Side-Channel Analysis of SM2:
A Late-Stage Featurization Case Study

Nicola Tuveri
Tampere University of Technology
Tampere, Finland
nicola.tuveri@tut.fi

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

Sohaib ul Hassan
Tampere University of Technology
Tampere, Finland
sohaibulhassan@tut.fi

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

ABSTRACT
SM2 is a public key cryptography suite originating from Chinese standards, including digital signatures and public key encryption. Ahead of schedule, code for this functionality was recently main-lined in OpenSSL, marked for the upcoming 1.1.1 release. We perform a security review of this implementation, uncovering various deficiencies ranging from traditional software quality issues to side-channel risks. To assess the latter, we carry out a side-channel security evaluation and discover that the implementation hits every pitfall seen for OpenSSL’s ECDSA code in the past decade. We carry out remote timings, cache timings, and EM analysis, with accompanying empirical data to demonstrate secret information leakage during execution of both digital signature generation and public key decryption. Finally, we propose, implement, and empirically evaluate countermeasures.

KEYWORDS
software engineering; applied cryptography; public key cryptography; side-channel analysis; timing attacks; cache-timing attacks; power analysis; TVLA; SM2; OpenSSL

1 INTRODUCTION
SM2\textsuperscript{1} is a suite of elliptic curve public key cryptosystems, standardized as part of Chinese commercial cryptography mandates. Support for SM2 in OpenSSL landed in the public GitHub repository through pull request (PR) \#4793,\textsuperscript{2} created in November 2017 by external contributors. During the review process, in January 2018, the OpenSSL team assigned the PR to the Post-1.1.1 milestone, marking functionality intended to be merged after the upcoming 1.1.1 release of OpenSSL.

Due to this, SM2 support was excluded from the two alpha releases for OpenSSL 1.1.1. But in March 2018, just before the release of the first 1.1.1 beta — and the associated feature freeze — the OpenSSL development team decided to merge the PR into the 1.1.1 beta development cycle, to have a chance to work on it and possibly include SM2 support as part of the upcoming minor release rather than waiting for the next one. Considering that new features can only be added with a new minor release and that the current one (OpenSSL 1.1.0) was released on August 2016, it is likely that a similar — if not longer — development cycle might be required before the SM2 functionality could be added to OpenSSL. The SM2 functionality has thus been part of the beta development cycle since the release of OpenSSL 1.1.1-pre3 (beta 1).

At the time of beta 1 release, the release timetable\textsuperscript{3} for OpenSSL 1.1.1 envisioned four beta releases, aiming at 15th May 2018 as the first possible final release date. As such, the addition of SM2 support into the active development branch occurred at an extremely late stage to be included in the upcoming release cycle, giving a remarkably short window for public review before the final release.

Motivation and goal. The first contribution of our work, our initial security review revealed that the late-stage featureization process resulted in various deficiencies, ranging from code quality issues to traditional software defects, and hinted at significant side-channel analysis (SCA) risks based on previous SCA results targeting ECC within OpenSSL. The goal of this research consists in empirically verifying these SCA deficiencies, and then responsibly mitigate them, aiming at intersecting the OpenSSL 1.1.1 release to ensure these vulnerabilities do not affect released versions of the library.

Furthermore, taking SM2 as a case study, we criticize the current status of the project. It demonstrates that implementing new functionality without reintroducing previously fixed vulnerabilities proves to be unnecessarily challenging, requiring intimate familiarity with internal details of lower level library modules (e.g. where, when, and how constant-time flags must be re-/enabled, which codepaths in the lower EC and BIGNUM modules require to use implementations with SCA mitigations, etc.). Hence, as a secondary goal, we also aim at reviewing the abstraction level at which current SCA countermeasures are implemented, and push for a secure-by-default approach — within the boundaries the project enforces for a minor release — so that future implementations will by default benefit from them.

Structure and our contributions. Section 2 reviews relevant background and previous work. We present our security analysis related to the integration of the SM2 functionality in the OpenSSL codebase in Section 3, offering an overview of the issues uncovered. In Section 4, Section 5, and Section 6, respectively, we evaluate SCA defects in the SM2 implementation related to remote timings, cache

\textsuperscript{1}https://tools.ietf.org/html/draft-shen-sm2-ecdsa-02
\textsuperscript{2}https://github.com/openssl/openssl/pull/4793
\textsuperscript{3}https://mta.openssl.org/pipermail/openssl-project/2018-March/000372.html
timings and EM analysis. We propose, implement and empirically evaluate appropriate mitigations in Section 7. Finally, we conclude in Section 8.

2 BACKGROUND

This section describes SM2, various SCA techniques that potentially apply to SM2 implementations, and summarizes previous work on SM2 implementation attacks.

2.1 SM2: Chinese Cryptography Standards

SM2 consists of a digital signature scheme (SM2DSA), a public key encryption scheme (SM2PKE), and a key agreement protocol. In this work, we restrict to SM2DSA and SM2PKE.

Elliptic curves and SM2. While the RFC contains cryptosystem test vectors for several different curves in simplified Weierstrass form (over both prime and binary fields), one required curve consists of all the $(x, y)$ points $(x, y \in GF(p))$ satisfying the equation

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

over $GF(p)$ along with the point-at-infinity (group identity element). The domain parameters are consistent with legacy ECC, setting $p$ a 256-bit Mersenne-like prime, $a = -3 \in GF(p)$, both $b \in GF(p)$ and generator point $G \in E$ seemingly random, and prime group order $n$ (i.e. co-factor $h = 1$) slightly below $2^{256}$.

SM2DSA digital signatures. The user’s private-public keypair is $(d_A, Q_A)$ where $d_A$ is chosen uniformly from $[1 \ldots n - 1]$ and $Q_A = [d_A]G$ holds. Denote $ZA$ the personalization string (hash) and $m$ the message. Digital signatures compute as follows.

1. Compute the digest $h = H(ZA \parallel m)$.
2. Select a secret nonce $k$ uniformly from $[1 \ldots n]$.
3. Compute $(x, y) = [k]G$.
4. Compute $r = h + x \mod n$.
5. Compute $s = (1 + d_A)^{-1}(k - rd_A) \mod n$.
6. If any of $r = 0, s = 0$, or $s = k$ hold, retry.
7. Return the SM2 digital signature $(r, s)$.

Hash function $H$ can be any “approved” function, including SM3 standardized in a parallel effort. Verification is not relevant to this work, hence we omit the description.

SM2PKE public key encryption. SM2PKE is roughly analogous to ECIES [2, Sec. 5.1]. Denote the ciphertext $C = C_1 \parallel C_2 \parallel C_3$ where, at a high level, $C_1$ represents the sender’s ephemeral Diffie-Hellman public key (point), $C_2$ is the One-Time-Pad (OTP) ciphertext (with length $|C_2|$), and $C_3$ is the authentication tag. The recipient with private-public keypair $(d_B, Q_B)$ recovers the plaintext from $C$ as follows.

1. Convert $C_1$ to a point on $E$. If $C_1$ is not on the curve or does not have order $n$, return an error.
2. Compute $(x, y) = [d_B]C_1$, the shared ECDH point.
3. Compute $z = KDF(x \parallel y \parallel |C_2|)$, the OTP key; $|z| = |C_2|$.
4. Compute $m' = z \oplus C_2$, i.e. OTP decryption.
5. Compute $t' = H(x \parallel m' \parallel y)$, the purported tag.
6. If $t' \neq C_3$ holds, return an error.

(7) Return the plaintext $m'$.

Encryption is not relevant to this work, hence we omit the description.

2.2 Remote Timing Attacks

Timing attacks exploit differences in the time required by a specific implementation to perform an operation on different inputs. In the case of hardware or software cryptosystem implementations, if there is a correlation between the timing of an operation and some secret inputs, the leaked information might be used to mount an attack to recover secret material.

In his seminal work, Kocher [49] introduces a number of simple timing attacks on modular exponentiation and modular reduction implementations, affecting implementations of public key cryptosystems with a static key such as RSA and static Diffie-Hellman or DSA implementations that precompute the ephemeral part.

Brumley and Boneh [22, 23] demonstrate that timing attacks apply also to general software systems, defying contemporary common belief, by devising a timing attack against the OpenSSL implementation of RSA decryption – exploiting time dependencies introduced by the Montgomery reduction and the multiplication routines – and ultimately retrieving the complete factorization of the key pair modulus. Moreover, they demonstrate that such attacks are practical even in a remote scenario, mounting a real-world attack through a client timing RSA decryptions during SSL handshakes with an OpenSSL server. The attack is effective when performed between two processes running on the same host, across co-located virtual machines, and in local networks. They analyze three possible defenses, favoring RSA blinding, and as a consequence several cryptographic libraries, including OpenSSL, enable RSA blinding by default as a countermeasure.

Acușan et al. [5] further improve the original attack, by targeting Montgomery Multiplications in the table initialization phase of the sliding window algorithm used to perform the RSA exponentiation in OpenSSL, rather than the exponentiation phase itself, increasing the number of multiplications that leak timing information used to retrieve one of the secret prime factors of RSA moduli.

Chen et al. [26] build on these two attacks, improving the success rate through an error detection and correction strategy, thus reducing the number of queries required to mount a successful attack and affecting the total time of the attack and its detectability.

Brumley and Tuveri [21] present another end-to-end remote timing attack: it similarly demonstrates full key recovery in local and remote scenarios, and targets the OpenSSL Montgomery’s implementation for scalar multiplication on elliptic curves over binary fields. The Montgomery ladder algorithm is often recommended as a countermeasure to side-channel attacks due to a fixed sequence of curve operations, that does not depend on the values of individual bits in the secret scalar, while still being computationally fast with no large memory overhead. Nonetheless, the attack exploits exactly the regularity feature of the algorithm, as it creates a direct linear correlation between the binary logarithm (i.e. the bit

---

1. [OID 1.2.156.10197.1.301](https://tools.ietf.org/html/draft-oscca-cfrg-sm3-02)
2. The issue uncovered by their work was tracked in the public CVE dictionary with the id CVE-2003-0147, and addressing it, OpenSSL issued a Security Advisory (17 March 2003), and CERT issued vulnerability note VU99T7481.
length of the secret scalar and the number of iterations (and thus curve operations) in the ladder.

The authors exploit this vulnerability by mounting an attack that collects several measures of the wall-clock execution time of a partial TLS handshake, using an ECHE_ECDSA ciphersuite over a binary curve. The collected measures are heavily dominated by the EC scalar multiplication of the ECDSA signature generation, implemented using the Montgomery ladder, and thus can be directly correlated with the bit length of the secret scalar (the ephemeral nonce of the ECDSA signature generation algorithm). A second, offline, post-processing phase then uses this partial knowledge to recover the full secret key through a lattice attack.

The proposed countermeasure, adopted by OpenSSL, is based on conditionally padding the nonce before the actual scalar multiplication, to always work on scalars of fixed length (i.e. adding once or twice the group order to the scalar yields an equivalent scalar with the topmost bit set) which in turn fixes the number of curve operations in the ladder and the associated execution time.7

Timing measurement noise heavily affects the success rate of the described attacks, usually resulting in the attacks being unfeasible over a wireless link and having severely limited feasibility over a WAN connection due to both decreased accuracy and the total time of the attacks (which is generally further increased to compensate the noise by collecting more samples). However, more recent results [32] address the latter scenario, studying the statistical distribution of latency over different network environments and designing specialized filters to significantly reduce the effect of jitter (i.e. the random noise on the latency introduced by additional hops in the route(s) of a network connection). These filters allow attackers to measure events with higher accuracy over the Internet, with potential effects on the feasibility of remote timing attacks over WAN connections.

Timing as a side-channel is not limited to the execution time of a whole cryptographic operation, and is often a gateway to retrieve information from other resources shared between an attacker and a victim, including microarchitecture components, as in the cache-timing attacks covered below or, switching to the domain of web privacy, even virtual constructs in modern web browsers [71, 72].

Alternatively, the timing side-channel can be used to build reliable oracles, often circumventing trivial implementations of countermeasures to prevent other side-channel attacks. In 1998, Bleichenbacher [16] presented a famous adaptive chosen-ciphertext attack on SSL/TLS ciphersuites based on RSA and PKCS#1 v1.5 encryption padding, based on an oracle built on top of different error messages sent by servers in case of malformed ciphertexts during the SSL/TLS handshake. As a result of the work, subsequent specifications of the TLS protocol (starting from RFC 2246 [33] TLS 1.0, in the same year) recommend “to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks”. But when implementations fail to extend this recommendation to the execution time of handling different events and conditions, the timing side-channel can be used to build an alternative oracle, effective for remote exploitation, as presented in 2014 by Meyer et al. [56]. Their work targeted, among others, the default Java Secure Socket Extension (JSSE) and OpenSSL implementations of the SSL/TLS protocol.

2.3 Cache Timing Attacks

Cache-timing attacks are a subset of microarchitecture attacks targeting specifically the cache hierarchy. Cache-timing attacks against implementations of cryptography primitives exploit two key features: (1) the timing variation introduced by the cache hierarchy; and (2) the non-constant time execution of algorithms handling confidential data used by cryptography primitives and algorithms, e.g. key generation [8, 73], digital signatures [11, 65], encryption [12] and key exchange [39]. Typically, the ultimate goal of a cache-timing attack is to recover confidential information from an algorithm execution and this is done by correlating cache timing data to either the execution time of the algorithm in use, its internal state during execution, or the output of the algorithm. Cache-timing attacks are feasible due to several cache attack techniques proposed and used successfully in the past, e.g. Evict+Time [63], Prime+Probe [64] and Flush+Reload [75]. The choice of attack technique depends on the attack scenario since each technique has its own advantages and disadvantages.

**Cache Architecture.** Accessing data and instructions from main memory is not an instant operation since it takes time to locate and fetch the data, thus delaying the execution of the processor. To improve the efficiency of the processor, the memory hierarchy includes memory banks called caches, located between the CPU cores and the RAM. Caches are smaller and faster compared to RAM and main memory, helping to improve the performance by exploiting spacial and temporal locality of the memory access.

Modern CPUs contain multiple cache levels, usually L1 and L2 caches are private to a specific core and the last level cache (LLC) is shared among the cores. Typically, the LLC is said to be inclusive, meaning that it contains a superset of the data of the caches below it, i.e. L1 and L2, thus it contains both instructions and data. The caches are organized into fixed size cache lines which are grouped in cache sets. The number of cache lines in a cache set is the associativity, i.e., a cache with W lines in each set is a W-way set-associative cache.

When the CPU needs to fetch data from memory, it first checks in the caches; if the data is there, a cache hit occurs and the load delay is short. On the other hand, when the data is not found in the caches, a cache miss occurs and the data must be fetched from a higher level memory, causing a longer delay. A copy of the data fetched from a higher level is cached, exploiting temporal locality. In addition, data close to the accessed data will be fetched and cached too, exploiting spatial locality. If a cache miss occurs and all the cache lines are in use, one of the cache lines is evicted, freeing space for the new data. In order to determine the cache line to evict, modern CPUs use variations of the least-recently-used (LRU) replacement policy.

**Flush+Reload.** Proposed by Yarom and Falkner [75], this powerful technique positively identifies accesses to specific memory lines with a high resolution, high accuracy, and high signal-to-noise ratio. Moreover, the technique relies on cache sharing between the CPU cores, typically achieved through the use of shared libraries.
A round of attack consists of three phases: (1) the attacker evicts the target memory line using the CLFLUSH instruction; (2) the attacker waits some time for the victim to access the memory line; (3) the attacker measures the time it takes to reload the memory line. The timing reveals whether or not the memory line was accessed by the victim during the waiting period, i.e. identifies cache hits and cache misses.

In addition to cache-timing attacks on cryptography, the FLUSH+RELOAD technique applies in clever ways targeting the kernel [45], web server function calls [77], user input [42, 51], covert channels [55], as well as more powerful microarchitecture attacks such as Meltdown [52] and the Spectre [48] attacks.

2.4 Power Analysis

Introduced by Kocher et al. [50], power analysis exploits the correlation between sensitive data and changing power leakages on the device. These power fluctuations are a result of transistor switching between the logic levels of CMOS circuits, and the current flow on data lines, as a result of processor activity and memory accesses.

Due to the tightly packed components on modern devices, power analysis can be difficult to perform with limited or no access to power rails and only noisy global power consumption. As an alternative, Electromagnetic (EM) emanations—a by-product caused by the current flow on data lines and power rails—originally proposed as cryptographic side-channels by Quisquater and Samyde [66], provides a spatial dimension to perform side-channel analysis in isolation from unwanted leakage.

Various techniques exploit data dependent EM leakage such as Differential Power Analysis [50], Correlation Power Analysis [17], Template Attacks [25] and Horizontal attacks [30, 34]. Identifying data dependent EM leakage can be challenging due to additional noise and other unwanted artifacts, thus in addition to simple frequency analysis, requires additional leakage detection using statistical tools such as Mutual Information Analysis [29], $\chi^2$-test [58] and Test Vector Leakage Assessment (TVLA) [41, 67].

Originally developed by Cryptography Research, Inc. for AES [41] and later adapted for public-key cryptography [47], TVLA is a preferred choice for applying black-box leakage detection testing to identify side-channel weaknesses [28, 61]. TVLA is based on Welch’s T-test [74], which computes a statistical value $t$ to accept or reject the null hypothesis. More specifically, the test validate whether two sets of samples are taken from similar data by comparing the averages of the two data sets. Formally, for two sets $S_1$ and $S_2$, the T-test computes as

$$t = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

where $\mu_1$, $\sigma_1$, and $n_1$ are the mean, standard deviation, and cardinality of $S_1$, respectively, and similarly for $S_2$. The T-test will fail at some discrete sample point if the value is greater than some threshold $C_t$. In the context of side-channel data, usually fixed or random test samples are compared to identify points with data dependent leakage [41].

Contemporary works demonstrate the effectiveness of EM analysis on modern PCs, embedded and mobile devices on various open source libraries such as GnuPG and OpenSSL, for attacking cryptosystems like AES [54], RSA [38], ECDFI [36], and ECDSA [37]. Moreover, e.g. Goller and Sigl [40] successfully demonstrate the viability of EM attacks over varying distances from mobile devices on ECC and RSA.

Longo et al. [54] performed localized EM analysis on a modern embedded device running software based OpenSSL AES, a bit-sliced optimized implementation for SIMD NEON core, and an AES hardware engine. They applied TVLA to identify EM leakages and subsequently carry out template attacks. Genkin et al. [37] were able to filter out EM emanations from a mobile device at very low frequencies using inexpensive equipment and additional signal processing steps. Their attack successfully recovered a few bits of ECDSA nonces, targeting the OpenSSL wNAF implementation. With roughly 100 signatures, they then successfully mounted a lattice attack for key recovery.

2.5 SM2 Implementation Attacks: Previous Work

Due to only recently being standardized and coupled with lack of sufficient public implementations and deployments, academic results on attacking SM2 implementations are limited in number. Nevertheless, existing results suggest that implementation attacks on ECDSA generally extend—with slight modification—to SM2DSA. A brief review follows.

Liu et al. [53] were the first to construct an SM2DSA analogue of existing lattice-based ECDSA key recovery with partially known nonces. The authors model exposure of three LSBs, and with 256-bit $p$ and $n$ recover a private key from 100 signatures with reasonable probability and modest computation time.

Chen et al. [27] were the first to implement an SM2DSA lattice attack with real traces. They target an SM2DSA smartcard implementation and distinguish least significant byte collisions by detecting Hamming weight with PCA-based techniques. Restricting to byte values $0\times00$ and $0\timesFF$, the authors obtain 120K signatures with power traces, filter them to 48 pairs, and iteratively construct lattice problem instances to recover a private key. Interestingly, the target is not the underlying ECC itself, but data moves by the RNG during nonce generation. In that respect, their attack is independent of the underlying ECC arithmetic.

Building on [11, 20] that focus on the LSDs of the wNAF for ECDSA nonces, Zhang et al. [76] extend the analysis to SM2DSA. With their own implementation of ECC including traditional wNAF scalar multiplication paired with SM2DSA, they demonstrate it is possible to reliably capture the sequence of ECC doubles and adds through SPA on an Atmega128. Subsequently modeling the filtered nonces with sufficient zeros in the LSDs and constructing lattice problem instances, they recover private keys with high probability. Since they target least significant zeros in the wNAF expansion, their attack is largely independent of the scalar representation—for example, it immediately applies to binary, sliding window, and fixed window expansions. Their work provides even further evidence that ECDSA-type leaks are similarly detrimental to SM2DSA.

While no English version is available, the abstract of [68] suggests a CPA attack to recover the SM2PKE session key exploiting
potential leakage from the SM3 compression function execution. That is, the target is not the ECC but the subsequent KDF.

3 SM2 IN OPENSSL

Refer to Section 1 for the detailed timeline of the SM2 feature within OpenSSL. With the narrow review window induced by the release milestone shift, several security (and functionality) issues were mainlined into the OpenSSL codebase. We give an overview of these issues in this section. Listing 1 includes an extract of the SM2DSA signature generation implementation and Listing 2 for SM2PKE public key decryption, as of OpenSSL 1.1.1-pre5 (beta 3).

Code review. Due to the hasty review process, the code implementing SM2 in the beta releases is evidently not in line with the quality standards of analogous components of libcrypto, lacking test coverage, including critical bugs (e.g. double frees and wrong return values), a lack of return values checking and poor error handling. These defects are particularly evident in the integration with the EVP_PKEY (and EVP_DigestSign) API, which is the main entry point for libssl and internal and external applications for using the cryptographic functionality included in libcrypto.

SCA review. Beyond these traditional software issues, we performed an SCA evaluation of both SM2DSA and SM2PKE in OpenSSL. This integration provides a rare opportunity to see how a straightforward implementation of an EC cryptosystem mixes with the underlying EC module for arithmetic. Our review resulted in the following observations, leveraging existing SCA results (Section 2) on the OpenSSL EC module.

1. For SM2DSA, in Listing 1 there is no scalar padding before calling EC_POINT_mul, suggesting an SM2DSA analogue of CVE-2011-1945 for remote timing attacks; see Section 4 for our empirical evaluation.

2. For SM2DSA, since there is no custom EC_METHOD for the SM2 curve, EC_POINT_mul is a wrapper ec_wNAF_mul, suggesting an SM2DSA analogue for cache timing attacks targeting scalar multiplication; see Section 5.1 for our empirical evaluation.

3. The SM2DSA implementation uses BN_mod_inverse without setting BN_FLG_CONSTTIME, suggesting an SM2DSA analogue for cache timing attacks targeting inversion via BEEA; see Section 5.2 for our empirical evaluation.

4. For SM2PKE, in Listing 2 there are no SCA considerations, suggesting (at least) DPA-style attacks on EC_POINT_mul during decryption; see Section 6 for our empirical evaluation.

The remainder of this paper is dedicated to evaluating these SCA leaks, proposing and implementing mitigations (Section 7), and empirical SCA evaluation of the mitigations (Section 7.3).

4 SM2DSA: REMOTE TIMINGS

We note the lack of scalar padding before calling EC_POINT_mul, suggesting an SM2DSA analogue of CVE-2011-1945. To evaluate the impact of this vulnerability, we correlate nonce lengths and the

Listing 1: Source code from crypto/sm2/sm2_sign.c in OpenSSL 1.1.1-pre5 for SM2DSA signature generation.

Listing 2: Source code from crypto/sm2/sm2_crypt.c in OpenSSL 1.1.1-pre5 for SM2PKE decryption.
execution time of signature generations, adopting a process similar to the one presented by Brunley and Tuveri [21].

We wrote an OpenSSL client application which repeatedly generates SM2DSA signatures for a given plaintext, under the same private key. For each generated signature, the program measures the execution time of the operation (in CPU cycles) and retrieves the associated nonce by monitoring the PRNG. We repeated the experiment using both the recommended SM2 prime curve and the standardized K-283 binary (Koblitz) curve [3], as the library executes two different code paths for EC_POINT_mul over prime and binary curves. We then analyzed the captured data to correlate the timings with the binary logarithm (bit-length) of the nonces.

We ran these experiments on a 4-cores/4-threads Intel Core i5-6500 CPU (Skylake) running at 3.2GHz, with Enhanced Intel SpeedStep Technology and Intel Turbo Boost Technology disabled. Figure 1 shows cumulative distribution functions (CDF) for different nonce bit-lengths for the two curves, collating 4 million samples for each curve. Both plots show a strong correlation between the bit-length of the nonce and the execution time of the signature generation, which in turn is distinctly dominated by the execution time of the underlying EC_POINT_mul operation.

Figure 1: SM2DSA latency dependency on the nonce length on amd64 architecture in OpenSSL 1.1.1-pre3. Top: K-283 binary curve. Bottom: Recommended SM2 prime curve.

Generic binary curves. The top plot of Figure 1 shows that, using a generic binary curve as the underlying elliptic curve for SM2DSA, the timing correlation appears easily exploitable to mount a remote timing attack similar to [21]. For generic binary curves, OpenSSL implements the EC_POINT_mul operation through a Montgomery ladder algorithm, which due to its extreme regularity in the sequence of EC additions and doublings, results in an overall execution time directly proportional to the binary logarithm of the secret EC_POINT_mul scalar (i.e. the SM2DSA nonce). As a result, each nonce bit-length exhibits a clearly distinct CDF, and suggests simple thresholding on the execution time to filter signatures associated with a specific nonce length with high probability.

Recommended SM2 curve. When using the recommended SM2 prime curve. OpenSSL 1.1.1-pre3 implements the EC_POINT_mul operation using the generic prime curve codepath, using a wNAF algorithm (see Section 5.1). The bottom plot of Figure 1 shows that, similarly to the previous case, there is a strong correlation between the execution time of SM2DSA and the associated nonce length. We note that in this case, mounting a practical attack poses more challenges due to a less distinct separation between the different CDFs, likely compensated by collecting more samples.

5 SM2DSA: CACHE TIMINGS

As mentioned in Section 2.5, several previous works show SM2DSA vulnerable to ECDSA-type SCA attacks. For that reason, we explore and analyze the cryptosystem applying existing cache-timing attack techniques to code paths known for leaking information, and exploited successfully in the past for ECDSA [11, 65].

For our analysis, we use the FLUSH+RELOAD technique [75] paired with a performance degradation attack [9, 65]. This combination of techniques allows us to accurately probe relevant memory addresses with enough granularity to confirm bit leakage on both scalar multiplication and modular inversion operations.

5.1 Scalar Multiplication

SM2DSA in OpenSSL performs scalar multiplication operations by calling the EC_POINT_mul function in SM2 sig_gen @ crypto/rsa/sm2/sm2_sig.c, which is only a wrapper to the underlying ec_wNAF_mul function. The ec_wNAF_mul function is a generic code path performing scalar multiplication, i.e. \( [k]G \) in SM2DSA, by executing a series of double and add operations based on the wNAF representation of \( k \). This code path is vulnerable to cache-timing attacks due to its non constant-time execution, targeted previously using cache-timing techniques [9, 20, 70, 75]. Generally, the strategy is to trace the sequence of double and add operations, which leaks LSDs of \( k \), leading to private key recovery.

Unlike previous attacks, during our analysis we do not probe memory lines directly used in functions EC_POINT_add and EC_POINT_db1, but instead we focus in low level functions BN_rshift1 and BN_lshift. The BN_rshift1 function is one of several functions called during EC_POINT_add execution and, unlike the rest of the functions in the routine, BN_rshift1 is a representative of the add operation. Similarly, BN_lshift is a representative of the double operation, allowing to identify add and double operations respectively during scalar multiplication. Therefore, these low level functions allow accurately detecting when add and double operations execute. By tracing the sequence of BN_rshift1 and BN_lshift operations, we are able to determine with high accuracy the sequence of double and add operations, leaking LSDs of \( k \). Top trace in Figure 2 shows a post-filtered cache-timing trace of a scalar multiplication with a random nonce \( k \) during SM2DSA. The probes detect the sequence of curve operations from left to right as follows: 1 double, 1 add, 4 doubles, 1 add, 4 doubles, 1 add, 7 doubles, 1 add, 4 doubles, and 1 add; thus revealing partial information on \( k \).
5.2 Modular Inversion

Modular inversion is a common operation during digital signatures and in OpenSSL, SM2DSA uses the \texttt{BN\_mod\_inverse} function for this purpose. This function executes one of several GCD algorithm variants. Unfortunately, most of these variants are based on the Euclidean algorithm which executes in a non constant-time fashion. The Euclidean algorithm and variants are highly dependent on their inputs and previous research exploits some of these variants [4, 8, 65].

During SM2DSA execution, none of the input values has the flag \texttt{BN\_FLG\_CONSTTIME} set when entering to the \texttt{BN\_mod\_inverse} function, therefore the function takes the default insecure path, calculating the modular inverse of $d_A + 1$ through the Binary Extended Euclidean Algorithm (BEEA). More importantly, this operation executes every time a signature is generated with the exact same input values, therefore an attacker has several opportunities to trace the BEEA execution on the private key.

Similar to the scalar multiplication case, we identify the low level operations leaking bits from the input values. In the BEEA case, this means functions \texttt{BN\_rshift1} and \texttt{BN\_sub}. By placing probes in memory lines in these routines, we are able to trace the sequence of shift and subtraction operations performed during modular inversion, leading to partial bit recovery of $d_A + 1$. Using algebraic methods [7], it is possible to recover a variable amount of private key LSDs from these sequences. Bottom trace in Figure 2 shows the end of a post-filtered cache-timing trace capturing the execution of the BEEA during SM2DSA. The trace matches the sequence 2 shifts, 1 subtract, 1 shift, 1 subtract, 2 shifts, 1 subtract, 1 shift, 1 subtract, 1 shift; obtained from a perfect trace computed by executing BEEA on the inputs taken from the SM2DSA signature test, demonstrating private key leakage.

6 SM2PKE: EM ANALYSIS

As discussed in Section 2.1, SM2PKE decryption computes the shared secret using the receiver’s private key $d_B$ and the sender’s ECDH public key $C_1$. The point multiplication $[d_B]C_1$ can leak intermediate values which can be exploited using both vertical attacks [31, 35, 62] and horizontal attacks [10, 34] for key recovery.

To evaluate the side-channel leakage for SM2PKE, we applied Test Vector Leakage Analysis (TVLA) using Welch’s T-test [41, 67]. We took an approach similar to [28, 61] for ECC, with a reduced set of test vectors. We divided the test vectors into three different sets $\{S_i\}$ for $i = 1, 2, 3$. The sets $S_1$, $S_2$ and $S_3$ contained traces for fixed key $d_B$ and fixed cipher text $C_1$, fixed $d_B$ and varying $C_1$, fixed $C_1$ and varying $d_B$, respectively. We performed the tests in pairs, such that $(S_1, S_2), (S_1, S_3)$ would fail the T-test if the resulting confidence threshold satisfies $|\tau| > 4.5$. We selected the value $\tau$ (a function of number of samples) based upon empirical evidence from Jaffe et al. [47].

Experimental setup. We performed the experiments on an AM335x Sitara SoC ♯ featuring a 32-bit ARM Cortex-A8 embedded on a BeagleBone Black ¶ development board. We used the standard BeagleBone Debian distribution (“Wheezy” 7.8) while keeping all the default configurations intact. For capturing the EM traces, we used a Langer LF-U5 near-field probe (500kHz to 50MHz) and 30dB Langer PA-303 low noise amplifier. We positioned the probe head directly on the SoC, seeking to strengthen the acquisition quality. We procured the traces using a PicoScope 5244B digital oscilloscope at a sampling rate of 125 MSamples/sec with a 12-bit ADC resolution. Figure 3 shows our setup for the EM analysis.

EM acquisition. For the purpose of this analysis, we captured 1500 EM traces for each set $(S_1, S_2, S_3)$ while performing the decryption operation. We fixed the clock frequency at 1GHz to avoid any bias in the captured traces. To acquire traces, we initially utilized the GPIO pin of the board to trigger the oscilloscope. However, this trigger proved unreliable as it encountered random delays. To improve this, we applied correlation based matching to locate the beginning of the trace. As most of the EM signal energy was concentrated at much lower frequencies, we also applied a Low Pass filter with a cut-off frequency at 15MHz.

Due to noise in the traces, we performed additional processing steps. For the envelope detection, we applied a Digital Hilbert Transform, followed by a Low Pass Filter to smooth out any high frequency noise. From the sets, we dropped traces containing noise.

---

♯http://www.ti.com/processors/sitara/arm-cortex-a8/am335x/overview.html
¶https://beagleboard.org/black
due to preemptive interrupts and other unwanted signal features. In the end, we retained a total of 1000 traces per set. Since the T-test required averaging multiple traces, we aligned the traces at each point of interest (i.e. ECC operations). Figure 4 shows part of an actual processed EM trace, depicting a sequence of ECC double and add operations.

**T-test.** To validate the results, we divided each set into subsets \( \{S_{ia}\} \) and \( \{S_{ib}\} \) and performed an independent T-test between sets \( \{(S_{ia}, S_{ia})\} \) and \( \{(S_{ib}, S_{ib})\} \) for \( k = 2, 3 \). We performed a further test by combining an equal number of randomly selected traces from both \( \{S_1\} \) and \( \{S_k\} \) such that the two resulting subsets were disjoint. A correct T-test for the random sets \( R_1 = \{(S_1 \cup S_k)\} \) and \( R_k = \{(S_1 \cup S_k) - R1\} \) should result in confidence threshold \( |C_T| < 4.5 \) for all the points in the traces.

The experiments showed multiple points where the T-test failed for both \( \{(S_1, S_2), (S_1, S_3)\} \). Figure 5 shows two T-test results for \( \{(S_{ia}, S_{ia})\} \) and \( \{(S_{ib}, S_{ib})\} \). It is clear from the figure that the T-test values have a significant number of peaks satisfying \( |C_T| > 4.5 \) for both tests, roughly at the same points. This demonstrates there is a strong leak at these points, since we performed both tests on different sets of traces. From the random sets \( \{R_1, R_2\} \) the confidence threshold remains \( |C_T| < 4.5 \) which further validates our hypothesis. Similarly, Figure 6 shows the failed T-test for \( \{(S_{ia}, S_{ia})\} \) and \( \{(S_{ib}, S_{ib})\} \).

### 7 SCA MITIGATIONS

In this section, we describe our results on mitigating the discovered leaks. We claim no novelty for the mitigations themselves, only their application and implementation within the OpenSSL library; they are standard techniques known since at least the 90s.

We stress the focus of our mitigation effort is not on SM2 nor any individual cryptosystem, but rather on the EC module itself, to provide transparent secure-by-default behavior to cryptosystems at the architecture level. That is, conceptually it should be completely reasonable to drop in a cryptosystem implementation like was done with SM2DSA or SM2PKE and have it resist SCA, with absolutely no esoteric knowledge of OpenSSL internals that control SCA features such as constant-time flags.

#### 7.1 Scalar Multiplication: SCA Mitigations

**Ladder.** While it is indeed feasible to reduce leakage in OpenSSL’s \( wN\text{AF} \) scalar multiplication code path [18], tediously straightlining conditions and making table lookups regular adds significant code complexity, increases the probability of defects, and generally results in low maintainability code. Even then, there is no guarantee that all leakage issues are addressed: the code path was not initially intended to resist SCA, and retrofitting mitigations becomes awkward.

We instead implemented an early exit from \( \text{ec}_\text{wN\text{AF}_\text{mul}} \) that—irrespective of the constant time flag—diverts to a new single scalar multiplication function for all instances of \( k|G \) (fixed point, e.g. ECC key generation, SM2DSA signing, ECDSA signing, first half of ECDH) or \( |k|P \) (variable point, e.g. SM2PKE decryption, last half of ECDH), and falls back to the existing (insecure) \( wN\text{AF} \) code in all other cases (e.g. \( |a|G \) + \( |b|P \) in various digital signature scheme verifications). For cryptosystem use cases internal to the OpenSSL library, this provides secure-by-default scalar multiplication code path traversal.

For this new functionality, we chose the traditional powering ladder due to Montgomery [57], heralded for its favorable SCA properties. In modern implementations, straightlining the key-dependent ladder branches happens in one of two ways [60, Sec. 2]: “either
by loading from (or storing to) addresses that depend on the secret scalar, or by using arithmetic operations to perform a conditional register-to-register move. The latter approach is very common on large processors with cache, where the former approach leaks through cache-timing information."

We see both in practice: For example, TomsFastMath11 does not branch but reads and stores using (secret) pointer offsets, while Mbed TLS12 parses all the data and performs a manual conditional swap with arithmetic, even documenting their function mbedtls_mpi_safe_cond_swap with the comment: "Here it is not OK to simply swap the pointers, which would lead to different memory access patterns when X and Y are used afterwards." This is in contrast to e.g. [44, Sec. 8.5]: "we implement the conditional swap operation after each ladder step by swapping pointer variables instead of data. We expect slightly better performance and also a reduced side-channel leakage." While that is perhaps a valid strategy on architectures lacking cache memory, we feel it is generally dubious advice since typical engineers are usually unaware of SCA subtleties.

Regardless, the "standard way" according to Bernstein [14, Sec. 3] uses arithmetic to implement conditional swaps on the data, not the pointers; the work also reviews a slight optimization, which we also implement. The two contiguous swaps conditional on bits $k_1$ and $k_{−1}$ reduce to a single swap by XOR-merging the condition bits, i.e. only swap if the bit values differ. This optimization halves the number of conditional swaps.

Scalar padding. The above conditional swaps ensure favorable SCA behavior for ladder iterations. But [21] exploits the number of said iterations, fixed in an ECDSA-only fashion in 2011 by padding nonces. We remove this padding, and instead push it to the underlying EC module to ensure a constant number of ladder iterations. To accomplish this in an SCFA-friendly way, we construct two values $k' = k+n$ and $k'' = k'+n$, subsequently using the above conditional swap to set $k$ to either $k'$ or $k''$, whichever has bit-length precisely one more than $n$. We apply this padding directly preceding ladder execution.

Coordinate blinding. Originally proposed by Coron [31, Sec. 5.3] for standard projective coordinates as a DPA countermeasure, coordinate blinding transforms the input point to a random representative of the equivalence class. For generic curves over $\text{GF}(p)$, OpenSSL’s formulae are a fairly verbatim implementation of Jacobian projective coordinates [1, A.9.6] where the relation

\[(X, Y, Z) \equiv (\lambda^2 X, \lambda^3 Y, \lambda Z)\]

holds for all $\lambda \neq 0$ in $\text{GF}(p)$. Our implemented mitigation generates $\lambda$ randomly, applying the map a single time directly preceding the ladder execution. This is, for example, the approach taken by Mbed TLS (function ecp_randomize_jac).

7.2 Modular Inversion: SCA Mitigations

Directly due to the work by Gueron and Krasnov [43, Sec. 6], OpenSSL integrated a contribution from Intel that included (1) high-speed, constant-time P-256 ECC on AVX2 architectures; (2) constant-time modular inversion modulo ECDSA group orders. It did the latter by internally exposing a function pointer within the EC_METHOD structure. If set, ECDSA signing code path calls said pointer (for which the custom P-256 method has a dedicated function), otherwise a series of default fallbacks including (1) FLT inversion with Montgomery modular exponentiation; (2) normal EEA-based inversion. We refactored the structure to expose this default behavior within the wrapper that checks the function pointer, the end goal being to expose it to the EC module as a whole and not limit to ECDSA, in turn allowing SM2DSA access to a strictly secure-by-default functionality. We explored two different options for inversion default behavior that resist SCA, summarized below.

Blinding. The classical way to compute modular inversions is through the EEA utilizing divisions, or binary variants utilizing shifts and subtracts. However, as previously described their control flow can leak critical algorithm state. Nevertheless, to prevent direct input deduction from this state one option is to choose blinding value $b$ uniformly at random from $[1..n]$ then compute $k^{-1} = b(bk)^{-1}$ at the additional cost of two multiplications. This is, for example, the approach taken by Mbed TLS for ECDSA nonces.

Exponentiation. Although initially motivated by binary fields with normal basis representation where squaring is a simple bit rotation, the algorithm by Itoh and Tsujii [46] is one of the earliest examples of favorable implementation aspects of using FLT for finite field inversion. SCA benefits followed thereafter, e.g. Curve25519 where Bernstein [13, Sec. 5] weighs blinded EEA methods versus FLT: "An extended-Euclid inversion, randomized to protect against timing attacks, might be faster, but the maximum potential speedup is very small, while the cost in code complexity is large."

Performance and security. Regarding security, it is clear that either method is a leap forward for OpenSSL with respect to secure-by-default. We feel that blinding has an intrinsic advantage over FLT-based methods, since the former resists bug attacks [15, 19] that exploit predictable execution flows. Regarding performance, we benchmarked both approaches to measure the potential differences alluded to by Bernstein, and found the results consistent. On an Intel Core i5-6500 CPU (Skylake) running at 3.2GHz, after all of our described and implemented countermeasures, one SM2DSA execution takes on average 1760913 cycles with FLT, and 1750984 cycles with blinded BEEA—a difference of a fraction of a percent.

In the end, the OpenSSL team declined our blinding contribution. They plan to increase the usage of the Montgomery arithmetic context within the EC module, so in that sense their decision is rational from a software architecture perspective. The team instead integrated our FLT refactoring, sufficient to thwart the attack in Section 5, and furthermore provide secure-by-default behavior to future callers conforming to the convention set by this API.

7.3 SCA Mitigations: Evaluation

Remote timings: evaluation. Using the same approach adopted in Section 4, Figure 7 shows the cumulative effect of three countermeasures: adopting the Montgomery ladder instead of the νNMF algorithm for regular scalar multiplication, scalar padding, and computation of modular inversion via exponentiation through FLT.
Both plots clearly show that the latencies measured for signature generation using nonsces of different bit-lengths are indistinguishable, effectively preventing the attack, and a comparison with Figure 1 immediately shows the extent of the leakage reduction.

Cache-timings: evaluation. After introducing the mitigations, when SM2DSA performs a scalar multiplication it first calls the EC_mul function, a wrapper to ec_wNAF_mul. There the code takes an early exit, jumping to the powering ladder regular algorithm to perform a fixed point scalar multiplication \(|k|G\). From the cache perspective, the ladder implementation consists of an always-double-and-add algorithm, largely unrelated to the wNAF representation of the nonce \(k\). To support our claim, we follow the same approach as in Section 5, placing probes in the same underlying functions BN_rshift1 and BN_lshift—called by EC_POINT_add and EC_POINT_db1—to trace the sequence of operations during scalar multiplication. Top trace in Figure 8 shows an example trace, which indeed tracks the sequence of double and add operations successfully, but due to the regular nature of the powering ladder algorithm, no meaningful information can be retrieved from this sequence.

During modular inversion, the high level function SM2SigGen in SM2DSA no longer calls BN_mod_inverse but instead it calls directly EC_GROUP_do_inverse on the private key \(dA + 1\). This function computes modular inversion by performing an exponentiation using FLT, therefore the underlying algorithm and its implementation are completely different compared to the Euclidean algorithm (and variants) used previously. Recall that during modular inversion using FLT, the exponent value is public; said value does not require SCA protection. Bottom trace in Figure 8 shows an example trace during modular inversion, probing the square and multiply operations based on the public exponent.

EM leakage: evaluation. To validate the efficacy of the applied mitigations, we repeated the T-test experiments (Section 6). Figure 10 shows the results of the new T-test for both fixed vs random key \((S_1, S_2)\) and fixed vs random point \((S_1, S_2)\). Figure 9 shows the EM traces, reflecting a regular sequence of ECC double and add operations due to ladder point multiplication. One interesting observation is the increase in the number of peaks for the add operation compared to Figure 4. This is due to the fact that ec_wNAF_mul uses mixed coordinates (projective and affine), a code path with less field operations compared to the fully projective coordinate path taken by the ladder.

It is clear from Figure 10 that the T-test shows a significant improvement due to the combined ladder application and projective coordinate randomization. The T-test easily passed for fixed vs random point \((S_1, S_2)\) with \(|CT| < 4\). In case of fixed vs random key \((S_1, S_2)\) we still observe a marginal number of peaks with magnitude roughly 6. In theory, it is still possible to exploit this; e.g. a key value that leading to special intermediate points on the curve such as zero-value \([6]\) or same value points \([59]\). However, the leakage is so minimal, our analysis suggests mounting such attacks would be extremely difficult and feature significant data complexity. Moreover, the scalar randomization countermeasure \([31]\) to thwart this leak introduces performance overhead, in this case unacceptable to OpenSSL when weighed vs risk.

8 CONCLUSION

Subsequent to an accelerated OpenSSL milestone to support SM2 cryptosystems, our work began with a security review of SM2DSA and SM2PKE implementations within OpenSSL pre-releases. Part of our review uncovered several side-channel deficiencies in the merged code, which we then verified with empirical remote timing, cache-timing, and EM traces. To mitigate these discovered vulnerabilities, we proposed and implemented several mitigations, now mainlined into the OpenSSL codebase. These mitigations target the underlying EC module, providing secure-by-default behavior not
outcome. We strongly encourage adhering to the above two points to assist avertting future security vulnerabilities.

**Acknowledgments**

Supported in part by Academy of Finland grant 303814.

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

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

---

**REFERENCES**


---

only for SM2 but future cryptosystems in the ECC family. Notably, the mitigations also bring security to the generic curve scalar multiplication code path in OpenSSL, a longstanding vulnerability since 2009. Finally, we performed an empirical SCA evaluation of these countermeasures.

We met our goal to intersect the upcoming OpenSSL 1.1.1 release and ensure these vulnerabilities do not affect release versions. However, given a more relaxed schedule, we outline future work to improve this secure-by-default approach: (1) the antiquated ECC point addition and doubling formulae should be renovated to more recent exception and/or branch-free versions; (2) support for ladder step function pointers, for more efficient ladder operations w.r.t. finite field operations; (3) at the standardization level, SM2DSA private key formats that, similar to RSA private keys with CRT finite field operations; (3) at the standardization level, SM2DSA private key formats that, similar to RSA private keys with CRT

---

**Figure 9:** The filtered EM trace after applying the ladder countermeasure. As expected, it clearly reveals the sequence of ECC double and add operations during SM2PKE decryption, yet this sequence is regular and not useful for SCA-enabled attackers.

---

**Figure 10:** Top: T-test results between sets $S_1$ and $S_2$ versus sample index; for fixed vs random $k$ the test marginally fails with leaks at the few points where the threshold is around 6. Bottom: T-test results between sets $S_1$ and $S_3$ versus sample index; for fixed vs random $c$ the test passes since no peaks exceed the 4.5 threshold.
As stated in the Introduction, as a secondary goal, we aimed at discussing in this work at the lowest possible abstraction level. Through a systematic review of the abstraction level at which SCA countermeasures are presented, we proposed to move each one of the SCA countermeasures secure-by-default approach, reviewing the abstraction level at which SCA countermeasures were presented.

In this paper, we focus on two families of attacks: (1) those that only exploit timing information, enabling us to perform a comparative study of the effectiveness of the SCA countermeasures against them, and (2) those that leverage both timing and memory information, which can be more difficult to mitigate as they are often more stealthy. We aim to provide insights into the effectiveness of SCA countermeasures and to guide future research in this area.

A REMOTE TIMINGS SCA EVALUATION: ECDSA

As stated in the Introduction, as a secondary goal, we aimed at reviewing the abstraction level at which SCA countermeasures are implemented. Specifically, pushing for a secure-by-default approach, we proposed to move each one of the SCA countermeasures discussed in this work to the lowest possible abstraction level.
As a result, the changes we proposed affected also other existing cryptosystems, increasing their resistance to SCA. In particular, in this section we evaluate the impact of our patchset on the ECDSA cryptosystem, specifically when using generic prime curves (as opposed to curves for which an alternative optimized implementation is specifically provided).

Figure 11 shows an empirical evaluation—similar to the one presented in Section 4 and Section 7—on the impact of the proposed mitigations on ECDSA over the secp256k1 [24] GLV prime curve used in the BitCoin protocol.

Both plots show the latency dependency on the nonce length: the top plot related to the OpenSSL implementation as of version 1.1.1-pre3, while the bottom plot shows the results after applying the proposed patchset. Specifically, the original implementation already applied the nonce padding and the FLT modular inversion countermeasures, so the main change between the two implementations is due to adopting a Montgomery ladder algorithm for EC scalar multiplication instead of the wNAF algorithm adopted in the original implementation (see Section 7).

Comparing the two plots, our mitigations introduce an improvement centered around the median values. This is due in part to the fact that timings in the top plot depend on the weight of scalars, while the timings in the bottom plot are independent of the weight. This leads to lower deviations for the majority of data points centered around the median.

Figure 11: Latency dependency on the nonce length for ECDSA signature generation over the secp256k1 prime curve on amd64 architecture. Top: OpenSSL 1.1.1-pre3, in which a wNAF algorithm is used to implement EC scalar multiplication (see Section 5.1). Bottom: After applying our patchset (see Section 7), most notably switching to a Montgomery ladder algorithm for scalar multiplication instead of wNAF.