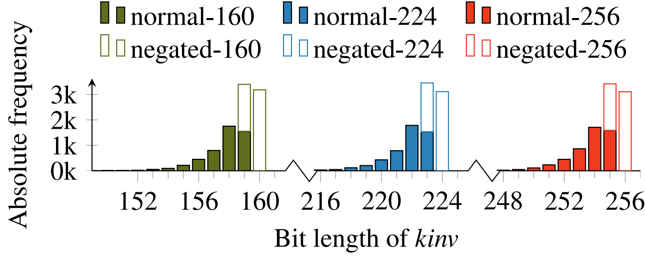

Figure 4: OpenSSL DSA leaks the topmost bit of the inverse nonce after Euclidean inversion (V9).

ber allocation routines suffices, as with (V2). Since $L = 1$, a Bleichenbacher attack is needed to recover the private key.

We proposed to abandon Euclid inversion in favor of a safer method. One could either use blinding to decorrelate side-channel leakage from the nonce, or use Fermat's little theorem, as done by BoringSSL. OpenSSL decided to implement Fermat's little theorem[16] by computing $kinv = k^{q-2} \mod q$. Although we reported this vulnerability also to LibreSSL on May 17, 2019, they did not apply the patch.

Euclid Negation (V9). The Euclidean algorithm is inherently non-constant-time and leaks the number of iterations. We initially tried to correlate the number of iterations to the nonce length. By doing simulations, we found that the iterations fluctuate significantly, and cannot be used as a reliable side channel for learning the nonce length. However, when applying our automated statistical methods, our tool reported a significant correlation on the bit length of the *inverse* nonce *kinv*. In particular, the Euclidean algorithm keeps track of the inverse's sign bit and conditionally negates Y in the end as shown in Algorithm 3 line 10. We found that negation causes larger inverses on average, presenting a useful side-channel.

To visualize the leakage, we repeat the experiment from (V8). Figure 4 plots the bit length of inverse nonces *kinv*. In our experiments, negation gives a large *kinv*; however, the topmost bit is not necessarily one ($num_bits(kinv) \geq qbits - 1$). In contrast, "normal" inversion without negation causes the MSB of *kinv* to be zero, which happens in around 70% of the cases, giving 0.88 bits of leakage per signature.

This vulnerability can be exploited via a Flush+Reload attack on the leaky `BN_sub` function and only collecting signatures where no negation happens. A Bleichenbacher attack can be used to recover the actual private key.

The patch introduced in (V8) also fixes this vulnerability. LibreSSL remains vulnerable, as they did not apply this patch.

6.6 Modular Multiplication (V10)

As Bignumber primitives are not constant-time in several places [46], OpenSSL blinds[17] the actual computation of the signature s in Equation (6) to avoid leaking the private key x. This works by applying a random blinding value b, as follows:

$$b \xleftarrow{R} [1, q-1] \quad (15) \qquad s \leftarrow s \cdot k^{-1} \mod q \quad (17)$$

$$s \leftarrow (bm + bxr) \mod q \quad (16) \qquad s \leftarrow s \cdot b^{-1} \mod q \quad (18)$$

This makes leakage during addition, modular reduction, and multiplication in Equation (16) independent of the private key as well as the inverse nonce in Equation (17). Unfortunately, when LibreSSL applied the patch,[18] they swapped Equation (17) and Equation (18), causing the multiplication with *kinv* to be unprotected. In particular, the routine `BN_mul` leaks the value of `kinv->top` at various locations.

This vulnerability is conceptually the same as the small nonce vulnerability (V1), affecting the same curves listed in Table 3. It leaks whether the inverse nonce is one limb smaller than the modulus. Since leakage of the inverse nonce is equally dangerous as leakage of the nonce itself, an attacker can mount the same key recovery attack as for (V1). In response to our disclosure, LibreSSL fixed this issue.[19]

7 Evaluation

Having detailed all vulnerabilities, we now evaluate our analysis methodology as well as the leakage models.

Analysis Methodology. Investigating the leakage reports of DATA represents a chicken-and-egg problem. The results of DATA phase one cover all discovered differences (*i.e.*, potential leaks), but are tedious to analyze. Developing precise leakage models to filter those results requires an intuition about the nature of leakage, which in turn demands some manual analysis of phase one results. As described in Section 6, we concurrently followed both approaches. By manually analyzing phase one results, we gained an understanding of the libraries. Although we found vulnerabilities related to k-padding as well as (V8) that way, this task is tedious. Thus, we derived the leakage model *num_bits* which captures the bit length of k, $k+q$ and $k+2q$ to detect k-padding leaks automatically. We used the gained knowledge to search for other Bignumber-related leaks, and also included inverse nonces *kinv* in our models. Our leakage models confirmed initial results and helped us discover more Bignumber-related vulnerabilities such as (V1), (V9) and (V10). Moreover, since *num_bits* correlates with the bit length rather than the word length of the nonce, we also found leakage on a byte granularity (V6) and window granularity (V7).

The choice of library configurations and algorithm parameters is essential. E.g., we realized that (V2) does not show up for DSA-160 on a 64-bit system, while 32-bit systems leak for all parameter sets. Also, the choice of the modulus `q` is essential in causing leakage to show up. In order to confirm (V2) also for ECDSA, we analyzed all ECDSA moduli offline and

[16] The patch was introduced in OpenSSL commit `415c335`.

[17] Blinding was introduced via OpenSSL commit `7f9822a`.

[18] See LibreSSL commits `2cd28f9` and `2a937ef`.

[19] See LibreSSL commits `1f6b35b` and `159fbd1`.

Table 6: Evaluation of leakage models. Depending on the triggered vulnerabilities, differences (Diffs) found by DATA are filtered via our leakage models. The overall reduction is computed when filtering almost non-matching leaks (<1%), somewhat matching leaks (<50%), or all leaks except for perfect correlation (<100%).

Tested configuration	Vulnerabilities	Diffs	Leakage model (max. correlation)								Overall reduction Diffs vs. Leaks		
			num_bits				*hw*						
			k	$k+q$	$k+2q$	*kinv*	k	$k+q$	$k+2q$	*kinv*	<1%	<50%	<100%
LibreSSL sect131r1	(V1),(V9),(V10)	1450	**100.0%**	0.0%	0.0%	**100.0%**	7.4%	18.0%	9.4%	10.0%	90.2%	97.9%	99.0%
OpenSSL DSA-256	(V2),(V5),(V8),(V9)	663	**100.0%**	**100.0%**	**100.0%**	79.8%	0.0%	2.7%	17.8%	0.0%	23.7%	26.4%	27.5%
OpenSSL secp521r1[a]	(V6)	88	**100.0%**	0.0%	0.0%	1.5%	11.4%	20.3%	0.0%	1.8%	84.1%	94.3%	94.3%
BoringSSL secp521r1	(V7)	26	**100.0%**	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	96.2%	96.2%	96.2%
OpenSSL secp521r1	artificial leak	535	32.4%	0.0%	0.0%	8.3%	14.0%	**100.0%**	13.5%	0.0%	98.1%	99.8%	99.8%

[a]compiled with `enable-ec_nistp_64_gcc_128`

found that only Brainpool curves are vulnerable. Similarly, discovering and analyzing leakage of small nonces (V1) demanded careful investigations of (V10). Both issues depend on ECDSA curve parameters that are slightly above a word boundary, which led us to specifically testing the sect131r1 curve showing small nonces every fourth signature. Thus, we were able to find numerous instances of (V1) in the code with the help of our tool. Also, we could generalize these results to other curves. E.g., for secp521r1, the (V1) vulnerability only shows up every 512th signature on average, which cannot be easily discovered by DATA within a reasonable time.

Leakage Models. We evaluate the leakage models on OpenSSL 1.1.1, BoringSSL `chromium-stable` commit `2e0d354`, and LibreSSL 3.0.0. We used GCC 6.3.0, tested DATA phase one with 16 and phase three with 200 traces.

Table 6 summarizes our results. We benchmark different configurations to trigger all major vulnerabilities and count all potential leaks (differences, or Diffs) found by the original DATA phase one. For each implemented leakage model, we print the maximum correlation, which reveals the strongest leak found by a leakage model. To capture how often leakage models match, the last three columns represent the overall reduction of phase one when filtered by the models. In particular, we discard leaks with less correlation than the thresholds 1%, 50%, and 100%. For example, the 100% threshold only preserves leaks that fully match the model.

LibreSSL sect131r1 leaks small nonces via the *num_bits* model on k in several places with 100%. Moreover, LibreSSL uses leaky Euclidean inversion also for ECDSA, resulting in 100% leakage for $num_bits(kinv)$. Since LibreSSL does not work with so-called heap tracking of DATA phase one, it has over 1000 differences, most of which are filtered by our leakage models. Thus, the overall reduction is over 90%. Analyzing those leaks by hand would be quite tedious.

For OpenSSL DSA-256, the leaky k-padding addition (Listing 2 line 14) is captured by the *num_bits* models on $k+q$ and $k+2q$, showing 100% correlation. The corresponding leaky resize operation influences the heap layout and causes several subsequent Bignumber operations to leak via data accesses. Due to the high number of these actual data leaks, which are all instantiations of (V2) the reduction is "only" around 25%.

To trigger (V6), we compiled OpenSSL to use the optimized secp521r1 implementation. Indeed, $num_bits(k)$ shows 100% correlation during conversion of the nonce buffer and during scalar multiplication, as this implementation is also vulnerable to (V7). We also triggered (V7) for BoringSSL, showing 100% correlation. Other leakage models remain insignificant, and the overall reduction is above 96%.

The Hamming weight model *hw* did not show high correlation. DSA uses fixed window multiplication rather than square-and-multiply, for which *hw* is designed. ECDSA uses a blinded double-and-add by default, for which *hw* applies. However, the actual computation does not leak. To test the correctness of *hw*, we artificially introduced a conditional code execution during double-and-add, leaking the current nonce bit. Indeed, *hw* shows 100% on the padded nonce $k+q$.

8 Discussion

Proper tool support significantly improves side-channel analysis and facilitates discovery of unknown weaknesses. However, tools do not fully discharge an analyst from thorough investigations. Knowledge of the nature of expected leakage is required to leverage tool support and interpret the results. Yet, we believe this is a valuable path to follow.

The process of vulnerability patching has been tedious in the past, as evidenced by numerous issues involving the `BN_FLG_CONSTTIME` flag [23, 24, 55]. Also, patching of (V2) introduced new leakage in OpenSSL (V4) and LibreSSL (another instance of (V2) for ECDSA). We believe this is due to a lack of practical tools for developers to test their patches thoroughly. Luckily, our tool uncovered both issues with little effort. Also, regression testing with respect to already discovered leakage is promising in this regard [25].

While most OpenSSL vulnerabilities were patched or are in the patching process, the issues (V1) and (V5) related to minimal Bignumbers (`top`) remain unpatched. The OpenSSL team decided to target a fix in version 3.0, as it requires a major redesign of their Bignumber primitives. According to [19], reworking Bignumber arithmetic in BoringSSL prior to this work took between one and two months. While BoringSSL immediately fixed (V7), LibreSSL only fixed (V10),

and (V2) partially. We also were in contact with the vendors of libgcrypt, fixing (V2), and the ring library, fixing (V7) in their code, without further in-depth analysis.

Due to a change in their security policy in May 2019, OpenSSL does not consider Flush+Reload attacks in their threat model anymore, since they are mounted on the *same physical system* [41]. We see this downgrading questionable, as it not only tempers efforts to analyze OpenSSL's side-channel security but also undermines software relying on the previous threat model. For example, Intel SGX SSL [18] faces adversarial code on the *same physical system* by design. Also, vendors notified of (V2) by the CVE system were not notified of the equally dangerous (V1) due to this policy update.[20]

In the long term, more compiler support with respect to side-channels is needed [49]. As of today, compilers might optimize constant-time code in a way that re-introduces side-channel leakage. Thus, a notion of side-channel invariants like constant-time guarantees is needed on a language level.

9 Conclusion

In this work, we showed that nonce leakage is far from being abandoned and requires attention both from academia and practitioners. For our systematic study, we extended the DATA framework to detect nonce leakage and developed an easy-to-use GUI. We found that having an intuitive GUI representation of the discovered leakage is imperative for productive analysis of complex reports. E.g., it helped us to easily determine whether a leaky function deeply nested in the call stack is given public or secret input. The visualization of leakage model results furthermore helped to identify hotspots, especially if the number of potential leaks is large.

For OpenSSL and LibreSSL, we found numerous side-channel vulnerabilities leaking secret (EC)DSA nonce bits that allow full key recovery in many cases. They mostly result from weaknesses in the underlying Bignumber implementation. We open-source our tools to help developers embrace and include them in their development and patching process.

Acknowledgments

We thank our reviewers and our shepherd, Deian Stefan, for their helpful feedback. This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, via the competence center Know-Center (grant number 844595), which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria, and via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia.

[20] https://www.cvedetails.com/cve/CVE-2018-0734/

References

[1] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019:213–242, 2019.

[2] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Annual Computer Security Applications Conference – ACSAC'16*, pages 422–435. ACM, 2016.

[3] 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 *Advances in Cryptology – ASIACRYPT'14*, volume 8873 of *LNCS*, pages 262–281. Springer, 2014.

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

[5] Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. "Pseudo-Random" Number Generation Within Cryptographic Algorithms: The DDS Case. In *Advances in Cryptology – CRYPTO'97*, volume 1294 of *LNCS*, pages 277–291. Springer, 1997.

[6] 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 *Cryptographic Hardware and Embedded Systems – CHES'14*, volume 8731 of *LNCS*, pages 75–92. Springer, 2014.

[7] David Benjamin. BIGNUM code is not constant-time due to bn_correct_top, 2018. OpenSSL issue #6640, https://github.com/openssl/openssl/issues/6640.

[8] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In *Cryptographic Hardware and Embedded Systems – CHES'17*, volume 10529 of *LNCS*, pages 555–576. Springer, 2017.

[9] Daniel Bleichenbacher. On the generation of one-time keys in DL signature schemes. In *Presentation at IEEE P1363 working group meeting*, page 81, 2000.

[10] Dan Boneh and Ramarathnam Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In *Advances*

in Cryptology – CRYPTO'96, volume 1109 of *LNCS*, pages 129–142. Springer, 1996.

[11] Dan Boneh and Ramarathnam Venkatesan. Rounding in Lattices and its Cryptographic Applications. In *Symposium on Discrete Algorithms – SODA'97*, pages 675–681. ACM/SIAM, 1997.

[12] Jurjen N. Bos and Matthijs J. Coster. Addition Chain Heuristics. In *Advances in Cryptology – CRYPTO'89*, volume 435 of *LNCS*, pages 400–407. Springer, 1989.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Workshop on Offensive Technologies – WOOT'17*. USENIX Association, 2017.

[14] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In *Financial Cryptography – FC'19*, 2019.

[15] Billy Bob Brumley and Risto M. Hakala. Cache-Timing Template Attacks. In *Advances in Cryptology – ASIACRYPT'09*, volume 5912 of *LNCS*, pages 667–684. Springer, 2009.

[16] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In *European Symposium on Research in Computer Security – ESORICS'11*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011.

[17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *System Software for Trusted Execution – SysTEX*, pages 4:1–4:6. ACM, 2017.

[18] Intel Corporation. Using the Intel Software Guard Extensions (Intel SGX) SSL Library. https://software.intel.com/en-us/sgx/resource-library, 2017.

[19] Paul Dale. Close side channels in DSA and ECDSA, 2019. OpenSSL Pull Request #8906, https://github.com/openssl/openssl/pull/8906.

[20] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018:171–191, 2018.

[21] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *Conference on Computer and Communications Security – CCS'16*, pages 1505–1515. ACM, 2016.

[22] Jean-Charles Faugère, Christopher Goyet, and Guénaël Renault. Attacking (EC)DSA Given Only an Implicit Hint. In *Selected Areas in Cryptography – SAC'12*, volume 7707 of *LNCS*, pages 252–274. Springer, 2012.

[23] Cesar Pereida García and Billy Bob Brumley. Constant-Time Callees with Variable-Time Callers. In *USENIX Security'17*, pages 83–98. USENIX Association, 2017.

[24] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *Conference on Computer and Communications Security – CCS'16*, pages 1639–1650. ACM, 2016.

[25] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. Triggerflow: Regression Testing by Advanced Execution Path Inspection. In *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA'19*, volume 11543 of *LNCS*, pages 330–350. Springer, 2019.

[26] Martin Hlavác and Tomás Rosa. Extended Hidden Number Problem and Its Cryptanalytic Applications. In *Selected Areas in Cryptography – SAC'06*, volume 4356 of *LNCS*, pages 114–133. Springer, 2006.

[27] Nick Howgrave-Graham and Nigel P. Smart. Lattice Attacks on Digital Signature Schemes. *Des. Codes Cryptography*, 23:283–290, 2001.

[28] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA), 2017. Request for Comments: 8032.

[29] Cameron F Kerry and Patrick D Gallagher. Digital signature standard (DSS); FIPS pub 186-4. *Information Technology Laboratory, National Institute of Standards and Technology: Gaithersburg, MD, USA*, 2013.

[30] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[31] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-Channels. In *Computer Aided Verification – CAV'12*, volume 7358 of *LNCS*, pages 564–580. Springer, 2012.

[32] Adam Langley. ctgrind: Checking that Functions are Constant Time with Valgrind. https://github.com/agl/ctgrind.

[33] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *Security and Privacy – S&P'15*, pages 605–622. IEEE Computer Society, 2015.

[35] 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 *Cryptographic Hardware and Embedded Systems – CHES'13*, volume 8086 of *LNCS*, pages 435–452. Springer, 2013.

[36] 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: extended version. *J. Cryptographic Engineering*, 4:33–45, 2014.

[37] David Naccache, Phong Q. Nguyen, Michael Tunstall, and Claire Whelan. Experimenting with Faults, Lattices and the DSA. In *Public Key Cryptography – PKC'05*, volume 3386 of *LNCS*, pages 16–28. Springer, 2005.

[38] Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *J. Cryptology*, 15:151–176, 2002.

[39] Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.*, 30:201–217, 2003.

[40] Phong Q. Nguyen and Jacques Stern. Lattice Reduction in Cryptology: An Update. In *International Algorithmic Number Theory Symposium – ANTS'00*, volume 1838 of *LNCS*, pages 85–112. Springer, 2000.

[41] OpenSSL. Security policy, 2019. https://www.openssl.org/policies/secpolicy.html (Accessed 26/07/2019).

[42] Colin Percival. Cache Missing for Fun and Profit, 2005. Available online at http://daemonology.net/hyperthreading-considered-harmful/.

[43] Andy Polyakov. Improve ECDSA sign by 30-40%, 2018. OpenSSL Pull Request #5001, https://github.com/openssl/openssl/pull/5001#discussion_r159935593.

[44] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), 2013. Request for Comments: 6979.

[45] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe – DATE'17*, pages 1697–1702. IEEE, 2017.

[46] Keegan Ryan. Return of the Hidden Number Problem. A Widespread and Novel Key Extraction Attack on ECDSA and DSA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019:146–168, 2019.

[47] Claus-Peter Schnorr. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology – CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, 1989.

[48] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA'17*, volume 10327 of *LNCS*, pages 3–24. Springer, 2017.

[49] Laurent Simon, David Chisnall, and Ross J. Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *IEEE European Symposium on Security and Privacy – EURO S&P'18*, pages 1–15. IEEE, 2018.

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

[51] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a Little Bit More. In *Topics in Cryptology – CT-RSA'15*, volume 9048 of *LNCS*, pages 3–21. Springer, 2015.

[52] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. *CoRR*, abs/1905.13332, 2019.

[53] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security'17*, pages 235–252. USENIX Association, 2017.

[54] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *Asia Conference on Computer and Communications Security – AsiaCCS*, pages 575–586. ACM, 2018.

[55] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security'18*, pages 603–620. USENIX Association, 2018.

[56] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Annual Computer Security Applications Conference – ACSAC'18*, pages 161–173. ACM, 2018.

[57] David Wong. Timing and Lattice Attacks on a Remote ECDSA OpenSSL Server: How Practical Are They Really? *IACR Cryptology ePrint Archive*, 2015:839, 2015.

[58] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Conference on Computer and Communications Security – CCS'17*, pages 859–874. ACM, 2017.

[59] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Security and Privacy – S&P'15*, pages 640–656. IEEE Computer Society, 2015.

[60] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[61] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security'14*, pages 719–732. USENIX Association, 2014.

[62] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Cryptographic Hardware and Embedded Systems – CHES'16*, volume 9813 of *LNCS*, pages 346–367. Springer, 2016.

[63] Andreas Zankl, Johann Heyszl, and Georg Sigl. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In *Smart Card Research and Advanced Applications – CARDIS'16*, volume 10146 of *LNCS*, pages 228–244. Springer, 2016.

[64] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Conference on Computer and Communications Security – CCS'12*, pages 305–316. ACM, 2012.

A DATA GUI

Figure 5 shows the DATA GUI. It consists of several views: The left side sorts all leaks according to their call stack (top) and library (middle). Moreover, it shows for each function the number of data (D) and control-flow (CF) leaks as well as the maximum correlation with the leakage models in percent. One can see several other potential (false-positive) leaks which do not correlate with any of the predefined leakage-models. The center box gives a list of data and control-flow leaks for the selected function. The right side highlights leaks in the disassembly and the source code, if available, which is crucial for the analysis. The summary tab on the bottom left gives details about a particular leak, including correlations for various leakage models. Also, it allows the analyst to comment and rate leaks for documentation and communication purposes. Clickable elements and the synchronization of different views help to quickly navigate through complex reports.

```
1 int BN_is_bit_set(const BIGNUM *a, int n) {
2     ...
3     if (a->top <= i)
4         return 0;
5     return (int)(((a->d[i]) >> j) & ((BN_ULONG)1));
```

Listing 4: OpenSSL k-padding leaks `k->top` (V5).

B SGX Controlled-Channel Attack on (V5)

Fixing some of our reported vulnerabilities demand significant changes to the code base. For example, k-padding (V2) was fixed in OpenSSL, while the underlying problem of minimal Bignumbers still persists until OpenSSL has reworked the Bignumber implementation. Below, we show how to exploit residual leakage via the k-padding top vulnerability (V5).

During k-padding, `BN_is_bit_set` is called with the intermediate nonce buffer `l`, as shown in Listing 2 line 18. If `l->top` is smaller than `q_bits`, this causes an early abort in Listing 4 line 4. In order to exploit this leakage, an attacker needs to detect whether or not line 5 is executed.

While Flush+Reload might not work due to the small amount of leaky code, we demonstrate a controlled-channel attack [59] on an SGX enclave running the vulnerable DSA sign operation from the SGX SSL library [18]. Controlled-channel attacks detect individual memory accesses on a page granularity, be it code or data. Since the vulnerable function is likely on a single *code* page, probing this page does not suffice. Although more elaborate techniques to single-step enclave execution exist [17], we distinguish whether line 5 accesses the *data* page covering buffer `a->d`.

For the attack, we need to trace execution to the vulnerable k-padding. We do this with the SGX-Step framework [17] without using its single-stepping functionality. We unmap all relevant enclave code pages on which the following functions reside: `dsa_do_sign`, `BN_generate_dsa_nonce`, `BN_MONT_CTX_set_locked`, `BN_add`, and `BN_is_bit_set`. As soon as one of those pages is fetched by the enclave, a page fault is triggered, which we capture in user space via a custom signal handler. Then, we selectively enable only the faulted page until we hit the vulnerable `BN_is_bit_set` function. Now we also unmap the data page holding the nonce buffer `a->d`. If the next step throws a page fault on `a->d`, we know that line 5 has been executed. If not, we know that the

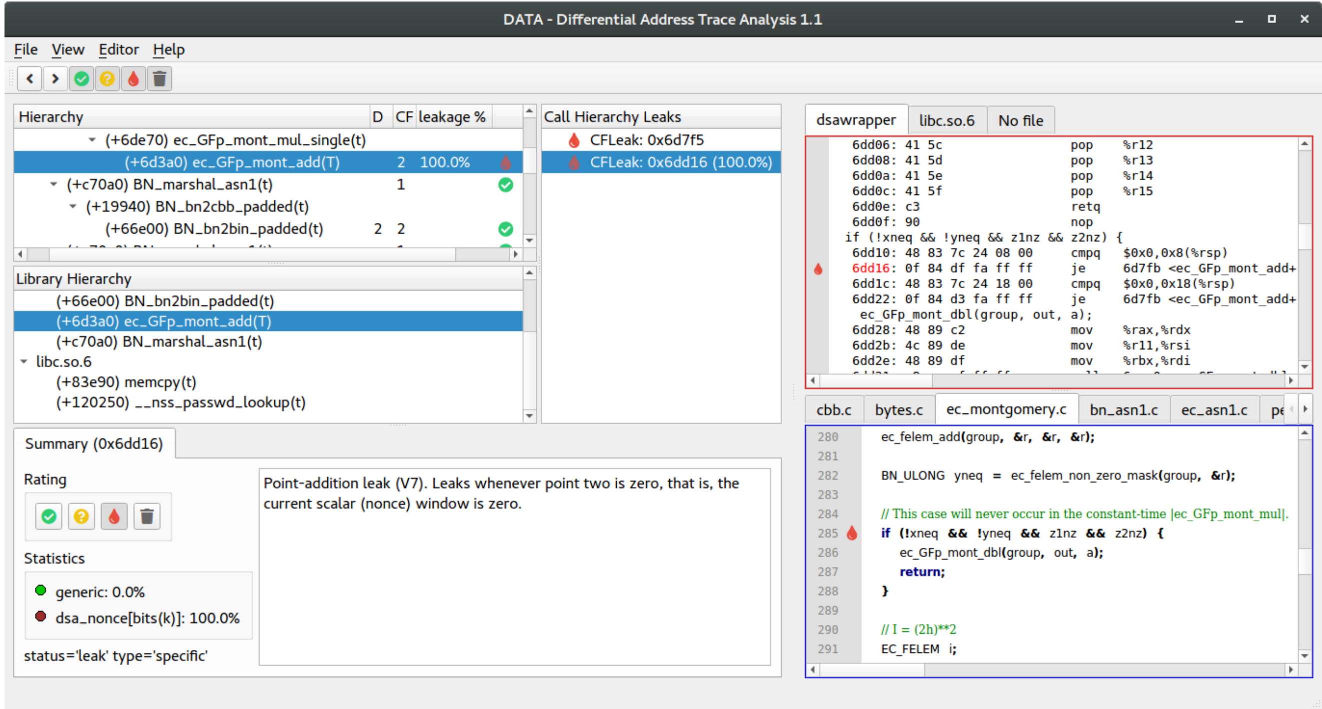

Figure 5: DATA GUI showing the point addition vulnerability (V7) in BoringSSL where the ECDSA scalar multiplication is leaking $bits(k)$ with 100%.

early abort in line 4 has been triggered. In that case the nonce was not resized in the first addition of k-padding (line 13 of Listing 2) and, thus, is smaller than the average. We only collect such signatures and mount a lattice attack.

We build the lattice according to Equation (13) and gradually fill it with leaky signatures until the lattice reduction reveals the private key. For the actual reduction, we use the BKZ algorithm with a block size of 30. For a DSA-256 modulus leaking $L = 8$ bits, recovery succeeded with 36 signatures within 3.3s. For $L = 6$ bit leakage, recovery took 47 signatures and 7.8s. $L = 4$ required 79 signatures and took 111 hours with an increased BKZ block size of 50, since it is closer to the estimated bound in Section 2.2, demanding at least $L = 3$.

For the attack to work, `a->top` in line 4 needs to be on a different page than `a->d`. This can be easily achieved if the enclave copies variably-sized attacker-controlled arguments such as messages to sign to the enclave heap. By changing the argument's size, Bignumber `a` can be shifted appropriately.

C Small Nonce Leakage Details

OpenSSL leaks the word length of small nonces in several places. Nonce generation in `BN_generate_dsa_nonce` relies on `BN_div` for nonce reduction, which is non-constant time and leaks the length of small nonces, e.g., via `BN_rshift`. Also, the nonce is stripped by skipping leading zero limbs via `bn_correct_top`, which leaks the nonce length in limbs in subsequent steps. OpenSSL's default scalar multiplication uses a blinded version of double-and-add in `ec_GF2m_simple_points_mul`. Before blinding is applied, the nonce length leaks when being copied from `scalar` to `k` via `BN_copy`, when checking its bit length via `BN_num_bits`, and during the first addition of the nonce with the cardinality via `BN_add`. Also, the NIST-optimized curves call `BN_num_bits` with the nonce as input, e.g., in `ec_GFp_nistp521_points_mul`, which also leaks (cf. [43]).

During the nonce inversion done via `BN_mod_exp_mont`, which is invoked by `ec_group_do_inverse_ord` and `ec_field_inverse_mod_ord`, there is an early abort when comparing the Bignumbers k and q via `BN_ucmp`. While their exploitation might be tricky due to the small amounts of code or data being accessed conditionally, we also found an easy-to-exploit leak, which we describe in Section 6.1.

For LibreSSL, the situation is similar. Nonce generation leaks the nonce length via an early abort condition when checking for a proper nonce during rejection sampling via `BN_ucmp`. LibreSSL also leaks the nonce length during the first addition of the nonce and the group order in `BN_add` (k-padding). However, LibreSSL accidentally performs k-padding twice; 1) in `ecdsa_sign_setup` and 2) in `ec_GFp_simple_mul_ct`. Unlike OpenSSL, nonce inversion still uses the extended Euclid and is subject to vulnerability (V8). Also, LibreSSL used an old non-constant-time version of `BN_num_bits_word` which was patched in OpenSSL already in January 2018 via commit `972c87df`. Due to our reporting, LibreSSL patched this issue in commit `9046ac5`.