



SoK: The Constant Time Model

Billy Bob Brumley
Rochester Institute of Technology

Abstract

Constant time programming patterns is the primary defense against timing attacks on cryptographic implementations, yet what "constant time" means varies across academia and industry. This work systematizes constant time models and their evolution, identifies a recurring gap between what models protect and what specifications assume, and distills an offensive methodology for discovering timing vulnerabilities that originate outside the cryptographic primitive boundary. Applying this methodology, we locate a specification-level vulnerability related to private key loading, and confirm the leak in both OpenSSL and BoringSSL. Counterintuitively, BoringSSL's per-observation signal is several orders of magnitude stronger than OpenSSL's, despite an explicitly stricter threat model.

1 Introduction

Timing attacks on cryptographic implementations date to the seminal work of Kocher [33], who demonstrated that secret-dependent execution time in modular exponentiation leaks private key material. The three decades since have produced a rich body of work exploiting timing leaks across increasingly sophisticated microarchitectures, from data cache attacks on AES [11, 42] and RSA [44] to speculation attacks [32, 36] and beyond, broadening timing attacks from strictly a cryptography concept to a system security one.

The primary defense is constant time programming paradigms, ensuring that execution flow and memory access patterns are independent of secret data. In practice, what "constant time" means varies considerably. Multiple models have emerged over the past two decades, each making different trade-offs between security guarantees and practical constraints such as performance, deployability, intellectual property, and risk. The security implications of those trade-offs are not always obvious.

An orthogonal problem is that constant time models typically focus on cryptographic primitives in isolation, or "crypto in a vacuum." Specifications that invoke these primitives (e.g., TLS, SSH, PKCS, X.509, etc.) routinely mandate operations on secret data that are difficult or impossible to implement in constant time. Historical examples include padding validation with error handling [17, 54], variable-length encoding of secrets [40], and key persistence formats that leak through encoding [51]. These specification-level vulnerabilities affect multiple implementations independently, not due to shared code but because they all must satisfy the same requirements to achieve interoperability.

Exacerbating the situation, compilers and hardware can silently undermine constant time programming patterns in source code. Compiler optimizations have introduced branches on secret data in Curve25519 [29] and ML-KEM [9] implementations, while variable-time instructions such as integer division affected a wide range of Kyber implementations [15]. These failures exist only in compiled binaries, invisible to source-level analysis.

Contributions. This paper makes three contributions. First, we systematize the evolution of constant time models (Section 2) and compare their features, guarantees, and documented failures (Section 3), identifying gaps between what models protect and what specifications assume. Second, we distill an offensive methodology (Section 4) from a broad analysis of specification-level vulnerability classes that provides a systematic framework for discovering constant time violations that originate outside the cryptographic primitive boundary. Third, we apply this methodology to EC private key loading (Section 5), demonstrating that both OpenSSL and BoringSSL accept non-conformant variable-length keys and that BoringSSL's per-observation signal is several orders of magnitude stronger than OpenSSL's, despite an explicitly stricter threat model.

Outline. Section 2 introduces the constant time models and traces their evolution. Section 3 compares these models and identifies gaps in empirical validation, compiler optimization, and at the protocol level. Section 4 develops our offensive methodology through case studies of specification-level vulnerabilities. Section 5 applies the methodology empirically to

EC private key loading. We conclude in Section 6.

2 Constant Time Models: A Brief History

The evolution of constant time models in cryptography reflects the symbiotic relationship between implementers focused on efficiency and attackers discovering increasingly subtle leakage sources. What began as simple considerations for embedded systems has evolved into a complex landscape of threat models, each attempting to balance security guarantees with practical requirements. Understanding this evolution helps show why the traditional constant time model, despite numerous attempts to relax its constraints over the years, remains the gold standard for secure implementation.

2.1 The Program Counter Security Model

The earliest formal constant time considerations emerged in the context of embedded systems and smart cards, where microarchitectures were relatively simple and lacked complexity such as cache hierarchies and memory management units. At this point, seminal work by Kocher [33] in 1996 had already demonstrated the risk that timing attacks pose to implementations of various public key cryptosystems, and Quisquater and Samyde [48] in 2001 on the physical channel side for symmetric key cryptosystems on embedded devices.

The Program Counter Security Model proposed by Molnar et al. [41] formalized the approach that was being deployed in the embedded space at the time (2004–2005). The model allows an attacker access to all values of the program counter as the cryptography application executes. If this sequence is fixed and constant, the implementation should resist timing attacks, because constant program counter values leads to constant execution time in their model where all instructions have fixed latency. The authors admit this assumption is unlikely to hold outside the embedded space: "This is not true on some architectures, due to, among other things, cache effects, data-dependent instruction timing, and speculation" [41, Sec. 3].

The rationale for allowing table lookups in this model was device driven. In processors without caches, accessing any memory location required the same number of cycles. A lookup table implementation of AES S-boxes, for instance, appeared to offer constant-time execution since each table access took identical time regardless of the index. A square-and-multiply-always algorithm for modular exponentiation (RSA, Diffie-Hellman, DSA, etc.) that calls a single multiplication function for both the square step and the multiply step would not, in this model, allow attackers to distinguish between calls with distinct operands (multiply) and calls with identical operands (square). This model assumes that the primary observable characteristic was branching and execution time at the instruction level, not the pattern of memory addresses (containing data, not code) being accessed.

However, this simplistic model failed to account for several crucial factors, including, for example, the observation by Kelsey et al. [30, Sec. 5]: "We believe attacks based on cache hit ratio in large S-box ciphers ... are possible." This is not a failure in the model design per se, as a major goal of the model was to reflect industry practice at the time. As these implementations were ported to more complex processors, the security assumptions of the Program Counter Security Model began to break down.

2.2 The Traditional Constant Time Model

The traditional constant time model emerged gradually through the work of several researchers, rather than being defined in a single seminal paper. The model took form through a series of contributions that each highlighted different threats not accounted for in the program counter model.

The 2002–2003 work by Page [43] on the theoretical side and Tsunoo et al. [53] on the empirical side showed that secret-dependent lookups from memory-resident tables in DES implementations leads to key recovery attacks through cache timings. This eventually fed into the 2004–2005 work by Bernstein [11], a contribution on the offensive side with a data cache-timing attack on AES, but also on the defensive side with guidelines for implementers of cryptosystems. Bernstein warned from the data perspective: "Using secret data as an array index is a recipe for disaster" [11, Sec. 1]. Even if less relevant for AES with its largely straight-line structure, Bernstein furthermore warned from the code perspective and input-dependent branches: "Skipping an operation is faster than doing it" [11, Sec. 9]. These two principles eventually became the core pillars of the traditional constant time model.

In parallel work, Osvik et al. [42] introduced their Prime+Probe and Evict+Time microarchitecture timing attack techniques, applied to AES implementations and targeting data caches. The authors give similar warnings as Bernstein regarding table lookups and conditional branches, an example of the latter being with RSA "when exponentiation is performed using a precomputed table of small powers" [42, Sec. 6]. Indeed in more parallel work, Percival [44] carried out such an attack on RSA, an L1 data cache-timing attack on Intel HyperThreading architectures. Percival concludes with similar mitigation strategies for cryptography applications: "the code path and sequence of memory accesses are oblivious to the data and key being used" [44, Sec. 7].

In summary, the combined results and recommendations of Bernstein [11], Osvik et al. [42], and Percival [44] led to the traditional constant time model as we know it today. In this light, these recommendations culminate in the 2006 design of Curve25519, which Bernstein states "avoids all input-dependent branches, all input-dependent array indices, and other instructions with input-dependent timings" [12, Sec. 1]. Subsequent work supports the model and continues to make it explicit. For example, the NaCl library has the design

goal of "No data flow from secrets to load addresses" and "No data flow from secrets to branch conditions" [14, Sec. 3].

The traditional model thus operates under a simple but powerful principle: the sequence of memory addresses accessed during cryptographic operations must be identical regardless of secret values. This includes both data addresses (no secret-dependent array indexing) and code addresses (no secret-dependent conditional branches). From the offensive perspective, model-wise the attacker sees all addresses accessed while the cryptography application executes, both data and code. As we later discuss in Section 3, modern compilers have become quite skilled in recognizing related constant-time programming patterns and instead emitting branches (see [29] from 2016 and [47, Sec. 2] for a wider 2025 survey), and the model does not consider the latency of instructions being potentially operand dependent, both of which can violate the guarantees of the traditional constant time model.

2.3 The Intel Scatter-Gather Model

Percival's work resulted in CVE-2005-0109, where Intel contributed a patch[1] to OpenSSL (0.9.7h, October 2005) for hardening against timing attacks in the case of modular exponentiation. The case is non-trivial, since high-speed implementations of Diffie-Hellman, RSA, and DSA typically use a square-and-multiply variant of modular exponentiation with memory-resident table lookups, costly to accomplish in constant time w.r.t. the traditional model. Intel introduced a new software implementation strategy, leveraging "scatter-gather" to transpose data across cache lines (scatter) then assemble the data with a series of secret-dependent table lookups (gather) that have lower than cache line granularity. Data cache attacks at the time were only exploiting cache set access, with no published method to determine access within a single cache line (intra-cache-line access).

However, this approach flouts the traditional constant time model and disregards previous advice explicitly disallowing intra-cache-line lookups. All three of the seminal works that led to the traditional constant time model warn against such behavior. Percival calls it "dangerous at best" [44, Sec. 7]. Bernstein documents the related cache bank problem [11, Sec. 15], and Osvik et al. worry it "leaks information on memory accesses with resolution better than [a cache line]" [42, Sec. 3.7].

Undeterred by academic advice, Brickell [20] (Intel) explicitly articulated the security model in an invited talk at CHES 2011, stating that Intel recommends "No secret key or data dependent: memory access (at coarser than cache line granularity), code branching." Yet Intel continued to develop and push the scatter-gather technique, for example, 2012 work by Gueron [23] (Intel) on efficient modular exponentiation.

To summarize the Intel model, the fundamental assumption is that if all accesses fall within the same sequence of cache lines, the cache behavior, and thus the timing, will be identical regardless of the specific location accessed within those lines. The scatter-gather technique organizes data so that all possible accesses for a given operation map to the same sequence of cache lines, ensuring uniform cache behavior. As techniques to glean intra-cache-line timings via cache bank conflicts later emerged [13, 59], these days the scatter-gather technique is no longer en vogue and the Intel model is understood to be significantly weaker than the traditional constant time model.

2.4 The OpenSSL Security Policy

The OpenSSL project has developed a pragmatic threat model that focuses on the capabilities of attackers rather than properties of the source code. Prior to the Heartbleed vulnerability in 2014 (CVE-2014-0160), OpenSSL had no explicit release strategy or security policy. The catastrophic impact of Heartbleed, a simple out-of-bounds read that exposed sensitive data from millions of servers, forced a fundamental reassessment of the project's approach to security [55].

Until 2019, OpenSSL would selectively issue CVEs for various types of timing side channels (excluding physical attacks such as power analysis [34], electromagnetic analysis [48], fault injection [16], hardware faults [31], data remanence [24], etc.), including those requiring colocation, such as the CacheBleed attack [59] exploiting cache bank conflicts targeting "constant time" RSA in OpenSSL (CVE-2016-0702), a Flush+Reload attack [58] on "constant time" P-256 arithmetic [45] in OpenSSL (CVE-2016-7056), or Controlled Channel attacks [57] on DSA and ECDSA [56] in OpenSSL (CVE-2018-0734, CVE-2018-0735). However, the disclosure process was subjective and based on undocumented aspects such as perceived risk. This evolved further with the advent of speculative execution attacks like Spectre [32] and Meltdown [36] in 2018, leading to the current security policy[2] that explicitly defines which attack classes warrant CVEs and security bulletins.

The current policy attempts to strike a balance between security and practicality. OpenSSL recognizes that all side channel threats are important, but issuing CVEs requires significant project resources and creates downstream burdens, particularly for FIPS-certified versions that cannot be easily modified. Their model therefore prioritizes attacks that can be executed by less capable attackers, i.e., those that can be mounted remotely or with limited access and resources. The change in fact represented a major shift in the security policy of the project, since attacks requiring colocation are not purely remote, therefore no longer captured by the model. This excludes the vast majority of microarchitectural attacks that require local spy processes, starting from the seminal work by Percival [44].

[1] https://github.com/openssl/openssl/commit/ecb1445ce2df

[2] https://openssl-library.org/policies/general/security-policy/

To summarize, OpenSSL explicitly excludes certain attack classes from warranting security bulletins and CVEs, including same-physical-system side channels requiring colocation, and physical side channels such as those mentioned above. The model assumes the offensive role and focuses on how attackers gather data and potentially exploit vulnerabilities.

3 Analysis: Constant Time Models Compared

The evolution of constant time models highlights the struggle between theoretical security guarantees and practical implementation constraints. Each model makes different trade-offs, and history has shown that deviations from the most conservative approach consistently introduce vulnerabilities. Table 1 systematizes these features, guarantees, constraints, trade-offs, deviations, and risks. This section consists of a discussion centered around the contents of Table 1.

3.1 Model-Specific Analysis

Program Counter. This model assumes that instruction timing is predictable and data-independent. While largely suitable for simple embedded systems without caches, the model immediately fails when applied to large classes of modern processors. Three seminal attacks invalidated this model: (i) the AES data cache attack by Osvik et al. [42], demonstrating that data cache attacks could reveal the complete granular memory access pattern to S-boxes through timing analysis; (ii) the RSA data cache attack by Percival [44], a practical attack on HyperThreading architectures leading to complete private key recovery via highly granular L1 data cache timing data; (iii) the AES data cache attack by Bernstein [11], a purely statistical attack focusing on threat models involving remote attackers with only coarse timing data. The fundamental flaw in the model was defining table lookups with secret indices as safe, an assumption that fails when applied to the vast majority of COTS processors today that typically feature several layers of cache memory. The model's validation approach of analyzing PC sequences indeed allows verification that no secret-dependent branches occur, yet fails to capture the threat of secret-dependent data access.

Traditional. The main focus of the traditional constant time model is on the source code itself. The model considers only a single seemingly simple requirement: all addresses are public, the consequence being that all code must ensure no branching on secrets and no data access using secret indices. These features are typically accomplished with a combination of side channel friendly algorithm selection and (now) well-understood constant time programming paradigms. Regardless of its theoretical soundness, the traditional model faces many practical challenges.

First, compiler optimizations can silently counteract constant time programming patterns. Work by Kaufmann et al. [29] demonstrated how constant time source code for Curve25519 became variable time after compilation. These implementations contain several multiplications with public curve-related constants, and Microsoft's MSVC 2015 toolchain expanded one such multiplication by a 32-bit constant inside a subroutine using a branch on the sign of the second operand, since the upper 32 bits of said constant are known to be zero a priori. The authors outline a method for complete private key recovery leveraging this leak, using modest computation power.

Second, certain instructions have input-dependent timing. An excellent example is early-termination multipliers on ARM processors [22]. Multiplication unit designs in older 32-bit ARM chips leveraged smaller parallel multipliers that would exit early when encountering leading zero bytes in one of the operands. This meant the latency of a multiplication instruction would vary between 2–5 clock cycles, depending on the number of leading zero bytes. A more recent example is the KyberSlash attack [15]. Many Kyber implementations feature an integer division by a small public constant, and expect the compiler to either replace that with a multiplication by a constant or a small number of integer shifts and integer additions. Yet some compilers with appropriate optimization flags would faithfully emit an integer division instruction, which is notoriously variable time on Intel architectures. KyberSlash was both timely and impactful, discovered around the time of NIST standardization and applying to a large portion of software implementations, including the reference implementation by the Kyber authors themselves. It continues to resurface, e.g., CVE-2026-22705 in RustCrypto's ML-DSA implementation.

In summary, despite its age, the traditional constant time model has stood the test of time and proves to be the de facto standard for secure implementations.

Intel Scatter-Gather. Intel's constant time model attempts to balance security and performance. It is conceptually similar to the traditional constant time model, yet allows data flow from secrets to addresses at cache line granularity [20]. Put another way, in the traditional model where all addresses are public, in the Intel model all addresses sans the least significant $\log_2(\delta)$ bits are public (where δ is the number of bytes in a cache line). This relaxation was based on the assumption that accesses within the same cache line would have fixed timing. At least three works demonstrate concrete failure of the Intel model.

First, Bernstein and Schwabe [13] presented empirical data that showed data-dependent timing variations by artificially inducing cache bank conflicts. While the experiment in question targeted strawman code and not actual cryptography code, the fundamental microarchitectural effect was clear from the empirical data.

The CacheBleed attack built on that proof-of-concept soon thereafter [59]. The code targeted by CacheBleed was in fact a contribution from Intel as a countermeasure in OpenSSL to the attack by Percival [44]. This scatter-gather code used

Table 1: Constant time models compared across rules, scope, validation, and documented failures. Every relaxation of the traditional model has eventually been exploited.

Model	Year	Memory Access Rules	Control Flow Rules	Protocol-Layer Scope	Validation Approach	Why it Failed
Program Counter	2004–2005	Allows data flow from secrets to load addresses	No data flow from secrets to branch conditions	Crypto primitives only	Differential analysis of PC sequences	Data cache timing attacks
Traditional CT	2004–2006	No data flow from secrets to load addresses	No data flow from secrets to branch conditions	Crypto primitives only	Static code analysis	Compiler optimizations, variable-time instructions
Intel Scatter-Gather	2005	Allows data flow from secrets to load addresses (within cache lines)	No data flow from secrets to branch conditions	Crypto primitives only	Microarchitectural assumptions	Cache bank conflicts (CacheBleed)
OpenSSL	2019	Context-dependent restrictions	Context-dependent restrictions	Protocol aware	Threat modeling, empirical data	Limited threat model scope
BoringSSL	2025	Allows data flow from secrets to load addresses (variable-length inputs)	Allows data flow from secrets to branch conditions (variable-length inputs)	Crypto primitives only	Static code analysis, formal methods	This work

secret-dependent table lookups within cache lines during modular exponentiation, which the CacheBleed attack exploited due to timing variations caused by cache bank conflicts. The authors were able to recover an RSA private key using a modest number of signature queries.

Lastly, even secret-dependent control flow within a cache line is not a secure strategy. The PortSmash attack (CVE-2018-5407) exploited port contention in Simultaneous Multithreading (SMT) architectures (such as Intel's Hyper-Threading) to leak control flow [5]. One of the experiments the authors conducted targets code with multiple branches within a single cache line, and PortSmash is still able to distinguish between the branch targets [5, Sec. 4].

In summary, cache bank conflicts prophetically invalidated Intel's constant time model for data, and port contention accomplished the same for code. The Intel model nevertheless serves as an exemplary case study of why deviating from the traditional constant time model carries significant risk.

OpenSSL. OpenSSL's pragmatic, data-driven approach focuses on attacker capabilities and empirical data rather than source code analysis. Hardware-based side channels and any attack requiring co-location (e.g., most microarchitecture attacks) are out of scope. With these restrictions, OpenSSL is hoping to exclude local attackers from their model, yet retain remote attackers. This model fails in predictable ways (e.g., the vast majority of cache attacks), yet also fails in surprising ways, discussed below.

Weiser et al. [56] discovered several vulnerabilities in multiprecision integer arithmetic libraries used by cryptography libraries. Their "V8" vulnerability applied to a division function as part of the extended Euclidean algorithm, their main application being to DSA signing. The leak was due to data structure resizing, and was limited to a single top-most bit of the secret. Their "V9" vulnerability is in the same algorithm, a conditional negation that leaks the top-most bit of a secret inverse. These vulnerabilities are outside the scope of the OpenSSL security policy because (i) they are unlikely to be observable remotely; and (ii) it is unclear from the theory perspective how a remote attacker can effectively utilize such a small leak. Hence, while the vulnerabilities are certainly valid from the academic perspective, OpenSSL's security policy discards them.

A second example is the obscure key formats studied by Pereida García et al. [46]. The code loading these keys neglected to configure the private keys to ensure constant time code paths within the library, which led to a single trace private key recovery attack with local L1 data cache timings. While that is clearly outside OpenSSL's model, the same leak revealed information about the top set bit of the private key, potentially measurable in remote scenarios. Yet it is again unclear from the theory perspective how a remote attacker can effectively utilize such a small leak on fixed bits. Hence, while this vulnerability is certainly valid from the academic perspective, it is out of scope for OpenSSL's security policy.

A third example is the LadderLeak attack by Aranha et al. [8], capable of recovering ECDSA private keys with less than one bit of nonce leakage per signature, but requiring local cache timings to detect branch targets. Coupled with the large data requirement of the attack (needing approximately 2^{33} signatures for key recovery), LadderLeak falls outside OpenSSL's security policy, requiring local access and significant data, time, and resources. Yet again, this attack is certainly valid from the academic perspective.

A final and recent example is the SLasH-DSA attack by Boy et al. [19], a fault attack on hash-based PQC as implemented in OpenSSL 3.5. The authors utilize the RowHammer technique [31] to induce memory read faults while querying digital signatures. Hardware faults are indeed explicitly excluded in OpenSSL's security policy, and furthermore RowHammer requires local access. Yet again, this attack is certainly valid from the academic perspective.

An interesting model-related case study is the OpenSSL vulnerability linked to CVE-2025-27587, which led to conflict between OpenSSL and the wider security community. The vulnerability leaks part of the ECDSA nonce in OpenSSL's implementation on PowerPC architectures, yet the timing distributions are highly overlapping and it remains unclear how to separate them. Furthermore, the experiment data is with local timings, while remote timings exacerbate the situation from the attacker perspective, increasing the statistical overlap. OpenSSL typically manages its own CVEs, but in this case MITRE chose to issue its own against OpenSSL, who disputed the vulnerability because it fell outside their written security policy, for similar reasons as the LadderLeak

attack.

In summary, OpenSSL's model has the best and worst of both worlds. On one hand, it flippantly excludes large classes of attacks that the security community historically recognizes as valid, i.e., requiring co-location and local execution. On the other hand, their data-driven approach potentially scopes in threats that are typically not captured in other models, where "crypto in a vacuum" is the norm, source code analysis stops at the low-level crypto, and vulnerabilities in callers to the crypto are another layer's problem, even if attackers care little about such semantics. We elaborate in the next subsection.

BoringSSL. We defer this discussion to the end of the paper.

3.2 Constant Time Models: Gaps

Following the previous analysis surrounding Table 1, in this subsection we expand the discussion to consider areas where the models may not perform as expected. These issues are pervasive, and represent a security concern for all of the models.

Protocol-layer leakage. A critical limitation of most constant time models is their focus on cryptographic primitives and low-level arithmetic in isolation, ignoring how these primitives behave when integrated into protocols. This gap has led to numerous attacks that bypass even high-assurance and high-security cryptography implementations by exploiting protocol-level leaks. While we expand later in Section 4, here are a few seminal works that highlight this gap.

The Million Message Attack (MMA) by Bleichenbacher [17] demonstrated that implementing RSA PKCS#1 v1.5 (RFC 2313 [27], 1998) carries significant security risk. Implementations tend to leak whether decryptions produce valid padding or not, either directly through error messages or indirectly through timing. These leaks enable adaptive chosen ciphertext attacks to decrypt without the private key. Manger [39] subsequently showed that RSA-OAEP and PKCS#1 v2.0 (RFC 2437 [28], 1998), designed to resist MMA, remained vulnerable to similar oracle attacks through timing leaks in their more complex padding validation.

On the symmetric side, Vaudenay [54] targeted implementations of CBC mode for block ciphers with PKCS#7 padding, where timing differences in padding validation expose the plaintext. While Vaudenay described several different oracles, the general area of padding oracles emerged only later, encompassing both the attacks by Bleichenbacher and Vaudenay under the general umbrella that validating padding applied to plaintext is a dubious security practice. The work extends naturally to hardware implementations [10].

These seminal results reveal a fundamental class of vulnerabilities: protocols requiring conditional processing of padding or error conditions create timing channels that propagate beyond the cryptographic primitive layer.

Empirical validation. Consulting the rightmost column of Table 1, the failures across all models highlight the critical importance of empirical validation, both through direct measurement of execution latency on target hardware and through analysis of compiled binaries. Even when source code follows constant-time programming practices, the underlying hardware may still exhibit input-dependent timing variations, and mainstream compilers actively undermine constant time guarantees.

The tool dudect [49] takes a purely empirical approach, which utilizes classical leakage detection methods. It executes the target function with two classes of input data, collects cycle-accurate timings, and applies statistical tests to determine whether the resulting distributions are distinguishable. Because this approach requires no hardware model and no source code annotations, any timing variation observable on the target platform can in principle be detected, provided the effect is large enough relative to measurement noise. However, the tool cannot pinpoint the source of a detected leak, and subtle violations may require millions of measurements to reach statistical significance.

Constant time violations can originate in the toolchain rather than the hardware. The Clangover vulnerability (CVE-2024-37880, Antoon Purnal) in the ML-KEM reference implementation demonstrates a case where the source code used a constant-time conditional assignment, but modern Clang compiled it into a conditional branch on secret data [9]. The vulnerability is particularly stealthy because source-level analysis tools correctly report the code as constant-time, since the violation exists only in the compiled binary. LLVM recently implemented support for source code annotations that preserve select constant-time properties through optimization passes[3], demonstrating that compiler designers are aware of the issue, yet the narrow scope and opt-in nature make this approach inherently fragile. Arranz-Olmos et al. [9] further showed that popular decompilers compound the problem by performing optimizations that eliminate constant time violations during analysis, causing decompile-then-analyze approaches to miss vulnerabilities that exist in deployed binaries. In summary, this is a challenging compiler security gap since developers write correct constant-time code, but the toolchain silently transforms it into vulnerable binaries, and standard analysis techniques fail to detect the resulting violations.

To approach this challenge, TIMECOP uses Valgrind's Memcheck infrastructure to detect potential constant-time violations through dynamic binary analysis. It marks secret data as uninitialized via Valgrind client requests, causing Memcheck to flag any secret-dependent branch conditions or memory accesses encountered during execution. Because this analysis operates on compiled binaries, it can detect violations introduced by compiler optimizations that are not visible at the source level. TIMECOP originated as prototype patches

[3] https://blog.trailofbits.com/2025/12/02/introducing-constant-time-support-for-llvm-to-protect-cryptographic-code/

to the SUPERCOP benchmarking framework, with a leakage model limited to branching and memory accesses. From the other direction, patches to Valgrind added support for detecting potential variable-time instructions operating on secret data. Finally, the KyberSlash work [15] combines both approaches to allow the analysis to scale by scoping in more ISAs, extending the list of potentially vulnerable instructions, and enabling tests across a wide spectrum of implementations and architectures.

In summary, the two approaches complement one another, with dudect analyzing end-to-end timing behavior on real hardware, while binary analysis tools like TIMECOP pinpoint the location of the leak.

4 An Offensive Security Methodology

Most of the constant time models in Section 3 stop at the crypto layer, yet the protocols using these primitives often operate on secret data well beyond that boundary. Specifications, standards, and protocols are typically written with correctness and interoperability as primary goals, not constant time execution. They often require operations that create natural timing channels, such as padding schemes with conditional processing, error handling with different code paths, and variable-length encodings. No implementation can satisfy both the specification and constant time requirements, because the specification itself mandates contradicting behavior.

This section explores those gaps through various case studies, culminating in an offensive methodology which identifies standards whose flexibility or complexity creates said conflict, then tests implementations for any leaks predicted by the analysis. We organize the case studies by the specification at fault, not by the models from Table 1. Which constant time model a given attack violates depends on the threat model, while the specification does not. Each case below is an instance of protocol-layer leakage.

4.1 Case Studies

Specifications have been a steady source of timing leaks since (at least) the seminal attacks discussed in Section 3, with new instances appearing across protocols and implementations in the decades since. Standards and specifications play a significant role in proliferating these vulnerabilities, and in this section we investigate that relationship through several recurring cases (e.g., error handling, padding validation, key persistence formats, encoding of secrets) where compliant implementations undermine primitive-level constant time guarantees with system-level violations.

Public key padding oracles. As previously discussed, the way that PKCS#1 v1.5 (RFC 2313 [27]) specifies the handling of malformed ciphertexts lends itself to vulnerability, first exploited by Bleichenbacher [17] in 1998. Each fix has produced a new variant, several of those which we now discuss.

Jager et al. [26] observed that the XML encryption standard specified its own error handling requirements for RSA PKCS#1 v1.5, independently recreating the same class of oracle in a completely different protocol (CVE-2011-2487). This is despite the W3C having published the specification in 2002, four years after Bleichenbacher's original disclosure.

Kupser et al. [35] subsequently automated the attack, finding that several implementations (CVE-2015-0226, etc.) remained vulnerable even after deploying mitigations. The authors circumvent fixes via wrapping techniques that relocate encrypted elements within the XML document tree to bypass integrity checks.

The ROBOT attack [18] (CVE-2017-6168, etc.) demonstrated that even the TLS 1.2 specification's own countermeasure against Bleichenbacher was itself a source of new oracles. The mitigation requires servers to generate a random premaster secret on padding failure and continue the handshake. The work highlights that ensuring the success and failure codepaths are indistinguishable is indeed difficult to achieve in constant time.

Ronen et al. [50] identified nine distinct attack variations across the TLS ecosystem, each exploiting different leakage sources in implementations with mitigations (CVE-2018-12404, etc.). Each fix to a specific Bleichenbacher variant introduced new, subtly different oracles exploited by the authors. This shows that when a specification is fundamentally incompatible with constant time goals, patches merely shift the leak.

Symmetric key padding oracles. The 2002 padding oracle attack by Vaudenay [54], mentioned in Section 3, is the symmetric key counterpart. The specification-level root cause (protocols that require the receiver to act on plaintext properties before integrity can be verified) has proven equally persistent, with instances across SSH, XML encryption, and TLS.

Albrecht et al. [3] analyzed the SSH Binary Packet Protocol (BPP, RFC 4253 [38]), finding that it encrypts the packet length field within the first CBC ciphertext block (CVE-2008-5161, etc.). The receiver must decrypt this block to see how many additional bytes to read. The specification makes it impossible to authenticate the MAC before parsing this field, a parallel to the (legacy) TLS MAC-then-encrypt ordering. Observing how many bytes the server reads before responding allows an attacker to infer the decrypted length field, yielding a practical chosen-ciphertext attack enabled through the standard.

Jager and Somorovsky [25] showed that the XML encryption specification's use of CBC mode created a similar oracle, exploiting the server's error handling when decrypted plaintext contained malformed character encodings. This vulnerability (CVE-2011-1096) originated from the XML parsing layer rather than cryptographic padding validation.

Previously mentioned in the public key context, Jager et al. [26] further showed that XML encryption combines the two oracles on the symmetric key and public key sides. CBC malleability allowed an attacker to test candidate RSA session keys by observing whether the resulting symmetric decryption produced valid plaintext, enabling Bleichenbacher attacks without relying on error messages or timing from the RSA layer itself.

The Lucky Thirteen attack [6] (CVE-2013-0169) extended the CBC work to (pre 1.3) TLS with a timing attack on the MAC computation itself. The number of hash iterations depends on the data length after stripping padding (mandated by the specification), leaking whether a modified ciphertext contained valid padding through timing. The attack affected virtually every TLS implementation because the leak originated in the specification's ordering of operations, not in any particular cryptographic primitive. Apecechea et al. [7] subsequently showed that cache-based timing attacks could amplify these differences with local access.

Encoding leakage. Beyond padding validation and error handling, specifications that define key persistence formats are another source of timing leaks. Encodings such as hex and base64 (required by any library that loads cryptographic material from PEM files or similar formats) typically use a lookup table indexed by the ASCII value of each input character, easily leading to secret-dependent memory access, and furthermore entirely outside the constant time cryptographic primitive boundary.

Sieck et al. [51] demonstrated this concretely, showing that base64 decoding in OpenSSL, mbedTLS (CVE-2021-24119), Botan (CVE-2021-24115), Mozilla NSS, and GNU Nettle used secret-dependent memory accesses when loading RSA private keys from PEM files. Observing a single PEM key loading event allowed partial recovery of RSA private key material, leakage that no amount of constant time arithmetic in the cryptographic primitive layer can mitigate. Subsequent work increased the leakage with finer spatial resolution through advanced microarchitectural techniques [52].

Notably, BoringSSL had proactively addressed this leak in 2017 before either attack was published, implementing constant time base64 decoding by replacing table lookups with arithmetic range comparisons. Quoting the commit message:[4] "PEM files sometimes carry private keys so, in principle, we'd probably prefer not to leak the contents when we encode or decode them." BoringSSL chose to extend constant time techniques to key decoding well before any published attack, acknowledging that decoding can leak secret material and applying meaningful mitigations beyond the cryptographic primitive boundary.

Canonicalization leakage. Beyond character-level encoding, some specifications also mandate canonical representations for secret values. This might include stripping leading zeros, conditionally prepending sign bytes (e.g., for DER), or enforcing a particular integer representation. Compliant implementations would then perform variable-length processing on secret data, potentially leaking the length of the secret (confidential in many cases).

A good example is the SSH protocol (RFC 4253 [38], discussed earlier in the separate context of BPP), where the standard mandates that the Diffie-Hellman shared secret be represented as an MPINT. This data type is defined in RFC 4251 [37] with the explicit requirement that "unnecessary leading bytes with the value 0 ... MUST NOT be included." RFC 8731 [2] (which defines Curve25519 and Curve448 key exchange for SSH) explicitly acknowledges this weakness in its Security Considerations section: "The way the derived MPINT binary secret string is encoded before it is hashed (i.e., adding or removing zero bytes for encoding) raises the potential for a side-channel attack."

The Raccoon attack by Merget et al. [40] exploited similar canonicalization in (pre 1.3) TLS, where DH(E) cipher suites require stripping leading zeros from the premaster secret before key derivation. The attack affected several security-critical libraries (CVE-2020-1968 in OpenSSL, CVE-2020-12413 in Mozilla NSS, etc.), not due to shared code, but because they implemented the same specification. Subsequent work [4] demonstrated that microarchitectural timings from local attackers could amplify the leakage, extending it to CMS and S/MIME contexts. The Raccoon authors confirmed that SSH suffers from the same specification-level flaw (discussed previously), but could not exploit it due to SSH's strict use of fresh keys per session [40, Sec. 8]. Exploitability depends on the protocol context, even when the underlying specification-level vulnerability is identical.

4.2 The Offensive Methodology

In each of the previous case studies, the specification requires an operation on secrets that is inherently difficult to realize in constant time. Implementations that comply for interoperability leak in ways that no amount of constant time arithmetic or programming paradigms in the cryptographic primitive layer can prevent. We now distill this into a methodology.

Step 1: Identify specs with problematic operations. Locate operations a specification requires on secret data that conflict with constant time goals. For example, padding schemes with conditional processing (PKCS#1 v1.5), error conditions that mandate distinguishable responses (CBC), variable-length encodings (ASN.1, DER, PEM), canonicalization of keys (byte stripping), etc.

Step 2: Analyze implementation strategies. Investigate how systems handle these problematic operations. For example, they might knowingly tolerate the leak, deploy countermeasures that might expose corner cases, or compromise on compatibility to achieve better timing attack resistance. Implementation strategies may also vary across architectures and

[4] https://boringssl.googlesource.com/boringssl/+/536036ab

compiler toolchains, so the same code can exhibit different leakage characteristics depending on the target platform.

Step 3: Confirm the analysis. Run experiments against these implementations to confirm or refute the predicted leak. The systematization in Section 3 suggests that multiple analysis techniques apply in tandem. The scope, validation, and failure considerations in Table 1 highlight static methods, dynamic analysis, and targeted timing measurements as examples of analytical methods. Casting a wide net allows leveraging a fuller spectrum of evidence, from more formal techniques all the way through empirical timings.

In summary, different implementations of the same specification tend to leak in similar ways, not because they share code, but because they share the specification. Diverse analysis methods complement one another, and our methodology is sound largely due to inheriting more of the strengths of these methods and less of the inherent limitations. We validate our methodology in the remainder of this paper.

5 Analysis: Private Key Loading

Applying said methodology, this case study examines how a seemingly minor specification detail, the encoding of elliptic curve private keys, creates systematic timing leaks across implementations. By tracing the evolution from early standards through modern implementations, we demonstrate how specification-level decisions made decades ago continue to create meaningful timing leaks today.

5.1 Elliptic Curve Cryptography Standards

The encoding of elliptic curve private keys dates to the Standards for Efficient Cryptography Group's specification SEC 1 [1], published in 2000, which defined the `ECPrivateKey` syntax with the private key encoded as a fixed-length octet string derived from the Integer-to-Octet-String-Primitive (I2OSP). RFC 5915 [21] later echoed this requirement, specifying that the `privateKey` field must be of length $\lceil \log_2(n)/8 \rceil$ bytes, where n is a public constant (the curve order). For example, with the NIST standard curves P-256, P-384, and P-521, this mandates exactly 32 bytes, 48 bytes, and 66 bytes, respectively, regardless of the numeric value of the scalar, and any private key with a small value must be zero-padded to the full length. This fixed-length requirement happens to enable constant-time processing, as decoders can unconditionally read a publicly known number of bytes and convert them to an internal BIGNUM representation using fixed-width operations.

In practice, implementations are not always so rigid. The `privateKey` field is a DER-encoded OCTET STRING, but the value it carries is semantically an integer, and internally many implementations will convert it to a BIGNUM. DER encoding of INTEGER values requires minimal-length representation, and per X.690 rules, the output must use the smallest number of octets needed, canonicalization that strips leading zero bytes. While OCTET STRING and INTEGER are distinct ASN.1 types with different encoding rules, the boundary might blur in practice. BIGNUM libraries often strip leading zeros on import regardless of the source type, in which case subsequent serialization routines naturally produce minimal-length output. Ecosystem-wise, the result would be a population of existing keys with private key fields shorter than the specification mandates, encoding the effective (bit, byte) length of the private key (secret integer scalar) directly in the serialized representation. Implementations could choose to reject these non-conformant keys, yet lessons from padding oracle attacks tell us this is more difficult to do securely than it might seem.

Applying the Section 4 methodology, the first step is to confirm that the target implementations accept non-conformant keys at all. We generated P-384 private keys with effective bit lengths ranging from 128 to 384 in multiples of 8, encoding each as a truncated (leading-zeros-stripped) octet string within an otherwise valid `ECPrivateKey` structure. Both OpenSSL and BoringSSL accept every such key without error, confirming that the threat surface exists and justifying deeper analysis of the loading paths.

In summary, this suggests canonicalization leakage (previously discussed in Section 4), where a specification defines a fixed-length format, but implementations that tolerate variable-length inputs potentially introduce vulnerabilities outside the cryptographic primitive boundary.

5.2 Implementation Analysis

Having confirmed that both implementations accept variable-length keys, we now examine their respective loading paths to understand where secret-dependent processing occurs. We also discuss adversarial scenarios and the high level impact to security.

OpenSSL. EC private key loading follows the path `d2i_ECPrivateKey` → `EC_KEY_oct2priv` → `ossl_ec_key_simple_oct2priv`. Two operations process the variable-length private key material. First, OpenSSL's ASN.1 runtime (`d2i_EC_PRIVATEKEY`) decodes the DER structure, populating an `ASN1_OCTET_STRING` whose length field reflects the encoded size of the private key. For a non-conformant key this is shorter than the curve order, and the parser performs correspondingly less work. Second, `ossl_ec_key_simple_oct2priv` passes that variable-length buffer and its length directly to `BN_bin2bn` to construct a BIGNUM holding the private key, which is not a constant-time function. It processes exactly the number of bytes present, so the amount of work varies with the encoded length of the private key.

BoringSSL. BoringSSL's loading path is architecturally similar. The function `ec_key_parse_private_key` handles both

ASN.1 parsing and key construction in a single routine, using BoringSSL's CBS ("crypto byte string") interface for DER decoding. As with OpenSSL, two operations process the variable-length private key material. First, `CBS_get_asn1` extracts the OCTET STRING from the DER structure, parsing a TLV whose length field encodes the size of the private key material. For a non-conformant key, this length is shorter than the curve order, and the parser performs correspondingly less work. Second, the extracted bytes are passed to `BN_bin2bn` to construct a BIGNUM holding the private key, which as in OpenSSL processes exactly the number of bytes present with no padding to a fixed width. Both leakage points (the ASN.1 length parsing and the BIGNUM conversion) operate on the actual encoded length rather than the expected (public) curve order length.

Security implications. Some services amortize the cost of key loading using daemons that load keys once at startup and serve from memory. Others load per connection, a good example being the classical Unix service dispatcher inetd ("internet service daemon") where (typically) each incoming connection spawns a fresh process that parses its keys before serving the request. In the latter case, an adversary can force an arbitrary number of key load observations simply by opening connections.

For this leak, increasing samples does not provide new information in an unbounded way. The leak reveals fixed information, the byte length of the encoded scalar, which is a property of the long-term private key, not the session. Additional queries only increase confidence. To give a worked example, probabilistically $1/256$ keys will be a single byte shorter than a full-length key, and in that case it leaks 8 bits ("the top byte is zero"). So while this leak indeed leads to an information-theoretic reduction in confidentiality, it cannot be used to extract a growing fraction of the fixed secret.

The utility of the leak from the adversarial perspective is also dubious. Continuing the worked example, any cryptanalytic attack that benefits from a short fixed scalar can simply assume one and proceed, without any leak or side channel. Such an attack succeeds $1/256$ of the time. While the leak is real and any model violations are real, we are not aware of any exploit, and see little to no practical relevance in the leak.

5.3 Results

We constructed a harness that loads each PEM-encoded key file into memory once, then repeatedly parses it from an in-memory buffer. The timed region brackets a single call to `PEM_read_bio_PrivateKey` (PEM base64 decoding, DER parsing, and key object construction), with no disk I/O in the measurement path. The harness runs on a quad-core Intel i5-7500T at 2.70GHz with frequency scaling disabled, equipped with 16GB of DDR and running Debian 12.13 "bookworm." This CPU does not feature HyperThreading. It loads each key 1,000 times, each load timed in CPU clock cycles. We linked the same harness against both OpenSSL and BoringSSL, using a corpus of P-384 keys at 33 distinct bit lengths (128 to 384, multiples of 8) with equal sample counts per class. This uniform distribution is artificial (a randomly generated P-384 scalar has probability roughly 255/256 of occupying the full 48 bytes), but allows characterizing the timing distribution by class.

Table 2: Statistical comparison of key load timing leakage for OpenSSL, BoringSSL (unpatched and hedged), NIST curve P-384. MI and permutation z-scores characterize the channel; Cohen's d and Bayesian sample counts characterize the adversary's decision problem under a 255/256 prior for full-length keys.

Metric	OpenSSL (stock)	BoringSSL (stock)	BoringSSL (patched)
MI, 33-class (bits)	0.0031	0.6104	0.0055
MI permutation z-score	249	57,861	434
Cohen's d, binary	0.098	1.11	0.008
Binary MI (bits)	0.0003	0.037	0.0006
Binary MI z-score	151	19,021	250
Samples to $P > 0.5$	$\approx$1,000	$\approx$10	$\approx$100,000
Samples to $P > 0.999$	$\approx$10,000	$\approx$50	$\approx$1,000,000

Figure 1 shows the empirical CDFs of loading time grouped by encoded byte length. The OpenSSL distributions (left) overlap almost entirely, with no visually discernible separation. The BoringSSL distributions (center) separate visibly, with a staircase structure suggesting leakage at byte granularity. Table 2 summarizes the statistical comparison across three complementary metrics. The mutual information $I(\text{bitlength};\text{timing})$ across all 33 classes measures how much key-length information a single timing observation reveals. OpenSSL leaks 0.0031 bits of key-length information per timing observation; BoringSSL leaks 0.6104 bits, approximately $200\times$ more. To test the null hypothesis and establish a noise baseline, we shuffled class labels and recomputed MI over 100 rounds; the observed MI for all configurations exceeds this baseline, confirming the signal is genuine, though the magnitude differs significantly.

The 33-class MI characterizes the channel, but the adversary's task is rather to decide if a key is full length or not. Partitioning into 384-bit keys versus all shorter encodings, Cohen's d (where values near zero indicate overlap and values above one indicate separation) is 0.098 for OpenSSL, meaning the distributions are nearly indistinguishable, and 1.11 for BoringSSL, meaning the distributions barely overlap. This interpretation is consistent with the binary MI permutation z-score (standard deviations above the null hypothesis), which are 151 for OpenSSL and 19,021 for BoringSSL. All reject the null hypothesis, but by this metric the BoringSSL leak is two orders of magnitude stronger than that of OpenSSL.

For a randomly generated P-384 key, the probability of the scalar having full byte length is 255/256, a somewhat low

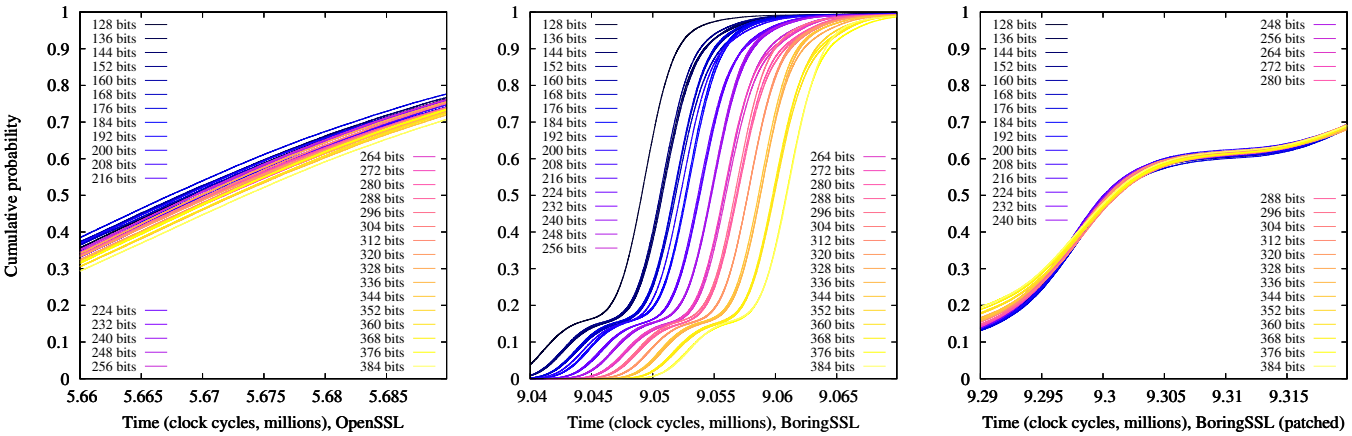


Figure 1: Empirical CDFs of EC private key loading time (P-384) grouped by effective byte length. OpenSSL (left): distributions overlap almost entirely. BoringSSL unpatched (center): distributions separate visibly, with a staircase structure at byte granularity. BoringSSL patched (right): with the hedge, distributions overlap, similar to OpenSSL.

probability for observing a truncated key. Nevertheless, this true a priori probability reflects the adversary's actual decision problem. For OpenSSL, the adversary requires approximately 1,000 observations to reach a posterior probability $P > 0.5$, and 10,000 for $P > 0.999$. For BoringSSL, averaging roughly 10 observations yields $P > 0.5$, and 50 for $P > 0.999$, two orders of magnitude less effort than OpenSSL.

5.4 Disclosure, Defense, and Discussion

We shared our findings with the BoringSSL security team[5]. The remainder of this section discusses their response, a simple hedge that we prototyped, and the corresponding empirical results.

What BoringSSL could've done. The leakage in BoringSSL spans the entire decode stack, where PEM base64 decoding, DER parsing, and BIGNUM conversion all process variable-length input proportional to the encoded key size. Achieving constant-time behavior across these layers would require invasive changes to the serialization infrastructure, which the implementation currently does not pursue.

As an alternative, we implement a timing hedge internal to `PEM_read_bio_PrivateKey`. On each call, it spins up a second thread and runs the real parse concurrently with that of a fixed, full-length private key, and returns only after both complete (post-join). The observed time is therefore dominated by the slower parse, which (in practice) will approach the time of a full-length parse. We acknowledge this strategy violates most of the models in Table 1, yet the goal of this experiment is to raise the cost for adversaries, not to eliminate all potential leakage.

We then gathered a third dataset from the methodology in Section 5.3 against this patched BoringSSL version. The hedged BoringSSL distributions (Figure 1, right) overlap, visually similar to OpenSSL. Regarding information-theoretic metrics (Table 2), 33-class MI drops from 0.61 to 0.005 bits (a 110× reduction, versus 0.003 for OpenSSL), and binary MI from 0.037 to 0.0006 (versus 0.0003 for OpenSSL). On the adversary's decision problem (short versus full) the hedge improves further. Cohen's d drops from 1.11 to 0.008, below OpenSSL's 0.098, and the adversary now requires roughly 100,000 observations to reach $P > 0.5$ and 1,000,000 for $P > 0.999$, notably four orders of magnitude more effort than against unpatched BoringSSL and two orders of magnitude more than even OpenSSL. Permutation testing for the null hypothesis still detects a signal across all datasets, yet our patched BoringSSL is harder to target than OpenSSL across every Bayesian metric.

What BoringSSL did. During disclosure, BoringSSL acknowledged the leak but declined to react with any code changes. On one hand, rejecting these malformed keys has unknown consequences since the key landscape is not well understood, which is a valid concern from an ecosystem and business perspective. Compounding this, in their view non-conformant keys fall outside their constant time boundary, which they subsequently clarified means constant time only for fixed-width inputs (see the bottom row of Table 1). The leak therefore falls outside their model by construction, and for variable-width inputs it follows that any mitigation that continues to accept malformed keys would also fail to satisfy the traditional constant time model. On the other hand, our data in Table 2 offers an adversary-driven perspective on the vulnerability. Even our simple hedge that clearly still violates the traditional constant time model increases the adversary's cost by four orders of magnitude, and it does so while targeting only the OpenSSL model, the weakest of all in Table 1. Counterintuitively, BoringSSL aims at considerably higher security in their model, yet the leak is measurably stronger

[5] https://marc.info/?l=oss-security&m=176046647932425

than in OpenSSL, which applies a significantly weaker model. In 1772, Voltaire wrote[6] "le mieux est l'ennemi du bien" or *the best is the enemy of the good*. Four orders of magnitude is a quantitative, objective improvement, and we claim that practical hedges have a useful place in security hardening, over arguments about model semantics.

6 Conclusion

This work systematizes the evolution of cryptographic constant time models in security-critical libraries, from early embedded-era assumptions through modern project-specific policies. The historical pattern is consistent, showing every relaxation of the traditional model has eventually been exploited. The traditional model's strength is its simplicity, where all addresses are public, making fewer assumptions about hardware behavior than any of the alternatives.

The gap between what constant time models protect and what specifications assume is a recurring source of vulnerabilities. Padding oracles, encoding leakage, and canonicalization issues all originate in specifications that mandate operations on secrets that are difficult or impossible to realize in constant time. The methodology we distill from these case studies provides a systematic framework for exploring this gap.

Applying the methodology to EC private key loading illustrates the limits of threat model strength as a proxy for implementation security. BoringSSL's constant time model is explicitly stronger than OpenSSL's, in that it aims to protect against local and microarchitectural attackers, while OpenSSL's security policy excludes colocation attacks entirely. Yet on this specific leak, our data from the adversarial perspective demonstrates that OpenSSL is empirically more resistant.

We close with the following joke posted to rec.humor.funny by Wolfgang Sohrt in 1989.

> An engineer, a physicist and a mathematician have to build a fence around a flock of sheep, using as little material as possible.
>
> The engineer forms the flock into a circular shape and constructs a fence around it.
>
> The physicist builds a fence with an infinite diameter and pulls it together until it fits around the flock.
>
> The mathematician thinks for a while, then builds a fence around himself and defines himself as being outside.

A strong threat model on paper means little if the boundaries are drawn too narrowly. Specifications exist in the wild, non-conformant inputs exist in the wild, and attackers do not respect scope declarations. Empirical measurement, not model coverage, is the final arbiter of whether an implementation is vulnerable to timing attacks.

Availability. In support of Open Science and to ensure reproducibility, we released[7] our tooling from the data procurement methodology in Section 5.3 as a freely available research artifact for the benefit of the community. The artifact can be used to reproduce at least Figure 1 (left, center).

[6] Voltaire, "La Bégueule" (1772)

[7] https://gitlab.com/platsec/boringssl-keyload-vuln

References

[1] SEC 1: Elliptic Curve Cryptography. SEC 1, Standards for Efficient Cryptography Group, Sep 2000. URL https://www.secg.org/SEC1-Ver-1.0.pdf.

[2] Aris Adamantiadis, Simon Josefsson, and Mark D. Baushke. Secure Shell (SSH) Key Exchange Method Using Curve25519 and Curve448. RFC 8731, RFC Editor, February 2020. URL https://datatracker.ietf.org/doc/rfc8731/.

[3] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA*, pages 16–26. IEEE Computer Society, 2009. URL https://doi.org/10.1109/SP.2009.5.

[4] Alejandro Cabrera Aldaya and Billy Bob Brumley. HyperDegrade: From GHz to MHz effective CPU frequencies. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2801–2818. USENIX Association, 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/aldaya.

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 870–887. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00066.

[6] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013. URL https://doi.org/10.1109/SP.2013.42.

[7] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information, Computer and Communications*

Security, AsiaCCS 2015, Singapore, April 14-17, 2015, pages 85–96. ACM, 2015. URL https://doi.org/10.1145/2714576.2714625.

[8] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 225–242. ACM, 2020. URL https://doi.org/10.1145/3372297.3417268.

[9] Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Youcef Bouzid, Sören van der Wall, and Zhiyuan Zhang. Decompiling for constant-time analysis. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1): 29–58, April 2026. URL https://doi.org/10.1145/3798201.

[10] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 608–625. Springer, 2012. URL https://doi.org/10.1007/978-3-642-32009-5_36.

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

[12] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. URL https://doi.org/10.1007/11745853_14.

[13] Daniel J. Bernstein and Peter Schwabe. A word of warning. CHES 2013 Rump Session, August 2013. URL https://cryptojedi.org/peter/data/chesrump-20130822.pdf.

[14] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012. URL https://doi.org/10.1007/978-3-642-33481-8_9.

[15] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(2):209–234, 2025. URL https://doi.org/10.46586/tches.v2025.i2.209-234.

[16] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997. URL https://doi.org/10.1007/BFb0052259.

[17] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998. URL https://doi.org/10.1007/BFb0055716.

[18] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of Bleichenbacher's oracle threat (ROBOT). In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 817–849. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/bock.

[19] Jeremy Boy, Antoon Purnal, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. SLasH-DSA: Breaking SLH-DSA using an extensible end-to-end Rowhammer framework. *Proc. Microarchitecture Secur. Conf.*, 2026, 2026. URL https://doi.org/10.46586/uasc.2026.009.

[20] Ernie Brickell. Technologies to improve platform security. CHES 2011 Invited Talk, September 2011. URL https://iacr.org/workshops/ches/ches2011/presentations/Invited%201/CHES2011_Invited_1.pdf.

[21] Daniel R. L. Brown and Sean Turner. Elliptic Curve Private Key Structure. RFC 5915, RFC Editor, June 2010. URL https://datatracker.ietf.org/doc/rfc5915/.

[22] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In Dong Hoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers*, volume 5984 of *Lecture Notes in Computer Science*, pages 176–192. Springer, 2009. URL https://doi.org/10.1007/978-3-642-14423-3_13.

[23] Shay Gueron. Efficient software implementations of modular exponentiation. *J. Cryptographic Engineering*, 2(1):31–43, 2012. URL https://doi.org/10.1007/s13389-012-0031-5.

[24] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60. USENIX Association, 2008. URL http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf.

[25] Tibor Jager and Juraj Somorovsky. How to break XML encryption. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 413–422. ACM, 2011. URL https://doi.org/10.1145/2046707.2046756.

[26] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *Lecture Notes in Computer Science*, pages 752–769. Springer, 2012. URL https://doi.org/10.1007/978-3-642-33167-1_43.

[27] Burt Kaliski. PKCS #1: RSA Encryption Version 1.5. RFC 2313, March 1998. URL https://www.rfc-editor.org/info/rfc2313.

[28] Burt Kaliski and Jessica Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437, October 1998. URL https://www.rfc-editor.org/info/rfc2437.

[29] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting Curve25519-donna built with MSVC 2015. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 573–582, 2016. URL https://doi.org/10.1007/978-3-319-48965-0_36.

[30] John Kelsey, Bruce Schneier, David A. Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *Computer Security - ESORICS 98, 5th European Symposium on Research in Computer Security, Louvain-la-Neuve, Belgium, September 16-18, 1998, Proceedings*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110. Springer, 1998. URL https://doi.org/10.1007/BFb0055858.

[31] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014. URL https://doi.org/10.1109/ISCA.2014.6853210.

[32] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00002.

[33] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_9.

[34] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. URL https://doi.org/10.1007/3-540-48405-1_25.

[35] Dennis Kupser, Christian Mainka, Jörg Schwenk, and Juraj Somorovsky. How to break XML encryption—automatically. In Aurélien Francillon and Thomas Ptacek, editors, *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*. USENIX Association, 2015. URL https://www.usenix.org/conference/woot15/workshop-program/presentation/kupser.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[37] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, RFC Editor, January 2006. URL https://datatracker.ietf.org/doc/rfc4251/.

[38] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, RFC Editor, January 2006. URL https://datatracker.ietf.org/doc/rfc4253/.

[39] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer, 2001. URL https://doi.org/10.1007/3-540-44647-8_14.

[40] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 213–230. USENIX Association, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/merget.

[41] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005. URL https://doi.org/10.1007/11734727_14.

[42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. URL https://doi.org/10.1007/11605805_1.

[43] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002(169), 2002. URL http://eprint.iacr.org/2002/169.

[44] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*, 2005. URL http://www.daemonology.net/papers/cachemissing.pdf.

[45] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 83–98. USENIX Association, 2017. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia.

[46] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. Certified side channels. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2021–2038. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/garcia.

[47] Thomas Pornin. Constant-time code: The pessimist case. *IACR Cryptology ePrint Archive*, 2025(435), 2025. URL https://eprint.iacr.org/2025/435.

[48] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and countermeasures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001. URL https://doi.org/10.1007/3-540-45418-7_17.

[49] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In David Atienza and Giorgio Di Natale, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 1697–1702. IEEE, 2017. URL https://doi.org/10.23919/DATE.2017.7927267.

[50] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache ATtacks on TLS implementations. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 435–452. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00062.

[51] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util: : Lookup: Exploiting key decoding in cryptographic libraries. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2456–2473. ACM, 2021. URL https://doi.org/10.1145/3460120.3484783.

[52] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. TeeJam: Sub-cache-line leakages strike back. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):457–500, 2024. URL https://doi.org/10.46586/tches.v2024.i1.457-500.

[53] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2003. URL https://doi.org/10.1007/978-3-540-45238-6_6.

[54] Serge Vaudenay. Security flaws induced by CBC padding. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002. URL https://doi.org/10.1007/3-540-46035-7_35.

[55] James Walden. The impact of a major security event on an open source project: The case of OpenSSL. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 409–419. ACM, 2020. URL https://doi.org/10.1145/3379597.3387465.

[56] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers - big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1767–1784. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/weiser.

[57] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015. URL https://doi.org/10.1109/SP.2015.45.

[58] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 719–732. USENIX Association, 2014. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[59] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 346–367. Springer, 2016. URL https://doi.org/10.1007/978-3-662-53140-2_17.