Analyzing Compiler-induced Constant-time Violations with Compiler at Scale

Jing Liu*
MPI-SP
Ruhr University Bochum
jing.liu@mpi-sp.org

Lucía Martinez Gavier*
MPI-SP
lucia.martinez-gavier@mpi-sp.org

Zhiyuan Zhang†
MPI-SP
zhiyuan.zhang@mpi-sp.org

Marcel Böhme
MPI-SP
CISPA Helmholtz Center for Information Security
marcel.boehme@acm.org

Gilles Barthe
MPI-SP
IMDEA Software Institute
gilles.barthe@mpi-sp.org

***Abstract*—The constant-time programming policy is a de facto standard for cryptographic libraries to prevent timing side-channel attacks. However, compiler optimizations may break constant-time source code during compilation. Systematically preventing this requires understanding *where* a violation arises, *how* it is introduced, and *which* compiler configurations are problematic. Existing approaches analyze a single compilation point, either the final binary or one intermediate representation, and thus only partially answer the three questions.**

We propose CTGREP, a compiler-based framework that answers all three questions by applying differential analysis across the *entire* compilation chain. We use CTGREP to analyze seven *complete* cryptographic libraries across three Clang versions, six target platforms, and thousands of compiler configurations. We systematically detect compiler-induced constant-time violations in the wild and collaborate with library developers to address 48 cases. We further identify seven new problematic LLVM passes, bypass and improve the state-of-the-art compiler-flag-based mitigation. Finally, we quantify how the compiler version, target platform, and optimization configuration affect the violations. By covering entire libraries across the full compilation chain, CTGREP effectively detects remaining compiler-induced constant-time violations and complements the community's ongoing hardening efforts.

1. Introduction

Starting from the first timing side-channel attack on cryptosystems [1], many cryptographic implementations, such as AES [2], [3], [4], [5], [6], RSA [7], [8], [9], [10], ECCs [11], [12] and KEMs [13], [14], [15], [16], have been broken as their execution time depends on secret data. A gold standard to protect cryptographic code from timing attacks is to follow the constant-time discipline. Informally, it requires that memory accesses (i.e., data flow) and the conditions of conditional branches (i.e., control flow) do not depend on user-defined secrets [17]. In exchange, the constant-time discipline promises that programs are protected against a class of timing side-channels, such as cache attacks [2], [6], [7], [18] and branch predictor attacks [12], [19], [20]. The promise can be made rigorous as a non-interference property [17], [21]. However, the promise may not hold after compiling the code as compiler optimizations may introduce variable-time code for better performance [22], [23], [24], [25], [26], [27], [28], [29]. Various works have also demonstrated that this compiler-induced variable-time code can be practically exploited [16], [30], [31]. In fact, as early as 1996, Kocher expressed concerns that compiler optimizations may affect constant-time code [1].

Systematically mitigating compiler-induced constant-time violations has remained difficult, as it requires understanding *where* constant-time code can be broken by the compiler, *how* compiler optimizations introduce such violations, and *which* compiler configurations are problematic. Each of these matters to a different party. Cryptographic library developers can harden their libraries by being informed of as many potential compiler-induced constant-time violations as possible. Compiler developers can patch the compiler to compile cryptographic code securely by knowing relevant optimization passes. End-users who compile cryptographic libraries themselves can avoid these vulnerabilities by being aware of "problematic" compiler configurations. This understanding must be comprehensive on all three fronts: an entire codebase rather than a few core cryptographic functions (e.g., AES and RSA); the full optimization pipeline rather than a few optimization passes; and a broad range of compiler configurations rather than a few optimization levels (e.g., -O2 and -O3).

In recent years, various studies have advanced the understanding of the *where*, *how* and *which* of compiler-induced constant-time violations [22], [23], [24], [25], [26], [27],

*Equal contribution.
†Corresponding author.

[29], [32]. Most of them analyze constant-time behavior at the binary level using either dynamic analysis [5] or symbolic execution [24]. Conceptually, their methodologies are similar to detecting source-code constant-time violations [5], [33]. Both approaches require analysts to first select functions of interest, then write harnesses and finally execute the programs. The main difference is that, when analyzing the compiler's impact, more compiler flags and processor platforms are tested [23]. Binary-level analysis directly confirms constant-time violations in the assembly, but because it examines only the output of the compilation chain, it requires extra work to distinguish whether the violations already exist in the source code [25]. Furthermore, the analysts need to manually inspect the compilation pipeline to identify problematic optimization passes [25]. Because of the human effort required in configuring the analysis and analyzing the results, binary-level analysis is typically applied to a handful of high-value functions. For example, the most recent large-scale study of compiler-induced constant-time violations [23] analyzes eleven cryptographic primitives from eight libraries, leaving the rest of each library unanalyzed.

To scale the compiler-induced constant-time violation analysis to an entire library, another line of work scans LLVM IR for two patterns, LLVM's *select* instruction [34] and certain bitwise operations [26]. Unlike binary-level analysis, these approaches first detect lines of constant-time code that could be affected by optimizations, and then examine whether these lines process secrets. These scans cover all compiled code and thus inherently have higher coverage than binary-level analysis. However, these scans examine only one point in the compilation chain. They cannot see how subsequent passes transform the flagged patterns, and cannot confirm violations at assembly level. Consequently, IR scanning answers *where* a violation might arise, but reveals neither *how* violations are introduced nor *which* compiler configurations are problematic.

In this work, we propose a new analysis methodology to systematically answer *where* the compiler can induce constant-time violations, *how* it induces the violations and *which* compiler configurations are problematic at once. The high-level idea is to analyze the entire compilation chain rather than a single point in it by instrumenting the compiler. Concretely, we record the intermediate representations before and after each optimization pass, and use differential analysis to compare them. Comparing the unoptimized input and the fully optimized output of the entire compilation chain reveals *where* the compiler introduces a constant-time violation. This bridges the whole-library coverage of IR scanning and the assembly-level confirmation of binary analysis. Comparing the input and output of each optimization pass automatically reveals *how* the compiler introduces a violation. Since the analysis is conducted by the compiler itself, it inherits support for all compiler flags and target platforms. Therefore, this new methodology can test thousands of compiler flags across many platforms to identify *which* compiler configurations are problematic.

We realize our analysis methodology with LLVM [35] and name the resulting tool CTGREP. Drawing on prior characterization of compiler-induced constant-time violations [22], [23], [25], [32], we define two general oracles for differential analysis to determine whether a constant-time violation has been introduced. The first oracle flags a potential control-flow violation (i.e., secret-dependent control-flow transfer) when a conditional branch appears in the IR after an optimization but not before. The second oracle flags a potential data-flow violation (i.e., secret-dependent memory access) when a memory access whose address is conditionally updated appears after an optimization but not before. We demonstrate the effectiveness of the two oracles by confirming compiler-induced constant-time violations reported in prior work [16], [23], [24], [34]. Like prior IR scanning approaches, CTGREP reports potential violations rather than confirmed ones. Confirming violations requires secret annotations after the analysis [26]. This is different from binary-level analyses, which annotate the program before the analysis. Our large-scale evaluation shows that this post-annotation burden is small — on average only 45 lines per library and 72% of them process secrets.

Using CTGREP, we conduct the largest study of compiler-induced constant-time violations to date. The study covers three versions of LLVM, seven complete cryptographic libraries, six target platforms and thousands of compiler optimization configurations. For detecting *where* constant-time violations are induced, we report remaining vulnerable constant-time code in libraries that have already been analyzed by prior work. We collaborate with library developers to address 48 cases to prevent compiler optimizations from altering constant-time code. Two CVEs have been assigned by WolfSSL and MbedTLS. For understanding *how* a compiler introduces the violations, we reveal seven new "problematic" LLVM passes and comprehensively characterize their joint effect. Crucially, the new findings allow us to bypass and improve the state-of-the-art compiler-flag-based mitigation [25]. For identifying *which* compiler configurations are problematic, we quantify the impact of compiler versions, target platforms and optimization configurations. We also provide a list of compiler flags that can induce constant-time violations and should therefore be used carefully when compiling cryptographic code.

To summarize, we make the following contributions:

1) We propose a new methodology to analyze compiler-induced constant-time violations. It analyzes the *entire* compilation chain with differential analysis to systematically answer *where* these violations occur, *how* they are induced and *which* compiler configurations are problematic.
2) We realize the methodology as CTGREP, an LLVM-based framework, and use it to conduct the largest-scale empirical study of compiler-induced constant-time violations. The study includes three LLVM versions, seven complete cryptographic libraries, six target platforms and thousands of compiler configurations.
3) We systematically reveal the remaining compiler-induced constant-time violations in libraries that have been analyzed before and collaborate with library de-

velopers to address 48 cases.
4) We comprehensively characterize how optimization passes interact to introduce constant-time violations. The new findings allow us to bypass and improve the state-of-the-art compiler-flag-based mitigation.
5) Our large-scale evaluation quantifies how compiler versions, target platforms and optimization configurations affect constant-time code, yielding a list of risky compiler flags for compiling cryptographic code.

Responsible Disclosure. We shared a draft of this paper and reported a list of potential compiler-induced violations to the relevant library developers. The Libgcrypt team has internally discussed our findings and patched potential vulnerabilities. The Libsodium team provided detailed feedback and patched potential vulnerabilities in ed25519_ref10.c. The affected code is under development and is not part of any stable release. The MbedTLS team acknowledged the findings related to `RISC-V` under specific flags, and issued a CVE (CVE-2025-66442). PQClean collects post-quantum cryptography implementations, so we contacted the maintainers of each affected scheme. The ML-KEM maintainers had mitigated the identified issues in mlkem-native before our report; the ML-DSA maintainers had independently identified the affected lines in ml-dsa-native and we collaborated with them to confirm and fix the issues; and the HQC team acknowledged our findings and requested a demo of our framework. The WolfSSL team has examined our report, patched potential vulnerabilities and worked with us to verify the effectiveness of the patches. One CVE (CVE-2025-13912) has been issued by WolfSSL. The BearSSL and Falcon teams thanked us for sharing the findings and shared their insights on fully assuring constant-time execution on actual hardware [28]. OpenSSL does not request an embargo unless a violation leads to a remote timing attack [36]. The LLVM security team did not request an embargo.

Artifact Availability. We have open-sourced our framework at https://doi.org/10.5281/zenodo.22837955.

2. Motivating Example

CTGREP applies differential analysis across the entire optimization pipeline instead of a single point in it. In this section, we use CLANGOVER (CVE-2024-37880) [31], a compiler-induced timing side-channel in the reference implementation of ML-KEM [37], to illustrate how the differential analysis works and its benefits for systematically analyzing compiler-induced constant-time violations.

ML-KEM and CLANGOVER. ML-KEM [37] is a NIST-standardized post-quantum key encapsulation mechanism. During the decapsulation, it re-encrypts the decapsulated secret message (*msg*) and checks if the resulting ciphertext matches the received one. One step of this processing converts the *msg* bit-by-bit into the coefficients of a polynomial (`r->coeffs`). Figure 1a shows the relevant source code. Bit by bit, it assigns a coefficient zero if the message bit is zero and a constant value otherwise. The code is constant-time as it always executes the same code path regardless of the secret message.

However, LLVM optimizations can introduce a conditional branch into the compiled assembly. Figure 1e presents the resulting assembly under `-Os` optimization on `x86-64`. The highlighted conditional branch is taken when the message bit is zero to skip the update. It violates the constant-time policy as the control-flow depends on the secret message bit. As demonstrated by CLANGOVER, the vulnerability allows an attacker to reconstruct the entire secret key even though the source code is constant-time.

Differential Analysis Approach. With debug information, each assembly code can be mapped back to its source code. Therefore, by comparing the source code in Figure 1a and the assembly code in Figure 1e, it is immediately clear that the vulnerability appears in the assembly but not in the corresponding source line, which means that it is introduced by the compiler. This intuition motivates the design of CTGREP's analysis methodology: a constant-time violation is introduced by the compiler when it appears at one point in the compilation chain but not at an earlier point.

Where a violation is introduced. CTGREP detects where a constant-time violation is introduced by comparing the unoptimized LLVM IR (Figure 1b) against the fully optimized LLVM Machine IR (Figure 1d). The unoptimized IR faithfully implements the coefficient update via an `AND` operation, while the Machine IR has replaced it with a conditional branch. LLVM's debug information maps the new branch back to the `AND` operation at the highlighted line in Figure 1a, precisely identifying where the compiler introduces the violation.

How a violation is introduced. A finer-grained differential analysis reveals how a violation is introduced: comparing the input and output of each optimization pass reveals which pass is responsible. In this example, LLVM's *InstCombine* pass converts the `AND` operation into a `select` instruction (Figure 1c), which selects between two values based on a condition. This instruction is constant-time, but prior work [25], [29], [32], [34] has shown that it might be lowered to conditional branches by subsequent passes. Applying the same per-pass differential analysis to Machine IR optimization passes reveals the *X86 cmov Conversion* [25] as the one that introduces the conditional branch.

Which configurations are problematic. Since CTGREP is built into LLVM, it is inherently compatible with all platforms and flags supported by the compiler. In this example, we test six optimization levels: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, and `-Oz`, in combination with the problematic flags identified by our large-scale evaluation for three mainstream platforms: `AArch64`, `RISC-V`, and `x86-64`. On average, each test takes only 0.5 seconds to analyze on an AMD Ryzen Threadripper PRO 7995WX CPU with 96 Cores.

Table 1 presents the analysis results. The vulnerability can only be triggered on `AArch64` and `RISC-V` with specific flags, while it can be triggered on `x86-64` with optimization levels alone. These flags may either be intentionally used by users or maliciously injected by adversaries (e.g., by a supply-chain attack where an attacker manipulates

```
for(i=0;i<KYBER_N/8;i++) { // iterate each byte
  for(j=0;j<8;j++) { // iterate each bit
    mask = -(int16_t)((msg[i] >> j)&1);
    r->coeffs[8*i+j] = mask & 1665;
  }
}
```

(a) Constant-time source code.

```
%sub = sub nsw i32 0, %conv5 ; compute mask
%conv6 = trunc nsw i32 %sub to i16
%conv7 = sext i16 %conv6 to i32
%and8 = and i32 %conv7, 1665
%conv9 = trunc nuw nsw i32 %and8 to i16
```

(b) Constant-time Unoptimized LLVM IR.

```
%3 = icmp eq i32 %2, 0
%conv9 = select i1 %3, i16 0, i16 1665
```

(c) Constant-time Optimized LLVM IR.

```
bb.2:
  %14 = MOV32r0
  BT32rr %12, %13,
  JCC_1 %bb.4, 3,
bb.3:
  %15 = MOV32ri 1665
bb.4:
  %16 = PHI %15, %bb.3, %14, %bb.2
```

(d) Variable-Time Optimized Machine IR.

```
  xorl    %edx, %edx
  btl     %ecx, %r8d  ; test the value of mask
  jae     .LBB0_4     ; branch based on mask
  movl    $1665, %edx ; constant is
  ↪ ((KYBER_Q+1)/2)
.LBB0_4:
  movw    %dx, (%rdi,%rcx,2) ; update
  ↪ r->coeffs[8*i+j]
```

(e) Variable-time x86-64 assembly code.

Figure 1: CLANGOVER: A compiler-induced timing side channel (using optimization level `-Os` for x86-64 architecture).

TABLE 1: Testing compiler configurations for CLANGOVER.

Arch	OPT Level	Compiler Flag
AArch64	O3	-select-opti-loop-cycle-gain-threshold=0, -disable-loop-level-heuristics
RISCV	All, except O0	-fglobal-isel
x86-64	All, except O0	–

the Makefile [38]). Therefore, systematically covering compiler configurations is essential for both determining whether a library is vulnerable under specific conditions and warning users about risky configurations.

3. Preliminary and Related Work

3.1. Timing Attacks and Constant-time Policy

Timing attacks [1] exploit secret-dependent variations in execution time to infer the secret. Most such variations arise from secret-dependent memory accesses [2], [4], [7], or from secret-dependent branches [15], [19], [30], [31]. Another source, less common, is the use of variable-time instructions [39], [40], [41], [42], such as divisions [13], [14], whose execution time depends on their inputs. These instructions are harder to exploit because the secret values must fall within the ranges that can trigger the variable timing behavior.

The Constant-time Policy [17] defends against timing attacks by disallowing secrets to flow into memory accesses and branch conditions. In some scenarios, it further disallows secrets to flow into the operands of variable-time operations [14], [34]. To help developers in writing constant-time code, over 50 tools [43] have been proposed to analyze source code, LLVM IR or the final binary.

Constant-time programming techniques [44] help developers write cryptographic code under this policy. One technique, *bit-slicing* [45], implements the algorithm using bitwise operations as if it were realized in hardware logic. This technique is often used to replace secret-dependent table lookups in symmetric ciphers such as AES. A more general approach is called *single-path coding paradigm* [46]. It executes all paths, or reads all memory locations, and then selects the desired result in a constant-time manner, e.g., via bitmask operations. Our motivating example (see Figure 1a) illustrates this paradigm: each secret message bit is encoded as a bitmask that selects between zero and a constant value, so the same code path is executed regardless of the bit value.

3.2. Compiler-induced Constant-time Violation

In 2018, Simon et al. [22] state that "*something as simple as selecting between two variables x and y, based on a secret selection bit b in constant-time*" can be broken by compiler optimizations. This high-level observation describes precisely the aforementioned single-path coding paradigm. Following work extends this observation in terms of finding more concrete examples [23], [24], [27], [32] and investigating how compilers introduce these violations [22], [25], [26], [29], [32].

Both Daniel et al. [32] and Schneider et al. [23] reveal two patterns of compiler-induced control-flow and data-flow constant-time violations. The first pattern, which is the more common one, is that a compiler may implement the constant-time selection logic with a conditional branch (e.g., as shown by our motivating example). The second pattern occurs more often when the selection logic involves memory accesses. In particular, a compiler may use a conditional move (e.g., `cmove` on `x86`) to update a memory address, which will be accessed later, based on a secret condition. Although the subsequent memory access is performed once, its access address depends on the secret condition and therefore violates the data-flow constant-time policy.

Besides compiler-induced constant-time control-flow and data-flow violations, two recent works [13], [14] have

exploited compiler's optimization on variable-time divisions. Unlike the patterns described above, where the source code is constant-time, the exploited divisions already exist in the source code. The common belief is that compiler optimizations should lower divisions into multiplications and bit-shifts for better performance [47], [48], which also makes the resulting code constant-time. However, under some scenarios, the compiler may choose to preserve them. Consequently, these "compiler-induced" divisions can already be detected by constant-time tools that operate at source or LLVM IR level.

3.3. Detection and Mitigation

3.3.1. Detection. Constant-time analysis tools that operate at assembly level can be directly used to detect compiler-induced constant-time violations. For example, Binsec [24] and Microwalk [5], have already been used by prior works [23], [24], [25], [29], [32]. Most recently, Schneider et al. [23] extend Microwalk, a dynamic constant-time analysis tool, with simulators to support additional CPU platforms and conduct a large-scale study, covering 11 cryptographic algorithms from eight cryptographic libraries on six platforms under six optimization levels. They report new violations, showing that carefully written cryptographic libraries remain vulnerable to compiler optimizations.

IR-level tools can also detect compiler-induced violations by scanning for LLVM IR patterns known to be vulnerable to compiler optimization. Zhang and Barthe [34] search for secret-dependent *select* instructions, which LLVM frequently produces from constant-time selection logic and may then lower to a conditional branch [25], [32]. Bernstein [26] patches LLVM to scan for `&1` and $>>$ *31* bitwise operations in the SUPERCOP benchmark package [49]. These operations are commonly used to build the bitmasks of single-path coding. Our motivating example exhibits both patterns: the bitmask is built with bitwise operations, and the selection logic is lowered first to a *select* instruction and then to a conditional branch.

Binary-level analyses produce fewer false positives, since they confirm violations in the actual binary, whereas IR scanning covers an entire library automatically but may flag patterns that never manifest as variable-time binary.

3.3.2. Mitigation. Two main source-level mitigations are commonly used to mitigate compiler-induced constant-time violations [22], [26], [44], [50], [51]. The first approach is to implement the affected code in (inline) assembly to avoid compiler optimization entirely. The second approach is to annotate variables with the *volatile* keyword to hint that the related code should not be optimized. LLVM additionally has an RFC for built-in constant-time selection primitives that always emit constant-time machine instructions [22], [50]. Applying these defenses requires developers to identify all source-code locations that are potentially affected by compiler optimizations.

More recently, Geimer and Maurice [25] propose a compiler-flag-based mitigation. By manually analyzing the LLVM and GCC optimization pipelines, they identify several problematic optimization passes for `x86-64` and propose disabling them via compiler flags. They also mention that this manual analysis could be automated by comparing the input and output of each pass, but do not investigate it further. Developed concurrently and independently, CTGREP automates the problematic pass identification with per-pass differential analysis. This enables CTGREP to identify seven additional problematic passes, including passes on other CPU platforms. The new results allow us to bypass the proposed compiler-flag-based mitigation [25], and improve it by disabling optimizations from these passes.

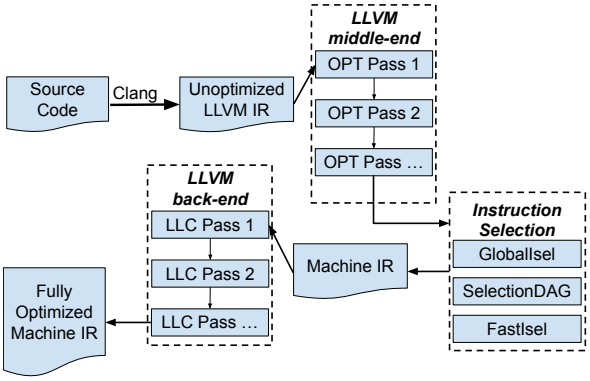

Figure 2: LLVM Optimization Pipeline Overview.

4. CTGREP Framework

We now present CTGREP, our framework for analyzing compiler-induced constant-time violations, in detail. It generates compiler configurations and applies differential analysis across the entire compilation chain to detect *where* and *how* a constant-time violation is introduced.

4.1. Background: LLVM Optimization Pipeline

Since CTGREP instruments the entire LLVM [35] optimization pipeline, we first review this pipeline to provide context for the instrumentation described in the rest of this section. Figure 2 illustrates how a program is optimized, progressing from the front-end through the middle-end to the back-end.

The compiler front-end (i.e., Clang) parses the source code and translates it into LLVM IR. Since no optimizations have been applied at this point, we refer to this artifact as the *unoptimized LLVM IR*. The middle-end then optimizes the LLVM IR with platform-independent optimization passes and produces optimized LLVM IR. Next, an instruction selection pass lowers the optimized LLVM IR into platform-specific Machine IR. Finally, the back-end optimizes the Machine IR with platform-dependent optimization passes and produces the *fully optimized Machine IR*, which is translated nearly one-to-one to the final binary. Figure 1 illustrates these stages on our motivating example. When link-time optimization [52] (`-flto`) is enabled, all object

modules are merged into a single LLVM module and re-optimized through the same pipeline.

A single optimization level invokes many optimization passes. For example, enabling `-O3` optimization on BearSSL [53] triggers 78 middle-end and 116 back-end passes. The set of passes that run is controlled by *compiler configuration*, which consists of four options: ❶ the target CPU platform; ❷ optimization levels, such as `-Ox`; ❸ middle-end flags, specified with `-fxxx` flags; and ❹ back-end flags, specified with `-mllvm --xxx` flags.

4.2. Analysis Scope and Oracle Definition

The analysis scope of CTGREP aligns with previous work [23], [25], [32] on compiler-induced constant-time violations. It detects compiler-induced control-flow and data-flow constant-time violations. We build two general differential oracles to detect known patterns of the two types of violations by distilling prior empirical findings [23], [32].

First, a potential compiler-induced control-flow violation is flagged if a conditional branch appears after an optimization but not before. Most reported compiler-induced timing leakages follow this pattern, including our motivating example. Second, a potential compiler-induced data-flow violation is flagged if a memory access whose address is defined by a conditional move operation appears after an optimization but not before. This pattern was first identified by Daniel et al. [32] and later reported by other work [23], [25]. Compared to control-flow violations, much fewer data-flow violations are reported in the literature. Furthermore, as shown in Table 4 of [32], a compiler-induced data-flow violation can be converted to a control-flow violation under a different compiler configuration.

In this work, we do not consider "compiler-induced" variable-time operations [13], [14], because they already exist in the source code (recall our discussion in Section 3.2). To further explore the potential of compiler-induced variable-time operations, one can design a new differential oracle that checks if a variable-time operation appears in the binary but not in the source code. Building such an oracle requires a complete list of variable-time operations [54], which is target-dependent.

4.3. Constant-time Violation Detection

Figure 3 provides an overview of the CTGREP framework. It fuzzes compiler configurations, compiles the target with an instrumented LLVM that records the relevant IR, and applies differential analysis to detect compiler-induced constant-time violations. We illustrate each step in detail.

4.3.1. Fuzzing Compiler Configurations. CTGREP uses pairwise testing to combine general and fine-grained LLVM flags with standard optimization levels. It supports cross-compilation without dedicated hardware. Evaluation flags and architectures are detailed in Section 5.

```
{"function": "pqcrystals_kyber512_ref_poly_frommsg",
  "file": "poly.c",
  "IR_BR_count": 2, "IR_BR_lines": [1,2],
  "IR_BR_srcs": [ "  for(i=0;i<KYBER_N/8;i++) {",
                  "    for(j=0;j<8;j++) {"],
  "IR_BR_chars": ["f", "f"], "IR_BR_cols": [3, 5],
  "MIR_BR_count": 3, "MIR_BR_lines": [1, 2, 4],
  "MIR_BR_srcs": ["  for(i=0;i<KYBER_N/8;i++) {",
                  "    for(j=0;j<8;j++) {",
    "      r->coeffs[8*i+j] = mask & ((KYBER_Q+1)/2);"],
  "MIR_BR_chars": ["f", "f", "&"],
  "MIR_BR_cols": [3, 5, 31] }
```

Listing 1: Recording relevant information before and after *all* compiler optimizations for our CLANGOVER example.

4.3.2. Recording (Un)optimized Code IR. In the *middle-end*, CTGREP records unoptimized LLVM IR (branches and memory operations) using a custom pass registered at the `PipelineStartEP` extension point. This pass is dynamically invoked without modifying LLVM source code.

In the platform-specific *back-end*, we record fully optimized Machine IR (branches and memory operations with addresses updated by conditional moves) immediately before machine code emission. Due to LLVM back-end constraints, this requires modifying LLVM directly via a 988-LoC patch compatible with LLVM 14 and later. Because CTGREP operates as a compiler instrumentation, it scales to arbitrary-sized programs, serializing all collected data to JSON.

4.3.3. Scalable Analysis of Potential Control-flow Violations. CTGREP flags conditional branches that appear in the fully optimized Machine IR but not in the unoptimized LLVM IR. The differential analysis is implemented as Python scripts that take the two IR implementations of one function, map IRs that have branches to their source code, and check if they match on locations that have conditional branches. Listing 1 shows a simplified output for our motivating example. It records the branches of the unoptimized LLVM IR (`IR_BR_*` entries) and the fully optimized Machine IR (`MIR_BR_*` entries). Entries `IR_BR_lines` and `MIR_BR_lines` record the corresponding source code of IRs that are conditional branches. Comparing them reveals that the compiler introduces an additional branch in line 4 of Figure 1a, which is exactly the CLANGOVER vulnerability.

False Positives Filtering. We consider a flagged case a false positive if the branch already exists in the source code. It typically arises from function inlining, where a branch in the callee's source code is inlined to the caller. CTGREP filters these false positives in two stages. First, it checks whether the unoptimized LLVM IR of the callee contains the corresponding branch. If so, the branch is not compiler-induced and is excluded. When the callee's IR is unavailable (e.g., it is inlined from another source file), CTGREP instead inspects the source line and excludes branches that arise from source-level branching constructs, including branch statements (e.g., `switch`, `if`), loop structure (e.g., `for`,

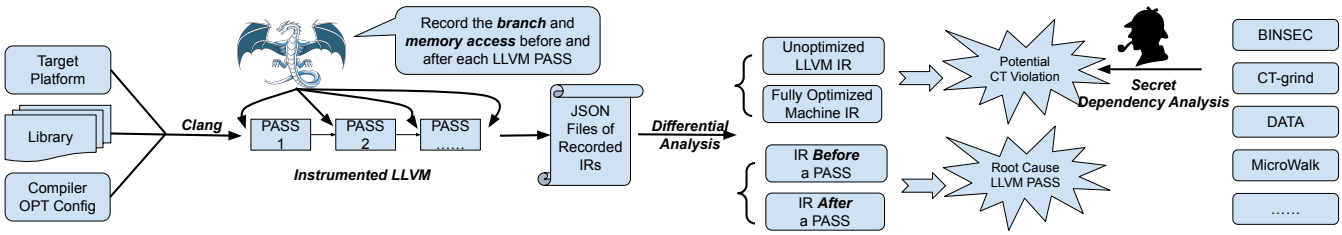

Figure 3: CTGREP Workflow.

`while`) and language features known to introduce branches (e.g., `catch`, `throw`).

4.3.4. Scalable Analysis of Potential Data-flow Violations. Unlike conditional branches, conditional moves are target-dependent and exist only in the Machine IR. Therefore, the data-flow oracle cannot be directly applied to compare the unoptimized LLVM IR and the fully optimized Machine IR. CTGREP instead first scans the fully optimized Machine IR for a memory access whose address is defined by a conditional move, and then filters out cases where the corresponding source code already contains a conditional branch. The filtered cases are already not constant-time in the source code and therefore the conditional memory access is not compiler-induced.

The scanning is implemented in the LLVM back-end: it locates each conditional move instruction and scans the subsequent instructions in the same basic block for a memory access that uses the defined register as its address. The accuracy highly depends on LLVM's support for correctly identifying the conditional move, the dependent memory access, and modeling the data flow between them. The analysis is precise on well-supported platforms: `x86-64` and `AArch64`. However, for the 32-bit ARM and MIPS backends we do not find a reliable way to always correctly determine whether a conditionally updated register is used as a memory address. For these platforms, we relax the analysis by accepting any memory access with at least one operand register defined by a conditional move.

False Positives Filtering. To further reduce false positives, CTGREP applies an additional filter: it keeps a data-flow violation only at source locations where a compiler-induced control-flow violation can be detected under another compiler configuration. As noted in Section 4.2, a compiler-induced data-flow violation can be converted to a control-flow violation under a different compiler configuration. The intuition is that a conditionally selected memory address can be lowered into either a conditional branch or a conditional move, depending on the activated optimizations.

4.3.5. Confirming Secret Dependency. The two oracles report *potential* compiler-induced constant-time violations, because they do not reveal whether the induced variable-time code processes secrets. Confirming secret dependency requires annotating instruction operands or function inputs, which is conventionally the first step of constant-time analysis. Like other IR-scanning tools [26], [34], CTGREP defers this step until after detecting all potential vulnerabilities in an entire library. The confirmation step can be done on source, LLVM IR, or binary level. One can annotate function inputs and run conventional constant-time analysis tools such as BINSEC [24], CT-grind [55], DATA [33] or Microwalk [5]. Alternatively, one can manually inspect the source code to confirm the secret dependency. For example, it is easy to confirm that the *mask* variable in the motivating example is a secret. With either method, annotation is reduced from the entire library to the small subset of reported violations.

4.4. Automatic Root Cause Analysis

After identifying *where* a constant-time violation is induced, CTGREP locates *how* it is induced by applying the two differential oracles to the input and output of each optimization pass. Before and after each middle-end and back-end pass, including link time optimization, CTGREP records the IR relevant to both oracles. In the middle-end, it instruments the `BeforeNonSkippedPass`, `AfterPass` and `AfterPassInvalidated` extension points, and adds hooks for passes still managed by the legacy pass manager [56]. In the back-end, it modifies `MachineFunctionPass`. A pass is flagged as the root cause if it introduces a conditional branch or a conditionally defined memory address that its input IR did not contain.

4.5. Ground Truth Validation

As a sanity check, we validate CTGREP against all 19 LLVM-induced constant-time violations reported by previous work [16], [23], [24], [34] (Appendix Section A). Of these, 18 are control-flow violations and one is a data-flow violation. Running CTGREP under the problematic compiler configurations reported by prior work, it recovers all 19 violations without producing false positives.

5. Experiment Design and Setup

With CTGREP, we conduct a large-scale empirical study to identify where a compiler can introduce constant-time violations in popular cryptographic libraries, reason how they are introduced and investigate the impact of compiler configurations. This section describes our experiment infrastructure and methodology.

5.1. Experiment Infrastructure

Tested Compiler Versions. We focus our qualitative evaluation on Clang 18 to verify the violations it induces and to collaborate with library developers to patch them. We expect Clang 18 to be widely used, as it is the default Clang version of Ubuntu 24.04 LTS, a major long-term-support Linux distribution release. To study how the compiler's evolution affects constant-time code, we additionally test Clang 14 and Clang 20 and evaluate the results quantitatively. We leave evaluating more compilers to future work.

Tested Compiler Configurations. We explore how different optimization flags affect constant-time properties. After excluding flags that are used for development debugging (e.g., `-fprofile-xx`) and induce non-exploitable branches (e.g., `-fwrapv`, which only tells if an integer overflow happens), we initially obtain 2371 optimization flags by parsing the help information of Clang, LLVM and GCC. Out of those, 646 flags configure the middle-end while 1725 flags configure the back-end. For flags taking values, we always use 0, 50, 100 for integer arguments, and 0.0, 0.5, 1.0 for floating-point arguments. For flags taking predefined enum values or boolean values, we use each possible value. As a result, we have *4318 concrete flags* across the three Clang versions, with 968 from GCC/Clang and 3350 from LLVM back-end. During the large-scale evaluation, each flag option is tested with one of the six optimization levels (`-O0`, `-O1`,`-O2`, `-O3`, `-Os` and `-Oz`) for pairwise testing.

Tested Target Platforms. We test three mainstream platforms: `AArch64`, `RISC-V (64-bit)` and `x86-64`; and three less common platforms: `ARM (32-bit)`, `MIPS32EL` and `MIPS64`. We run all experiments on `x86-64` machines and cross-compile for other platforms.

Tested Cryptographic Libraries. We test seven popular cryptographic libraries: BearSSL (3c04036), Libgcrypt (cbefff5), Libsodium (fd8c876), MbedTLS (f68f45c), OpenSSL (2e1b046), WolfSSL (f119086) and PQClean (2cc6471). Among them, BearSSL, Libsodium, MbedTLS and WolfSSL particularly focus on constant-time implementations; Libgcrypt and OpenSSL are widely deployed (e.g., default in many Linux distributions). PQClean is not a general-purpose cryptographic library but a collection of the clean implementations of various post-quantum cryptographic algorithms [57]. We focus on the algorithms that have been standardized by NIST [58].

Experiment Environment. The experiments compile libraries, instrumented by CTGREP, with different compiler configurations and target platforms. We parallelize the experiments with multi-threads on three machines. One machine has an AMD Ryzen Threadripper PRO 7995WX CPU with 96 cores and 256 GB of RAM, and the other two machines have an AMD EPYC 7713P CPU with 64 cores and 256 GB of RAM.

5.2. Experiment Methodology

Ideally, we would analyze each complete library under every combination of a compiler flag and a compiler optimization level for all six platforms. However, exhaustively testing all of them is computationally prohibitive. Therefore, we adopt a two-phase analysis strategy. We first run the full test (all paired compiler flags and platforms) with three libraries to identify a list of problematic configurations. Then, we apply them to the remaining libraries.

5.2.1. Phase 1: Problematic Configuration Identification. In the first phase, we use CTGREP to fully analyze BearSSL, MbedTLS, and PQClean to identify problematic compiler configurations. It results in around *1.2M* unique analyses ($\sim$4k flags $\times$ 6 archs $\times$ 3 compilers $\times$ 6 optimization levels $\times$ 3 libraries). We select these libraries because they provide a wide range of constant-time coding patterns. Furthermore, each analysis takes around three seconds for BearSSL and around 15 seconds for MbedTLS and PQClean, which is also time efficient. Overall, we identify 218 problematic concrete flags, with 148, 139, and 133 appearing for Clang 14, 18, and 20, respectively.

5.2.2. Phase 2: Violation Detection. With the flag set reduced to about 150 per compiler version, we apply CTGREP to the four remaining libraries under these flags, across all optimization levels and platforms. We parallelize each analysis with `-j32` and set a 300-second timeout.

5.2.3. Threats to Validity. As the first systematic study of how compiler configurations affect constant-time code, we have, to the best of our knowledge, covered the largest set of code, compiler flags and platforms in the literature [22], [23], [24], [25], [27], [32]. However, two limitations affect the completeness of our experiments.

Flag Combinations. First, our evaluation focuses on testing each flag with each optimization level, as testing all combinations of flags is not practical. We initialize a multi-flag analysis by testing all pairwise combinations of problematic flags for BearSSL on `x86-64` and `AArch64`. It results in 7,260 pairs of flags and over 100k unique analyses per platform. Compared to the single-flag analysis, this multi-flag analysis only reveals five unique potential compiler-induced constant-time violations and it does not reveal new problematic passes (see Appendix D). It indicates that combining problematic flags does not necessarily lead to more violations, and the impact of multi-flag combinations might be limited. A more systematic exploration of multi-flag combinations requires not only combining individually problematic flags, but also individually benign ones and mixed combinations of both. Flag ordering also matters as a later flag may override settings from an earlier one. We leave a more systematic multi-flag evaluation to future work.

Problematic Flags. Second, our Phase 1 analysis may miss problematic flags that do not induce violations in the three evaluated libraries but may do so in the remaining libraries. Overall, CTGREP reveals 121 problematic flags using BearSSL, and additional 59 and 38 after adding PQClean and MbedTLS, respectively. The decreasing number of newly identified problematic flags indicates diminishing marginal returns from testing more libraries in Phase 1. As

we have noted, our current Phase 1 analysis already involves 1.2M analyses, and extending the analysis to the remaining libraries would be computationally prohibitive.

6. Large-scale Cryptographic Library Analysis

We apply CTGREP to seven cryptographic libraries. With Clang 18, it detects 311 potential constant-time violations, of which manual analysis confirms 225 process secrets. We report the findings to the respective library developers and collaborate with five of them to address 48 out of 59 locations, with two CVEs assigned. These results show that compiler-induced constant-time violations persist even in carefully written and previously analyzed libraries. CTGREP automatically detects them and helps developers patch and verify the fixes. In this section, we focus the analysis on Clang 18 and report results for other compiler versions in Appendix C.

6.1. Quantitative and Qualitative Analysis

Quantitative Analysis. We report the number of compiler-induced potential constant-time violations in Figure 4, distinguishing those that can be induced by optimization levels from those requiring specific compiler flags. Each case, identified by a unique combination of source line and the column of affected operator, is counted once, regardless of function inlining or loop executions.

As Figure 4 shows, most libraries have fewer than 80 potential compiler-induced constant-time violations. OpenSSL exhibits more such cases as it has a much larger and more complex code base. Libsodium and MbedTLS have the least number of potential violations, 8 and 17 respectively, reflecting their long-standing emphasis on constant-time programming. Libsodium descends from NaCl [59], which is well known for its secure implementation. MbedTLS has likewise prioritized constant-time design, and in 2023 specifically hardened the library against compiler-induced violations (ChangeLog). In contrast, BearSSL and WolfSSL also prioritize constant-time programming, yet exhibit slightly more potential compiler-induced violations. This is likely because they more frequently use bitwise operations for constant-time selections.

On average, each library has 45 potential compiler-induced constant-time violations. This number is very small compared to the size of their code bases. It indicates that modern cryptographic libraries generally have good resilience against compiler optimizations. However, these violations are scattered across the entire library. This is why they remain in the wild, despite prior detection efforts, which typically focus on a few core cryptographic functions, and why full-library analysis is essential.

Qualitative Analysis. To determine how many of these potential violations are actual constant-time violations, we manually analyze the source code of each case to determine if it processes secrets. Conventionally, this annotation is done as the first step of constant-time analysis, since most of them require secrets to be specified upfront. CTGREP reverts this order by first scanning the entire library and then determining secret-dependency only for the reported cases. With an average of 45 potential cases per library, this manual analysis is feasible.

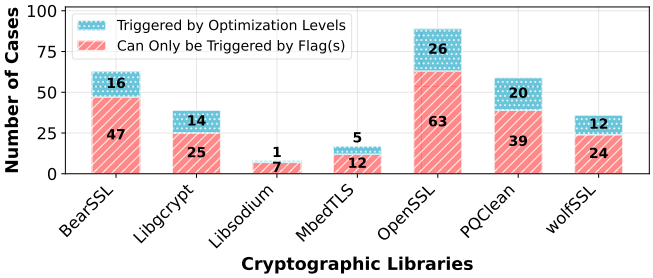

Figure 4: Number of detected Clang-18-induced potential constant-time violations per cryptographic library.

We present the analysis results in Table 2, where we further breakdown PQClean by the post-quantum cryptographic scheme. The secret-dependent rate highly varies between the libraries and schemes. Most of them are over 70%, indicating that the majority of reported potential violations do process secrets. Libsodium and MbedTLS have significantly lower secret-dependent cases, which are consistent with their strong emphasis on constant-time programming.

TABLE 2: Secret-dependency analysis of compiler-induced potential constant-time violations.

Library	Scheme	Potential Violations	Secret-dependent
BearSSL		63	73.0% (46)
Libgcrypt		39	74.4% (29)
Libsodium		8	37.5% (3)
MbedTLS		17	23.5% (4)
OpenSSL		89	73.0% (65)
WolfSSL		36	66.7% (24)
PQClean	Falcon	29	86.2% (25)
	HQC	25	100.0% (25)
	ML-DSA	2	100.0% (2)
	ML-KEM	2	100.0% (2)
	SPHINCS+	1	0.0% (0)

TABLE 3: A summary of addressed, potentially exploitable compiler-induced constant-time violations.

Library	Secret-dependent Cases	Addressed Cases
Libgcrypt	28	85.7% (24)
Libsodium	3	100.0% (3)
MbedTLS	3	100.0% (3)
ML-DSA	2	100.0% (2)
WolfSSL	23	69.6% (16)

6.2. Summary of Constant-time Violations

We note that not all secret-dependent variable-time code can be exploited. For example, some implementations may compute key length from the secret key. Although the key

length is secret-dependent and variable-time code that processes it is considered as not constant-time, leaking the key length often does not leak exploitable information [60]. We likewise consider violations on non-mainstream platforms, such as `MIPS`, not practically exploitable, as no attacks have been reported on them. Therefore, we report potentially exploitable violations on the three mainstream platforms to the relevant library developers and discuss with them whether patches are required.

Four libraries, Libgcrypt, Libsodium, ML-DSA and WolfSSL, have patched the corresponding code and MbedTLS has acknowledged the issues. We summarize them in Table 3. Overall, 81% of the secret-dependent compiler-induced constant-time violations are addressed. While this does not prove the violations are exploitable in practice, it shows that developers consider them a genuine security concern, reflecting their ongoing efforts to harden cryptographic code against compiler-induced violations. We provide a list of addressed cases in Appendix B (Tables 7, 8 and 10). We now discuss the findings that have been patched or acknowledged by the library developers.

Libgcrypt team has confirmed that the bitwise operations in some constant-time functions could be converted to variable-time binary due to Clang optimizations. For example, the bitwise operations in `do_setkey` and `_gcry_aes_xts_crypt` in `rijndael.c` file are affected. Similarly, arithmetic operations in `sntrup761.c`, such as `int32_MINMAX` and `uint32_divmod_uint14` are also affected. At the time of reporting our findings, Libgcrypt is also internally hardening constant-time functions against compiler optimizations. Our findings confirm their observation that Clang often replaces mask + AND operations with conditional branches.

Libsodium team has confirmed compiler-induced constant-time violations in file `ed25519_ref10.c`. They can be converted to variable-time code on `x86-64` by controlling the lowering of conditional move operations within LLVM. They appear in development code that should not be used in production and have been patched.

MbedTLS team has acknowledged that the reported bitwise operations may process secret information. According to our findings, they can be converted to variable-time binary on the `RISC-V` platform with a specific compiler flag: `-mllvm --disable-select-optimize=false`. One CVE has been assigned.

ML-DSA maintainer team considered that two functions: `caddq` and `decompose` might be affected by compiler optimizations before our disclosure. We collaborated with them to confirm the violations and patch the vulnerabilities.

WolfSSL team has confirmed that some constant-time functions could result in variable-time binary due to compiler optimizations. Some of them are related to big number operations, such as `sp_read_radix`, `sp_div_2_mod_ct` and `sp_addmod_ct`. Furthermore, operations related to asymmetric cryptography, such as RSA unpad operations and DH key pair generation could also be affected by specific compiler flags. All affected functions have been patched. One CVE has been assigned.

6.3. Evaluating Mitigation Effectiveness

Besides finding and fixing compiler-induced potential constant-time violations, it is also important to evaluate the effectiveness of the fix patches. A recent work shows that some side-channel vulnerability patches are not effective and could even introduce new vulnerabilities [61]. We find that the most common fix strategy used by developers, after our report, is to bind *volatile* keyword to secret variables. Such mitigation is more portable but might be ignored by the compiler [51]. CTGREP helps confirm whether the compiler may still optimize *volatile* variables by fuzzing a large number of compiler configurations.

We collaborate with WolfSSL developers to confirm the effectiveness of their patches. We find that most patches are effective except marking variables that are used in a long and complex expression as *volatile*. More specifically, marking a variable as *volatile* at its declaration may not prevent the compiler from optimizing a long expression that uses this variable which is far from its declaration. One solution is to decouple the long expression into multiple short expressions and mark the intermediate results as *volatile*. We have confirmed that this solution is effective using CTGREP.

7. Fine-grained LLVM Optimization Analysis

CTGREP automatically identifies which optimization passes induce constant-time violations. By analyzing all potential violations reported by the differential oracles, we identify 11 problematic optimization passes. We comprehensively characterize how they interact to induce constant-time violations and show how to bypass and improve the state-of-the-art compiler-flag-based mitigation [25].

7.1. Summary of Problematic LLVM Passes

Table 4 summarizes the eleven problematic LLVM passes identified by our framework, four of which have also been discovered and evaluated by Geimer and Maurice [25], while the remaining seven are newly identified. A violation can be either induced in the middle-end stage, where LLVM IR is manipulated, or in the back-end stage, where LLVM IR is lowered to Machine IR. We classify the passes based on the type of instructions they introduce: *select* instructions, conditional branch instructions, or conditional memory accesses. We observe that **InstCombinePass** could introduce *select* IR instructions that may be lowered to conditional branches by **X86CmovConversion** pass. Also, we confirm that **LoopVectorizePass** and **SimpleLoopUnswitchPass** could introduce conditional branches. As new observations, we find that **OptimizeSelects** and **CodeGenPrepare** may replace *select* instructions with conditional branches. Finally, we identify several platform-specific LLVM passes that could convert *select* instructions into conditional branches or conditional memory accesses.

TABLE 4: LLVM passes that may induce constant-time violations. Rows shaded yellow are middle-end passes and blue are back-end passes. We mark a pass with ✓ if it has been discussed by [25]; otherwise we mark it with ✗.

Instr	LLVM Pass	[25]
select	InstCombinePass	✓
branch	LoopVectorizePass	✓
	SimpleLoopUnswitchPass	✓
	Optimize selects	✗
	CodeGen Prepare	✗
branch	Finalize ISel and expand pseudo-instructions	✗
	X86 cmov Conversion	✓
	RISCV DAG→DAG Pattern Instruction Selection	✗
cmove/ branch	MIPS DAG→DAG Pattern Instruction Selection	✗
	ARM Instruction Selection	✗
	X86 DAG→DAG Instruction Selection	✗

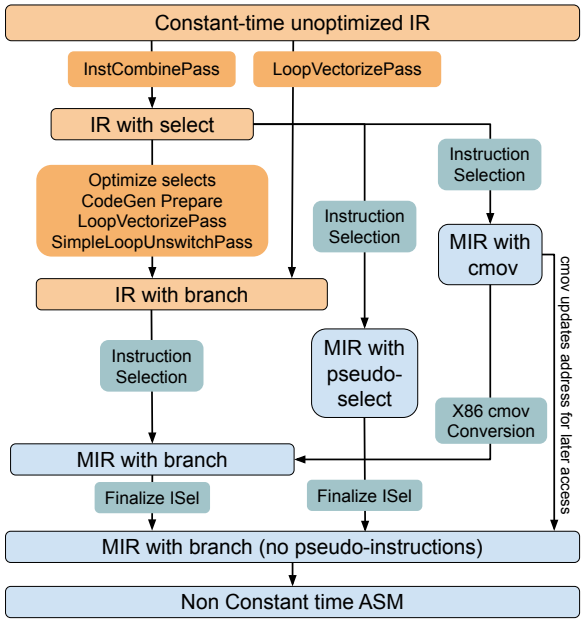

Figure 5: A comprehensive characterization of how LLVM optimization passes transform constant-time LLVM IR into variable-time Machine IR. Middle-end passes are shaded with yellow and back-end passes are shaded with blue.

7.2. LLVM Passes Interaction Characterization

Figure 5 visualizes the interaction among the identified problematic compiler passes. In some cases, constant-time code may be turned into variable-time code not by a single pass but by a sequence of transformations. For simplicity, we do not specify a target platform for the instruction selection in this figure. However, we note that `cmov` instructions are emitted only for targets that support them, whereas the `pseudo-select` instructions can be emitted on other targets.

We first report our findings that align with the results presented by Geimer and Maurice [25] in this paragraph. Same as [25], we also observe that lowering the *select* instruction is a main source of compiler-induced violations. The instruction selects between two values based on a condition and is issued by the **InstCombine** pass, which merges instructions to form simpler ones [62]. When testing compiler flags, we find that using certain LLVM flags, such as `--attributor-enable`, increases the likelihood of issuing *select* instructions. Besides, we also confirm that **LoopVectorize** and **SimpleLoopUnswitch** may introduce conditional branch IR instructions. The conditional branches at IR level are likely to be kept in the final binary, while the *select* instructions *might* be lowered to variable-time operations or constant-time operations, such as `cmov` instructions, depending on the subsequent passes and the target platform. For example, on the `x86-64` platform, the **X86CmovConversion** can convert *select* IRs into conditional branches [24], [25]. We next focus on the problematic LLVM passes newly identified by our empirical study.

SelectOptimize pass may replace a *select* instruction with a conditional branch IR instruction when it considers that the code would be more efficient when using a branch [63]. For instance, if a branch is highly predictable or if a cold path contains expensive operations, this pass converts the *select* instruction into a conditional branch so that the costly path can be skipped when the condition is not met. LLVM flags, such as `--mispredict-default-rate=0`, may make branches look cheap, which encourages this optimization (online demo: Godbolt #1). On `AArch64`, this optimization is enabled by default, and on other architectures it can be turned on with LLVM flag `--disable-select-optimize=false`.

CodeGenPrepare pass reshapes LLVM IR so that code generator passes, such as **SelectionDAG**, could emit faster machine code. This pass may convert *select* instructions into conditional branches at IR level. It happens when several *select* IRs use the same condition or, similar to **SelectOptimize** pass, when one of the two paths costs more to compute (online demo: Godbolt #2).

Finalize ISel and expand pseudo-instructions pass handles the lowering of pseudo-instructions to real instructions. For example, on platforms that do not support conditional moves, such as `RISC-V`, the pass converts pseudo conditional moves into conditional branches. Furthermore, we find the *Fast Instruction Selector* (enabled with `--fast-isel`) may fail to convert a *select* instruction into a conditional move when it has an unsupported return type like `i8`. If this happens, the compiler will fall back to a pseudo conditional move, which is later converted to a conditional branch (online demo: Godbolt #3).

DAG→DAG Pattern Instruction Selection passes are implemented differently for different target platforms to select machine instructions based on the target's features. It may lower the *select* instruction into a conditional move that updates the address of a memory access operation. The selectively updated address may later be accessed and hence reveal the condition of updating the address. It usually happens when the selection logic is

applied to memory access. For example, consider this code `r[i] ^= mask & (r[i] ^ rho[i])`. It uses constant-time logic to implement a conditional write. If *mask* is nonzero, the array entry `r[i]` is updated with `rho[i]`. Otherwise, `r[i]` remains unchanged. In the optimized IR, the **InstCombinePass** pass replaces this operation with a select instruction. At this point, the code is still constant-time, since both `r[i]` and `rho[i]` are always accessed regardless of the condition. However, the **X86 DAG→DAG instruction selection** pass may update the address to access with a conditional move. Therefore, the memory access only needs to be performed once (online demo: Godbolt #4).

7.3. Compiler-flag-based Mitigation Evaluation

After identifying problematic LLVM passes, one could set compiler flags to disable them to prevent compiler-induced constant-time violations. Geimer and Maurice [25] recently proposed the first compiler-flag-based mitigation for `x86-64` platform by disabling four problematic LLVM passes [1]. As noted by the authors, the mitigation may not be complete as it was tested only on a small set of functions for the `x86-64` platform. Using our framework, we are able to more comprehensively evaluate the effectiveness of their proposed mitigation for larger codebases across multiple platforms and compiler configurations.

Evaluation Setup. We configure CTGREP to always apply the mitigation flags and test whether adding other compiler options or other target platforms can induce potential constant-time violations. Specifically, we test whether Clang 18 can still induce violations in BearSSL library on six platforms with the presence of mitigation. We ensure the mitigation is enabled by appending the mitigation flags after the compiler configurations generated by our framework.

`x86-64` Evaluation Results. On `x86-64`, the mitigation is effective for nearly all configurations tested by CTGREP on BearSSL. The exceptions are *two* compiler flags that bypass it and still induce violations. Recall that we have identified that **SelectOptimize** pass could replace *select* instructions with conditional branches when using flag `--disable-select-optimize=false`. We find one constant-time violation is caused by using this flag with optimization level `-O1` and `-O2` (online example: Godbolt #6). The other two compiler-induced constant-time violations are caused by using the `--fast-isel` flag with various optimization levels (online example: Godbolt #7). This flag affects the **Finalize ISel and expand pseudo-instructions** pass identified by our empirical study.

Extending the Mitigation for x86-64. According to our findings, we extend the compiler-flag-based mitigation by further turning off two optimization passes with the following flags: `-mllvm --disable-select-optimize=true` and `-mllvm --fast-isel=false`. We confirm that CTGREP *does not* report any potential constant-time violations under all tested compiler configurations for the `x86-64` platform.

1. -mllvm --x86-cmovconverter=false -mllvm --disable-cgpselect2branch=true -mllvm --unswitch-threshold=1 -fno-vectorize

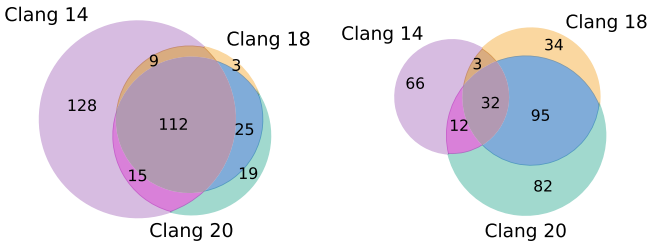

Figure 6: Cases of potential compiler-induced constant-time violations across compiler versions. Left: optimization-level induced; Right: optimization-flag induced.

Other Platforms Evaluation Results. We find the mitigation [25] is less effective on other platforms, which is expected. `RISC-V`, which does not support conditional move, is still more prone to compiler-induced constant-time violations. Therefore, we recommend either using assembly or *volatile* keywords to mitigate the issues for it. For the remaining platforms, we find that extending the mitigation with the two aforementioned flags can eliminate most of the rest potential constant-time violations. One exception is that several optimizations related to loop unrolling can still induce constant-time violations on the `AArch64` platform. These flags affect the **InstCombinePass** and **SelectOptimize** passes. We have confirmed that setting three additional flags [2] can eliminate the corresponding violations.

8. Impact of Compiler Configurations

After identifying *where* a compiler induces constant-time violations and *how* it induces them, we now investigate *which* compiler configurations are more problematic. In particular, we quantify how compiler version, platform and optimization configuration affect the number of potential compiler-induced constant-time violations.

8.1. Impact of Compiler Version

Figure 6 compares potential compiler-induced constant-time violations across three studied Clang versions, separately for those induced by optimization levels (left) and optimization flags (right). In general, the results confirm the intuition that newer compilers tend to induce more constant-time violations [23], [25]. Furthermore, most of the increase comes from optimization flags rather than optimization levels. One exception is Clang 14, which has more unique optimization-level-induced violations (128) than the newer versions. We find that 118 out of these are unique to `RISC-V` and that newer Clang versions induce fewer such violations. This is likely because `RISC-V` support was less mature in Clang 14 and it has been improved as the platform's adoption has grown.

2. -mllvm --disable-loop-level-heuristics=false -mllvm --unroll-runtime=false -mllvm --unroll-allow-partial=false

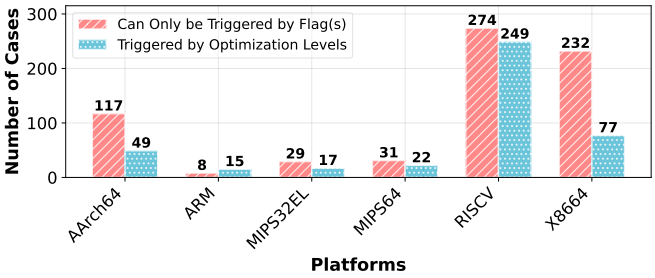

Figure 7: Number of potential constant-time violations across different target platforms, separated by whether they are triggered exclusively by flags or by optimization levels.

8.2. Impact of Target Platform

Figure 7 quantifies how constant-time code is affected by compiler optimizations across different target platforms. Mainstream platforms, `AArch64`, `RISC-V` and `x86-64`, are more affected by compiler optimizations, though for different reasons. On `AArch64` and `x86-64`, the more mature backends have more optimization strategies that may induce violations. On `RISC-V`, the lack of conditional-move instructions causes *select* instructions to be more likely lowered to conditional branches, especially with Clang 14. Overall, different platforms are affected by compiler optimizations to different degrees. It sheds light on building platform-specific mitigations (e.g., inline assemblies): platforms that are less affected require fewer mitigations and thus likely incur lower performance overhead.

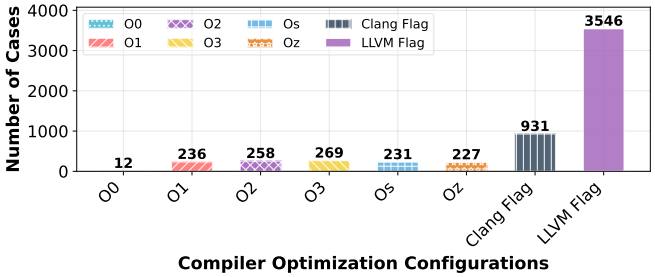

Figure 8: Number of potential constant-time violations across different optimization configurations.

8.3. Impact of Optimization Configuration

Figure 8 shows how different optimization configurations affect the number of compiler-induced potential constant-time violations. One case is counted multiple times if it can be triggered by different configurations. Surprisingly, we find that more aggressive optimization does not necessarily lead to significantly more variable-time code. Particularly, from `-O1` to `-O3`, the number of induced potential constant-time violations remains relatively stable. `-Oz`, which optimizes code for size more aggressively than `-Os`, introduces even fewer potential constant-time violations. Among the 12 potential constant-time violations induced by `-O0`, nine of them are on `RISC-V`, while the rest are on non-mainstream platforms. It is expected that `-O0` induces significantly fewer violations as it enables almost no optimizations. This finding also aligns with previous work [23]. Finally, we find that back-end flags (i.e., LLVM flags) induce significantly more variable-time code than general flags (i.e., Clang flags). This is because back-end flags have more fine-grained control on optimizations and can set optimization strategies that are not reachable through Clang flags (e.g., -fxxx). In addition, we identify a set of flags, such as `--x86-cmov-converter-force-all` and `--fast-isel`, that are more likely to introduce constant-time violations. We present a detailed list in Appendix Tables 5, 6 and 9. These flags should be used more carefully when compiling security-critical code.

9. Discussion

Through our collaboration with library developers, we observe a broader trend in writing constant-time code: having adopted constant-time programming at the source level, developers are now turning their focus to ensuring the final assembly is also constant-time. First, we find that many of the contacted library developers have already been actively hardening their libraries before our report. Second, they not only investigated and patched the cases reported by us, but also proactively hardened other places that are not yet affected by compiler optimizations.

Since these libraries are complex, it is challenging to manually select analysis targets without omissions. This explains why we can still identify compiler-induced constant-time violations in them. CTGREP helps developers to systematically harden their libraries by first reporting as many potential violations as possible and then letting developers determine if they process secrets and require patches. The required manual effort is minimal: on average only 45 cases per-library require manual analysis.

Beyond detecting remaining violations in the wild and revealing additional problematic passes and flags, CTGREP has two main limitations. First, as a differential analysis framework, CTGREP does not guarantee to detect all compiler-induced constant-time violations or report no false positives. The two oracles are designed based on all known compiler-induced constant-time violation patterns, and we are not aware of a real-world example that CTGREP fails to detect. However, future compiler optimizations may introduce new violation patterns that CTGREP cannot detect with the current oracles. For example, we find that LLVM may recognize loop memory idioms (e.g., constant-time comparison of two memory blocks) and replace them with intrinsics like `memcmp` in the future (LLVM TODO). This optimization will not introduce new branches or memory accesses, but an external function call, which is not constant-time, to a C library instead. More generally, future optimizations may replace a constant-time code block with an external function call, whose implementation may not be constant-time. To capture such potential constant-time violations, we need to adapt our oracles to detect if a

binary contains a non-constant-time external function call that does not exist in the source code. On the other hand, as shown in Table 2, false positives exist as a reported case may not process secrets. Another source of potential false positives may come from the correctness of the LLVM debug information that CTGREP uses to map optimized code back to its source. The second limitation is that CTGREP, as a compiler-dependent framework, currently only detects LLVM-induced violations and does not cover those induced by GCC or other compilers. While our methodology of instrumenting the entire compilation chain is general for modular compilers, instrumenting a new compiler requires non-trivial engineering effort. We leave the instrumentation of GCC, and a comparison with the findings reported in this paper, to future work.

10. Conclusion

With CTGREP, we systematically answer the questions of *where*, *how* and *which* in *entire* cryptographic libraries across various compiler versions, optimization configurations and target platforms. It helps cryptographic library developers to harden their libraries and improves a recently proposed compiler-flag-based mitigation based on the new problematic LLVM optimization passes automatically identified by our framework. As another result of our large-scale study, we also quantitatively reveal compiler optimization's impact on constant-time code. Overall, our study provides timely insights for understanding and mitigating compiler-induced constant-time violations at scale.

References

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *CRYPTO*, 1996, pp. 104–113.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA*, 2006, pp. 1–20.

[3] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[4] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *DIMVA*, vol. 9721, 2016, pp. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14

[5] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *ACSAC*, 2018, pp. 161–173. [Online]. Available: https://doi.org/10.1145/3274694.3274741

[6] S. H. W. Cheng, C. Chuengsatiansup, D. Genkin, D. McNeil, T. Murray, Y. Yarom, and Z. Zhang, "Evict+spec+time: Exploiting out-of-order execution to improve cache-timing attacks," *CHES*, vol. 2024, no. 3, pp. 224–248, 2024. [Online]. Available: https://doi.org/10.46586/tches.v2024.i3.224-248

[7] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014, pp. 719–732.

[8] C. Percival, "Cache missing for fun and profit," 2005.

[9] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[10] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[11] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519," in *CCS*. ACM, 2017, pp. 845–858. [Online]. Available: https://doi.org/10.1145/3133956.3134029

[12] Z. Zhang, M. Tao, S. O'Connell, C. Chuengsatiansup, D. Genkin, and Y. Yarom, "Bunnyhop: Exploiting the instruction prefetcher," in *USENIX Security*, 2023, pp. 7321–7337.

[13] R. L. Schröder, S. Gast, and Q. Guo, "Divide and surrender: Exploiting variable division instruction timing in HQC key recovery attacks," in *USENIX Security*, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/schr%C3%B6der

[14] D. J. Bernstein, K. Bhargavan, S. Bhasin, A. Chattopadhyay, T. K. Chia, M. J. Kannwischer, F. Kiefer, T. Paiva, P. Ravi, and G. Tamvada, "KyberSlash: Exploiting secret-dependent division timings in kyber implementations," Cryptology ePrint Archive, Paper 2024/1049, 2024. [Online]. Available: https://eprint.iacr.org/2024/1049

[15] Q. Guo, C. Hlauschek, T. Johansson, N. Lahr, A. Nilsson, and R. L. Schröder, "Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE," *CHES*, vol. 2022, no. 3, pp. 223–263, 2022. [Online]. Available: https://doi.org/10.46586/tches.v2022.i3.223-263

[16] Z. Lai, R. Zhang, Z. Zhang, J. Hermelink, M. Schwarz, V.-T. Pham, and U. Parampalli, "You only decapsulate once: Ciphertext-independent single-trace passive side-channel attacks on hqc," *Cryptology ePrint Archive*, 2025.

[17] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security*, 2016, pp. 53–70.

[18] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015, pp. 605–622.

[19] D. Evtyushkin, D. V. Ponomarev, and N. B. Abu-Ghazaleh, "Jump over ASLR: attacking branch predictors to bypass ASLR," in *MICRO*, 2016, pp. 40:1–40:13. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783743

[20] D. Evtyushkin, R. Riley, N. B. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *ASPLOS*, 2018, pp. 693–707. [Online]. Available: https://doi.org/10.1145/3173162.3173204

[21] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *CCS*, 2014, pp. 1267–1279. [Online]. Available: https://doi.org/10.1145/2660267.2660283

[22] L. Simon, D. Chisnall, and R. J. Anderson, "What you get is what you C: controlling side effects in mainstream C compilers," in *IEEE EuroS&P*, 2018, pp. 1–15. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00009

[23] M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun, "Breaking bad: How compilers break constant-time implementations," in *ASIA CCS*, 2025, pp. 1690–1706. [Online]. Available: https://doi.org/10.1145/3708821.3733909

[24] L. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *IEEE SP*, 2020, pp. 1021–1038. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00074

[25] A. Geimer and C. Maurice, "Fun with flags: How compilers break and fix constant-time code," *arXiv preprint arXiv:2507.06112*, 2025.

[26] D. J. Bernstein, "The cryptoint library," 2025.

[27] L. Gerlach, R. Pietsch, and M. Schwarz, "Do compilers break constant-time guarantees?" in *FC*, 2025.

[28] T. Pornin, "Constant-time code: The pessimist case," *Cryptology ePrint Archive*, 2025.

[29] F. Willi, "Identifying compiler optimizations that break constant time programming techniques," 2025.

[30] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, "When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015," in *International Conference on Cryptology and Network Security*. Springer, 2016, pp. 573–582.

[31] (2024) Clangover cve. https://nvd.nist.gov/vuln/detail/CVE-2024-37880,Accessed: 2025-7-27. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-37880

[32] L. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Symbolic binary analyzer for security with applications to constant-time and secret-erasure," *ACM Trans. Priv. Secur.*, vol. 26, no. 2, pp. 11:1–11:42, 2023. [Online]. Available: https://doi.org/10.1145/3563037

[33] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA - differential address trace analysis: Finding address-based side-channels in binaries," in *USENIX Security*, 2018, pp. 603–620. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser

[34] Z. Zhang and G. Barthe, "CT-LLVM: automatic large-scale constant-time analysis," *IACR Cryptol. ePrint Arch.*, p. 338, 2025. [Online]. Available: https://eprint.iacr.org/2025/338

[35] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

[36] OpenSSL. (2026) Security policy. https://openssl-library.org/policies/general/security-policy/, Accessed: 2026-1-12. [Online]. Available: https://openssl-library.org/policies/general/security-policy/

[37] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - kyber: A cca-secure module-lattice-based KEM," in *EuroS&P*, 2018, pp. 353–367. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00032

[38] C. Ferneding, "Analysis of vulnerability cve-2024-3094 in xz utils: Precedent supply chain attack and incident analysis," 2024.

[39] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE SP*, 2015, pp. 623–639. [Online]. Available: https://doi.org/10.1109/SP.2015.44

[40] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *USENIX Security*, 2017, pp. 69–81. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/kohlbrenner

[41] J. Jin, E. McMurtry, B. I. P. Rubinstein, and O. Ohrimenko, "Are we there yet? timing and floating-point attacks on differential privacy systems," in *IEEE SP*, 2022, pp. 473–488. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833672

[42] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: taking speculative load hardening to the next level," in *USENIX Security*, 2023, pp. 7125–7142. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh

[43] "Constant-time verification tool," Online, n.d. [Online]. Available: https://crocs-muni.github.io/ct-tools/

[44] (2022) cryptocoding. https://github.com/veorq/cryptocoding. Accessed: 2025-8-24. [Online]. Available: https://github.com/veorq/cryptocoding

[45] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *CHES*, vol. 5747, 2009, pp. 1–17. [Online]. Available: https://doi.org/10.1007/978-3-642-04138-9_1

[46] P. P. Puschner and A. Burns, "Writing temporally predictable code," in *WORDS 2002*, 2002, pp. 85–94. [Online]. Available: https://doi.org/10.1109/WORDS.2002.1000040

[47] LLVM. (2026) Division by constant source code. Accessed: 2026-09-20. [Online]. Available: https://llvm.org/doxygen/DivisionByConstantInfo_8cpp_source.html

[48] T. Granlund and P. L. Montgomery, "Division by invariant integers using multiplication," in *PLDI*, 1994, pp. 61–72. [Online]. Available: https://doi.org/10.1145/178243.178249

[49] D. J. Bernstein and T. Lange, "ebacs: Ecrypt benchmarking of cryptographic systems," 2011.

[50] T. of Bits. (2025) Introducing constant-time support for llvm to protect cryptographic code. Accessed: 2025-12-17. [Online]. Available: https://blog.trailofbits.com/2025/12/02/introducing-constant-time-support-for-llvm-to-protect-cryptographic-code/

[51] Intel. (2022) Guidelines for mitigating timing side channels against cryptographic implementations. Accessed: 2025-12-17. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html

[52] LLVM. (2025) Llvm link time optimization: Design and implementation. Accessed: 2025-12-29. [Online]. Available: https://llvm.org/docs/LinkTimeOptimization.html

[53] T. Pornin. (2017) Bearssl: Ssl for all things. Accessed: 2026-01-06. [Online]. Available: https://www.bearssl.org/BearSSL-BSidesEdinburgh2017.pdf

[54] Intel. (2025) Data operand independent timing instruction set architecture (isa) guidance. Accessed: 2026-8-25. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html

[55] A. Langley, "Ctgrind," https://www.imperialviolet.org/2010/04/01/ctgrind.html, 2010.

[56] LLVM. (2026) Using the new pass manager. Accessed: 2026-1-6. [Online]. Available: https://llvm.org/docs/NewPassManager.html

[57] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "Improving software quality in cryptography standardization projects," in *EuroS&P Workshops*, 2022, pp. 19–30. [Online]. Available: https://doi.org/10.1109/EuroSPW55150.2022.00010

[58] NIST. (2025) Post-quantum cryptography selected algorithms. Accessed: 2025-12-24. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/selected-algorithms

[59] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *LATINCRYPT*, vol. 7533, 2012, pp. 159–176. [Online]. Available: https://doi.org/10.1007/978-3-642-33481-8_9

[60] A. Geimer, M. Vergnolle, F. Recoules, L. Daniel, S. Bardin, and C. Maurice, "A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries," in *CCS*, 2023, pp. 1690–1704. [Online]. Available: https://doi.org/10.1145/3576915.3623112

[61] S. Deng, Z. Liu, S. Wang, and Y. Zhang, "An empirical study measuring in-the-wild cryptographic microarchitectural side-channel patches," in *CCS*, 2025, pp. 648–662. [Online]. Available: https://doi.org/10.1145/3719027.3744881

[62] LLVM. (2025) Instcombine source code. Accessed: 2026-1-11. [Online]. Available: https://llvm.org/doxygen/InstructionCombining_8cpp_source.html

[63] ——. (2025) Selectoptimize source code. Accessed: 2025-12-22. [Online]. Available: https://llvm.org/doxygen/SelectOptimize_8cpp_source.html

Appendix A.
Ground Truth Validation Test Cases

We conduct our sanity analysis with the following 19 examples picked from four prior works.

1) Schneider et al. [23]: godbolt.org/z/nx1crrxv3, godbolt.org/z/bnE1PrfKG, godbolt.org/z/qM9KT31vz, godbolt.org/z/zvh6Pjzar, godbolt.org/z/hhs1KWfz7, godbolt.org/z/xGh6Kjzfh, godbolt.org/z/z71Wv5b7j, godbolt.org/z/6s9daKhxr
2) Daniel et al. [24]: godbolt.org/z/jjK3EjP48, godbolt.org/z/j4dnGb1sP, godbolt.org/z/G3Mhq7Th5, godbolt.org/z/o4b96Wfhz, godbolt.org/z/Y7M9n63bM
3) Lai et al. [16]: godbolt.org/z/fdMWPq5d5, godbolt.org/z/YnxMTh657, godbolt.org/z/MPjq5odz1
4) Zhang and Barthe [34]: godbolt.org/z/EvsaW56j9, godbolt.org/z/W53n1vKh3, godbolt.org/z/7Gs53hTG6

Appendix B.
List of Upstream-addressed Constant-time Violations

Tables 7, 8, and 10 enumerate, library by library, the compiler-induced constant-time violations addressed upstream in response to our reports. Each row pairs an affected source-code line with the function(s) in which it appears.

Appendix C.
Per-Library Violations across Clang Versions

Complementing the Clang-18 results in Figure 4, Figures 9 and 10 report the number of detected potential constant-time violations per cryptographic library for Clang 14 and Clang 20, respectively.

Appendix D.
Root Causes of Multi-Flag Violations

The multi-flag analysis of Section 5.2.3 reveals five potential violations that single-flag analysis does not induce: three on `x86-64` and two on `AArch64`.

The three `x86-64` cases follow the path of Figure 5: **InstCombinePass** forms a *select* that **X86CmovConversion** lowers into a conditional branch. They pair an inliner configuration (e.g., `--inline-instr-cost=-1`) with `--x86-cmov-converter-force-all`: inlining moves a function into a context where its code becomes equivalent to a select.

In both `AArch64` cases, **SelectOptimize** converts a *select-like* expression introduced by **InstCombinePass** into a conditional branch. The combinations differ only in how they make the pass choose the branch: some modify its cost model (e.g., `--mispredict-default-rate=0`), others reshape the IR it receives (e.g., `--enable-module-inliner`).

None of the five cases involves a pass beyond those in Table 4, and the mitigations discussed in Section 7 prevent all of them.

Clang 14 Optimization Level and Flag	Count
--unroll-count	230
-O2	226
-O3	222
-O1	214
-Os	203
-Oz	201
--fast-isel	141
--unroll-force-peel-count	137
--attributor-enable	95
--instcombine-max-iterations	49
--unroll-threshold	47
--disable-peephole	45
-fno-unroll-loops	38
--enable-module-inliner	37
--unroll-threshold-aggressive	25
--unroll-full-max-count	25
--pre-RA-sched	23
--inline-threshold	20
--two-entry-phi-node-folding-threshold	18
--inline-call-penalty	18
--aarch64-stress-promote-const	15
--aarch64-enable-global-isel-at-O	14
--rotation-max-header-size	14
--partial-unrolling-threshold	14
--unroll-threshold-default	13

TABLE 5: Top 25 Clang 14 optimization flags that introduce the most potential constant-time violations. A case is counted multiple times if it can be introduced by combining the same flag with different optimization levels.

Clang 18 Optimization Level and Flag	Count
--x86-cmov-converter-force-all	638
-fglobal-isel	208
--fast-isel	169
--attributor-enable	116
-O3	79
-O2	73
--disable-loop-level-heuristics	68
--disable-select-optimize	63
-Oz	58
-O1	56
-Os	56
--inline-instr-cost	28
--mispredict-default-rate	25
--partial-unrolling-threshold	22
--select-opti-loop-cycle-gain-threshold	21
--sroa-skip-mem2reg	20
--unroll-count	20
--slp-threshold	18
--unroll-force-peel-count	14
--unroll-partial-threshold	13
--ifcvt-fn-start	12
-ftree-vectorize	11
-fvectorize	11
--unroll-runtime	10
--scalar-evolution-max-arith-depth	10

TABLE 6: Top 25 Clang 18 optimization flags that introduce the most potential constant-time violations. A case is counted multiple times if it can be introduced by combining the same flag with different optimization levels.

TABLE 7: Violations fixed upstream in **Libsodium** and **PQClean** (ML-DSA).

Affected function(s)	**Source-code line**
Libsodium	
_sodium_ge25519_from_hash	`fe_f[0] += (h[31] >> 7) * 19 + (h[63] >> 7) * 722;`
PQClean (ML-DSA)	
PQCLEAN_MLDSA44_CLEAN_use_hint, PQCLEAN_MLDSA44_CLEAN_decompose	`a1 ^= ((43 - a1) >> 31) & a1;`
PQCLEAN_MLDSA65_CLEAN_use_hint, PQCLEAN_MLDSA44_CLEAN_use_hint, PQCLEAN_MLDSA87_CLEAN_use_hint, PQCLEAN_MLDSA87_CLEAN_decompose, PQCLEAN_MLDSA65_CLEAN_decompose, PQCLEAN_MLDSA44_CLEAN_decompose	`*a0 -= (((Q - 1) / 2 - *a0) >> 31) & Q;`

TABLE 8: Violations acknowledged by **MbedTLS**.

Affected function(s)	**Source-code line**
get_one_and_zeros_padding	`return (mbedtls_ct_uint_t) ((condition & if1) | (not_cond & if0));`
mbedtls_rsa_rsaes_pkcs1_v15_decrypt	`return (mbedtls_ct_condition_t) (x | y);`
mbedtls_rsa_rsaes_oaep_decrypt, get_one_and_zeros_padding	`return (mbedtls_ct_condition_t) (x & y);`

Clang 20 Optimization Level and Flag	**Count**
--x86-cmov-converter-force-all	683
-fglobal-isel	659
--global-isel	555
--fast-isel	177
-O2	74
-O3	66
--unroll-count	61
--unroll-force-peel-count	58
-Os	56
-Oz	56
--disable-loop-level-heuristics	56
-O1	55
--attributor-enable	54
--disable-select-optimize	45
--sroa-skip-mem2reg	32
--partial-unrolling-threshold	29
--unroll-threshold	27
--mispredict-default-rate	23
--disable-peephole	18
--force-specialization	17
-fno-unroll-loops	17
--slp-threshold	16
--select-opti-loop-cycle-gain-threshold	16
--unroll-partial-threshold	16
-ftree-vectorize	15

TABLE 9: Top 25 Clang 20 optimization flags that introduce the most potential constant-time violations. A case is counted multiple times if it can be introduced by combining the same flag with different optimization levels.

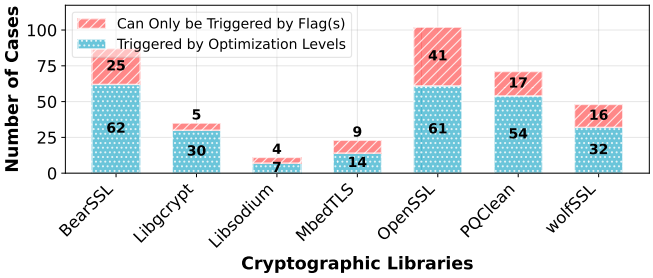

Figure 9: Number of detected Clang-14-induced potential constant-time violations per cryptographic library.

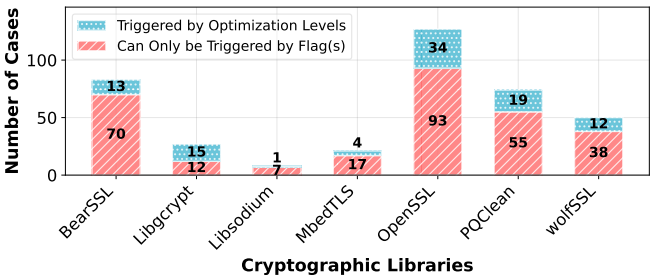

Figure 10: Number of detected Clang-20-induced potential constant-time violations per cryptographic library.

TABLE 10: Violations fixed upstream in **Libgcrypt** and **WolfSSL**.

Affected function(s)	Source-code line
Libgcrypt	
indcpa_enc_4	`t[k] += ((int16_t)t[k] >> 15) & KYBER_Q;`
indcpa_enc_2, indcpa_enc_3	`t[k] += ((int16_t)t[k] >> 15) & KYBER_Q;`
_gcry_cipher_xts_crypt	`u64 carry = -(hi >> 63) & 0x87;`
cbrecursion	`return crypto_int32_x ^ crypto_int32_z;`
_gcry_mceliece6688128f_dec	`mask = mask_nonzero(d) & mask_leq(L*2, N);`
_gcry_mceliece6688128f_dec	`b = (d & mask) | (b & ~mask);`
_gcry_mceliece6688128f_dec	`L = ((N+1-L) & mask) | (L & ~mask);`
_gcry_mceliece6688128f_keypair, layer	`d &= m;`
_gcry_mceliece6688128f_keypair, layer	`p[ i+j ] ^= d;`
_gcry_mceliece6688128f_enc	`e_int[i] |= val[j] & mask;`
_gcry_mceliece6688128f_keypair, gf_mul	`tmp = t0 * (t1 & 1);`
_gcry_mceliece6688128f_keypair	`pi[ row + j ] ^= d;`
_gcry_mceliece6688128f_keypair	`mat[ row ][ c ] ^= mat[ k ][ c ] & mask;`
_gcry_mceliece6688128f_keypair	`mat[ k ][ c ] ^= mat[ row ][ c ] & mask;`
_gcry_mceliece6688128f_keypair	`mat[ k ][ c ] ^= mat[ row ][ c ] & mask;`
mont_mul, _gcry_mpih_powm_lli, mont_reduc	`& mpih_limb_is_zero (cy1));`
do_setkey	`rcon = ((rcon << 1) ^ (-(rcon >> 7) & 0x1b)) & 0xff;`
_gcry_aes_xts_crypt	`carry = -(tweak_next_hi >> 63) & 0x87;`
_gcry_sntrup761_dec, Hide, Decode, _gcry_sntrup761_keypair, Rq_mult_small	`x += mask & (uint32_t) m;`
_gcry_sntrup761_dec, Hide, _gcry_sntrup761_keypair, Rq_mult_small	`ur += mask & m;`
_gcry_sntrup761_keypair	`swap = int16_t_negative_mask (-delta) & int16_t_nonzero_mask (g[0]);`
_gcry_sntrup761_keypair	`swap = int16_t_negative_mask (-delta) & int16_t_nonzero_mask (g[0]);`
_gcry_sntrup761_dec	`differentbits |= (*c++) ^ (*c2++);`
_gcry_sntrup761_dec	`r_enc[i] ^= mask & (r_enc[i] ^ rho[i]);`
WolfSSL	
TimingPadVerify	`good |= mask & (data[sz - 1 - i] ^ paddingSz);`
TimingPadVerify	`scanStart &= ctMaskIntGTE(scanStart, 0);`
TimingPadVerify	`macStart &= ctMaskIntGTE(macStart, 0);`
TimingPadVerify	`mac[j] |= started & notEnded & data[i + j];`
ProcessReplyEx	`padding &= ctMaskLTE((int)i,`
Hmac_UpdateFinal_CT, TLS_hmac	`return (byte)((b & ((byte)~(word32)m)) | (a & m));`
Hmac_UpdateFinal_CT, TLS_hmac	`b &= (unsigned char)~(word32)pastEoc;`
wc_AesGcmDecrypt	`ret = (ret & ~res) | (res & WC_NO_ERR_TRACE(AES_GCM_AUTH_E));`
wc_DhAgree_ct	`agree_src += (mask & requested_agreeSz);`
wc_Sha256Update, SendTls13Certificate, DoTls13HandShakeMsg, wc_HKDF_Expand_ex, SendCertificate, DoCertificateVerify, wolfSSL_GetSessionClient, wc_PBKDF2_ex, TlsSessionCacheGetAndLock, wc_PKCS12_PBKDF_ex, DoServerKeyExchange, BuildMessage, wc_PBKDF1_ex, Sha512Update, wc_Chacha_Process, AddCertExt, DoHandShakeMsgType, wc_Md5Update, _InitRng, wc_ShaUpdate, wolfSSL_get_shared_ciphers, wc_DhGenerateKeyPair, ProcessReplyEx	`return (a & ~gte_mask) | (b & gte_mask);`
wc_RsaUnPad_ex	`return (((word32)b & (word32)(~(signed int)(signed char)m)) |`
wc_RsaUnPad_ex	`inc &= 1 - (((word32)0 - pkcsBlock[i]) >> 31);`
wc_RsaUnPad_ex	`invalid |= (byte)~pastSep;`
wc_RsaPrivateDecrypt, wc_RsaPrivateDecryptInline	`invalid |= ctMaskNotEq(pkcsBlock[0], 0x00);`
wc_RsaUnPad_ex	`ret = ((int)-1 + (int)(invalid >> 7)) & ((int)pkcsBlockLen - i);`