Do Compilers Break Constant-time Guarantees?

Lukas Gerlach[1], Robert Pietsch[2], and Michael Schwarz[1]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
[2] Saarland University, Saarbrücken, Germany

Abstract. Side-channel attacks are a significant concern for the implementation of cryptographic algorithms. Data-oblivious programming is a discipline that helps mitigate side-channel attacks by preventing data leakage over side channels. However, due to various optimizations in modern compilers, data-obliviousness cannot be guaranteed in high-level languages. This work investigates to which extent compiler optimizations violate data-obliviousness. To this end, we present data-oblivious compiler checker (DOCC), an automated binary testing pipeline for detecting data-obliviousness violations under different compiler configurations. We show that DOCC is applicable across 6 widely used compilers. Additionally, DOCC can retrofit existing analysis tools with advanced leakage models, such as data-dependent instruction execution times and data-obliviousness under speculation. We evaluate DOCC on 5 major cryptographic libraries and the recently proposed NIST lightweight cryptography primitives. We reveal data-obliviousness violations in 93 out of the 127 tested algorithms and 1845 out of the 12 917 test cases across different cryptographic libraries, building blocks, and programming languages. We demonstrate that the choice of compiler and optimizations heavily influences the resulting binary's properties.

1 Introduction

Side-channel attacks threaten software security, compromising the confidentiality of functionally correct software systems. Examples of side-channels are the execution time of programs [12, 34, 60], the state of data in caches [47, 67], and the processor's power consumption [42, 35]. Attackers can observe side channels through software without physical access to the device. Side-channels often affect widely used libraries, such as OpenSSL [45], as evidenced by past CVEs such as CVE-2022-4304, CVE-2019-1547, and CVE-2018-5407. Side-channel leakage is typically discovered manually and addressed on a case-by-case basis. To prevent exploitation of such side channels, developers must write software in a side-channel-resilient manner. Constant-time or data-oblivious programming [28, 51] prevents software side channels by making the execution time, code, and data access patterns of programms independent of secret values.

However, testing whether a program is data-oblivious is challenging for several reasons. While a variety of different checking tools provide non-formal [61, 36, 63, 64, 53] as well as formal guarantees of data-obliviousness [14, 16], these

tools are not typically integrated into modern build systems and often have significant limitations both technically and from a usability perspective [24, 29]. Moreover, many checking tools consider a limited execution model based solely on architectural observations and do not consider leakage in speculative execution [8]. Furthermore, current approaches rarely model variable execution time instructions, which can increase the attack surface [3]. Finally and most importantly, data-obliviousness can only be defined at the machine-code level, not for high-level C code [30]. This lack of a clear definition is due to the compiler's freedom to perform optimizations that do not alter observable behavior [19].

In this paper, we therefore ask the following question:

Do compilers influence data-obliviousness, and can we automatically detect violations by combining different detection techniques?

We present DOCC, a systematic approach assessing the data-obliviousness of compiled code under various compiler parameters. DOCC integrates different compilers and testing tools in a flexible and expandable framework. It includes binary-rewriting-based techniques that enhance existing and enable additional testing capabilities for data-obliviousness. As a result, our framework can check properties such as data-obliviousness in speculative execution or data-dependent instruction execution. DOCC is efficient because it implements testing in 3 stages. In the first stage, DOCC conducts computationally inexpensive heuristic tests to determine if the binary is data-oblivious. If the binary passes the heuristic tests, DOCC proceeds to the second stage, which involves dynamic execution. If the tests of the second stage are passed, the third stage employs symbolic execution to verify data-obliviousness formally. Ultimately, DOCC offers 2 outcomes: Failure at any stage indicates non-data-obliviousness, and successful verification at the third stage denotes data-obliviousness.

To evaluate DOCC, we analyze data-oblivious building blocks [66, 5] and entire cryptographic algorithms from widely used libraries. We identify building blocks where seemingly data-oblivious code compiles to machine code violating data-obliviousness and also cases where code violating data-obliviousness compiles to data-oblivious executables. We evaluate the OpenSSL [45], BearSSL [49], mbedTLS [4], wolfSSL [65] and libsodium [38] cryptographic libraries. We analyze AES, Aria, ChaCha20, Camellia, and SHA implementations of OpenSSL, BearSSL, mbedTLS, wolfSSL, and libsodium using DOCC. We discover that OpenSSL's AES implementation exhibits key-dependent memory accesses when compiled using the `no-asm` flag. Similarly, AES, Aria, and Camellia commonly rely on non-data-oblivious lookup tables. In contrast, analyzing BearSSL and libsodium produces evidence supporting their data-obliviousness claims. Additionally, we analyze candidates of the NIST lightweight cryptography competition [43]. We discover that 9 out of the 10 reference implementations of the current NIST lightweight cryptographic competition are not data-oblivious.

Our evaluation shows that seemingly data-oblivious code only sometimes leads to data-oblivious binaries. We emphasize that employing DOCC enables automatic detection of issues, e.g., in a continuous-integration pipeline.

Contributions. The main contributions of this paper are:

1. We systematically evaluate how compilers influence data-obliviousness.
2. We introduce an extensible framework combining binary rewriting passes with data-obliviousness checkers in a 3-stage pipeline that outperforms single data-obliviousness checking tools.
3. We analyze the NIST lightweight cryptography finalists and 5 cryptographic libraries, revealing 1063 data-obliviousness violations in 53 of 76 algorithms.

We publish our DOCC framework as open source.[3]

Responsible Disclosure We disclosed our findings to OpenSSL on August 30, 2023. Monero removed the OpenSSL `no-asm` flag from their codebase to guarantee a constant-time AES implementation.

2 Background

2.1 Software Side Channels

Software side-channel attacks exploit meta-information leaked during regular system operation without requiring physical access or hardware bugs. Secret-dependent control and data flows are common leakage sources [28]. *Timing attacks* exploit differences in execution time resulting from secret-dependent control flow. Particular instructions, e.g., multiplication, division, and shifting, can also introduce timing variations based on operand values [51, 3]. *Hardware caching* is vulnerable to side channels since it induces timing differences depending on whether a memory address was previously accessed. There are various variants of cache attacks, with shared memory (e.g., Flush+Reload [67]) and without shared memory (e.g., Prime+Probe [47]). *Branch predictors* predict likely outcomes of branches to speed up instruction fetches in case of a stalled branch condition. Branch prediction side channels exploit secret-dependent timing differences by mistraining branch predictors [2].

2.2 Assisted Data-oblivious Programming

To defend against side-channel attacks, *data-oblivious programming* [34, 50, 51, 5] ensures that the sequence of memory accesses and control flow does not depend on secret data. In the past, programmers relied on manual auditing the compiler output for the absence of information leakage [29]. Nowadays, a wide variety of assistance methods to test for data-obliviousness has emerged [24].

Heuristic-based Approach. DUDECT collects program execution times for different program arguments [53]. The resulting execution times are analyzed statistically for secret-dependent execution times. This approach is inherently limited to the path coverage generated by the respective program arguments.

Taint Tracking & Memory Tracing. Another approach is to trace memory and instruction access patterns during the program's execution. CT-GRIND [36] performs *taint tracking* [17] to detect dependencies between memory

[3] `https://github.com/cispa/constant-time-compilers`

accesses and program arguments. Taint tracking propagates additional information (taint) during program execution to determine whether program arguments labeled as secret end up in branches or memory accesses. Similarly, DATA [61] is built upon Intel's binary instrumentation framework Pin [27], which allows for *address-tracing*, capturing a trace of the instruction and memory accesses of a program. DATA performs statistic tests to determine if there is a difference between sets of traces collected on different secret program inputs. Both taint tracking and memory tracing are dynamic approaches that can only test the parts of a program covered by the inputs during testing.

Symbolic Execution. *Symbolic execution* [32] symbolizes the program state to derive logical constraints, which can be used with a logic solver to obtain guarantees for each possible program execution under test. Unlike dynamic testing, symbolic execution can derive strong guarantees that hold for the entire program under test. Pitchfork [15] leverages symbolic execution to detect secret data flowing into address calculations or branch conditions.

2.3 Static Binary Rewriting

Binary rewriting is a technique that allows the modification of compiled binary code without recompilation [62]. The remainder of the paper builds upon e9patch [20] which uses trampolines patched into the code and, therefore, requires no changes to the control and data flow of the remaining unpatched code. Thus, the data-obliviousness properties of patched code remain unchanged.

3 Overview and Design of DOCC

We introduce DOCC, an automated approach testing data-obliviousness of machine code compiled with different compilers and compiler options under different testing strategies. We build a generic, extensible framework that performs compilation in different compiler configurations and tests the resulting binaries for data-obliviousness violations. In addition, DOCC supports binary rewriting passes to retrofit existing data-obliviousness analysis tools with new features, such as detecting violations under speculation or from instructions with non-constant runtimes. Moreover, our debranching technique can reduce the complexity of applications, improving testing performance.

At the core of our testing framework lies the code snippet under test. To determine how the snippet under test is compiled and run, we provide a checker- and compiler-agnostic testing specification. This specification must be provided once per snippet under test. Multiple compilers and architectures can be part of a generic pipeline layout. Instead of relying on a static pipeline for a single architecture incorporating a fixed number of compilers and methods to check each binary for data-obliviousness, DOCC is a flexible framework. For the remainder of the paper, we focus on the x86 architecture due to the variety of available compilers.

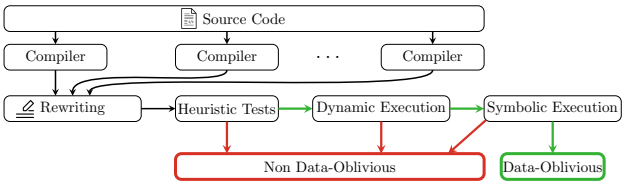


Fig. 1: Three-phase design of DOCC. The input code is compiled with different compilers, rewritten, and checked with increasingly powerful testing methods. Each code snippet is ultimatively categorized as data-oblivious or not.

We employ a 3-stage checking approach (with an additional optional binary rewriting pass), as shown in Figure 1, to find violations reliably and quickly. Before testing, the binary can be rewritten to enable more efficient checking of data-obliviousness violations or to test specific properties such as data operand-dependent instruction timing or data-obliviousness under speculative execution. The first 2 stages leverage heuristic tests and dynamic execution to find violations of data-obliviousness. The main intuition is that showing a violation of data-obliviousness requires only one such violating input (pair), while guaranteeing data-obliviousness requires showing that no such violating inputs exist. Hence, if DOCC cannot find inputs that violate data-obliviousness during a fixed test time or coverage, it resorts to the computationally more expensive verification of data-obliviousness in the third stage.

3.1 (Optional) Stage 0: Binary Rewriting

We introduce 4 novel rewriting approaches that broaden the applicability of already developed data-obliviousness analysis techniques. Instead of changing the checker, we change the binary under test such that the desired property can be evaluated using an off-the-shelf checker, while keeping its data-obliviousness properties. Our 4 rewriting strategies are:

- Rewriting non-constant-time instructions into data and control flow leakage.
- Debranching to reduce the false positive rate and increase the analysis speed of taint tracking and symbolic-execution-based checkers.
- Speculative execution emulation to evaluate the data-obliviousness properties of a binary under speculative execution.
- Coverage instrumentation to evaluate the completeness of the tests.

Non-Constant-Time Instruction Rewriting Execution times for most instructions are typically independent of operand values, but there are exceptions, such as integer divisions and shift operations [51, 1]. Such timing differences have been exploited in browser-based and cryptographic attacks [3]. However, such leakage is not detectable with off-the-shelf checkers. We present a two-stage approach to identify such non-constant-time instructions. First, we detect whether an instruction's execution time is influenced by its operands using statistical tests. Next, we make these instructions detectable via code rewriting.

Our testing method, inspired by test vector leakage assessment [55], involves evaluating instructions on two classes of inputs: one with fixed values and the other with randomly sampled values. Execution times are measured using randomly interleaved inputs from these two classes. A statistical test, specifically a Welsch t-test, is then applied to determine if there is a significant difference between the two sets of timing measurements. If the t-test reveals a significant difference (with a threshold of 4.5 [55]), we conclude that the instruction's timing likely depends on its operands.

In the second step, we rewrite variable time instructions into data and control flow leakage. Each instruction is replaced with a snippet of code called a trampoline, which performs artificial data and control flow constructs. The trampoline is designed such that any bitwise change in the input arguments causes a change in data and control flow. After the trampoline, the patched instruction is executed normally. Therefore, the semantics of the binary under test do not change. In addition, the patching itself does not induce new secret-dependent control or data flow. Therefore, the data-obliviousness properties of the binary are only changed by the contents of the trampoline.

We tested our rewriting pass on the RC5 implementation [46] of OpenSSL, which contains secret-dependent shift instructions in both the `E_RC5_32` and `D_RC5_32` macro. We confirmed that the shift instruction has a different, operand-dependent runtime when executed on an Intel i9-12900K CPU. DOCC automatically rewrites the OpenSSL binary to detect the shift instruction in our patched version using DATA.

We additionally test our rewriting pass on the examples snippets tested in Section 5.1. These examples contain 296 shift instructions on average. However, using a combination of DATA and our rewriting pass, we verify that none of them are secret dependent. By manually verifing that no secret dependent shift instructions exist in the resulting binaries we find that our binary rewriting pass is effective for our examples.

Debranching Both symbolic execution and taint tracking can produce false positives, such as in code like `if(never_taken & secret) func()`. Here, a secret value flows into a never-taken branch, leading to a false positive data-obliviousness violation. We implemented a binary rewriting strategy because addressing the issue at the source code level is tedious and ineffective; compiler optimizations can alter the binary's properties again.

Our approach benchmarks the program on random, well-formed inputs until sufficient coverage is achieved, using a custom Quiling-based tracer [23]. This tracer records all branches and whether they were taken, allowing us to identify branches that are always or never executed during regular operation. These static branches, often part of assertions, are then rewritten as unconditional branches. Since assertions are always true for valid input, they can safely be ignored in data-obliviousness tests.

We applied this technique to a base64 decoding routine in CTTK [50], where comparisons with whitespace—flagged as violations by both CTGRIND and PITCH-

FORK—were rewritten as unconditional branches, resolving the false positive. Similar issues were found and addressed in the mbedTLS base64 implementation and the hex decoding in libsodium. In all other tests debranching was not able to find and eliminates stale branches as they did not occur. In such cases, debranching does not reduce false positives as it simply has no effect.

Speculative Execution Emulation Typically, data-obliviousness checks focus on architectural execution. However, speculative execution can expose additional attack surfaces [33, 13]. An example of such a program contains a branch that is never architecturally taken, e.g., an assertion statement. Under speculative execution, an attacker can trick the branch predictor into entering the branch. If the branch contains a non-data-oblivious statement leaking the secret variable, i.e., a memory lookup, the attacker can leak the content of the variable [33, 13].

Rather than integrating speculative execution checks into the tools used by DOCC, we use binary rewriting to emulate speculative execution in two steps. First, we execute the binary while recording all conditional branches and their coverage. Once sufficient coverage is achieved, we identify static branches that are candidates for speculative execution rewrites, particularly those never taken during normal execution but potentially vulnerable under speculation. We rewrite each static branch by inverting its condition, similar to the approach used in SpecFuzz [44], effectively emulating speculative execution. This process is repeated for all branches in the binary.

We evaluated the rewriting approach on the wolfSSL library, where the Poly1305 authenticated encryption showed additional leakage during speculation. However, this leakage induced by an error-checking branch without a speculative leakage gadget inside the branch is likely not exploitable.

Coverage Instrumentation Dynamic testing methods only partially test a binary, resulting in data-obliviousness guarantees only for the covered code. We provide a coverage-instrumentation rewriting pass using bcov [10] to measure coverage. Rewriting the binary for coverage has the benefit that it is independent of the compiler. We tested our coverage of the mbedTLS [4] AES implementation and observed full test coverage on the encryption and decryption routines.

Takeaway Binary rewriting of programs under test is a viable alternative to adding functionality to individual checkers.

3.2 Stage 1: Inexpensive Heuristic Checks

The first stage of our pipeline uses efficient heuristic checks to identify potential data-obliviousness violations quickly. We employ a customized version of DUDECT [53], which runs the code snippet while measuring execution time and tracking microarchitectural events through performance counters. Performance counters allow for tracking events that are hard to detect using timing only,

such as cache misses. These events only produce timing differences on specifically crafted inputs, which are often not covered by DUDECT, as inputs for each execution are generated randomly for secret inputs and fixed for public inputs. If, during execution, the snippet induces statistically significant secret-dependent timing or counter differences, it is labeled as non-data-oblivious.

Despite its efficiency, this approach has limitations. False negatives may occur because simple heuristics might miss subtle violations, while false positives can arise due to measurement noise or runtime variance caused by external factors like interrupts. Stage 2 employs a more powerful dynamic test to verify negative results. In the case of false positives, the checks can be repeated or refined using more robust tests.

Takeaway Heuristic checks can be extended using performance counters to detect more subtle data-obliviousness violations.

3.3 Stage 2: Moderately-expensive Dynamic Execution

When heuristic tests fail to detect data-obliviousness violations, we move to more powerful dynamic execution methods. For instance, runtime-based heuristics are often insufficient for detecting secret-dependent memory accesses, as these do not always cause measurable timing differences—such as in small lookup tables where entries are mostly cached. Additionally, hardware optimizations like dynamic frequency scaling or prefetching can distort timing results, leading to inaccuracies.

We employ 2 different strategies of increasing computational power but also complexity, namely recording execution traces [61, 63, 64] and taint tracking [36]. Taint tracking can detect data-dependent edge cases that are unlikely to be found with trace recording as taint allows to track secret-dependent computations independent of concrete values.

While these techniques only cover executed code paths, this is not a significant problem, as cryptographic implementations contain few branches. Like previous work, generating random inputs is sufficient to achieve good coverage [61, 53]. Our coverage instrumentation rewriting pass described in Section 3.1 can validate this. If dynamic execution finds no data-obliviousness violations, DOCC transitions to symbolic execution in Stage 3.

3.4 Stage 3: Expensive Symbolic Execution

If dynamic execution fails to detect data-obliviousness violations, we proceed to symbolic execution [7, 32]. Symbolic execution explores all possible paths through a program, providing a complete analysis. The code is considered data-oblivious if no violations are found during this phase.

Symbolic execution for verifying data-obliviousness has been studied extensively in previous works [16, 15]. Unlike taint tracking, which dynamically executes the program, symbolic execution solves constraints over inputs to deter-

mine whether a secret input can influence a branch or memory access. If it finds such a case, the program leaks sensitive information.

Symbolic execution is the only approach employed by DOCC to provide strong guarantees. However, to do so, the entire program must be explored, which is only feasible if the generated program contains a manageable number of branches. For more complex programs, symbolic execution suffers from the problem of path explosion [7]. It is, therefore, more helpful in finding data-obliviousness violations than in proving their absence.

3.5 Result Analysis and Option Triage

Our pipeline allows to discover the root cause of data-obliviousness violations on the source code as well as compiler optimization level. Root cause analysis on the source code level is supported by data-obliviousness-checking tools [36, 61]. We provide an automated approach to detect which compiler optimization leads to the data-obliviousness violation. We first find the optimization level at which a binary is no longer data-oblivious or becomes data-oblivious. Then, we remove and add subsets of the optimization flags until the binary becomes data-oblivious again. We perform a search over the optimization flags, finding a minimal data-oblivious configuration. We evaluate this approach on the code samples in Section 5.1 with the gcc compiler. With `gcc` we only observe examples where code is transformed to constant time code in our building block tests. This is unanimously the case due to the `-fipa-pure-const` optimization. This optimization eliminates constants and allows artificial non-constant-time constructs to be optimized away. For `clang` we implemented a similar approach, here we do not observe a single flag that causes switches in constant time behavior. We tested up to a combination of 3 flags, but optimization pass ordering and the presence of multiple interdepent flags make exhaustie testing infeasible.

4 Implementation

In this section, we discuss the implementation details of DOCC that we use for our evaluation (Section 5.1 and Section 5.2). For our proof-of-concept implementation, we use 7 different compilers and 4 different checkers for data-obliviousness. Note that the implementation can be arbitrarily extended to use different compilers or checkers.

Compilers. We currently support GCC (`gcc`), Clang (`clang`), Intel C++ compiler (`icc`), AMD Optimizing C/C++ Compiler (`aocc`), Tiny C Compiler (`tcc`), CompCert C Compiler (`compcert`), and Zig (`zigcc`). This set of compilers includes the most commonly used openly accessible compilers. We support various optimization levels, including `-O1`, `-O2`, `-O3`, `-Ofast`, and `-Os`, to capture a range of compiler optimizations, along with the `-Obranchless` option for the CompCert compiler, which aims to minimize branching.

Checkers. We currently support DUDECT, DATA, CTGRIND, and PITCHFORK. The PITCHFORK checker was chosen over the similar BINSEC/REL checker,

as it supports 64-bit binaries. As described in Section 2.2, these methods can be categorized into heuristic tests (DUDECT), dynamic taint tracking (CTGRIND), address-tracing (DATA), and symbolic execution (PITCHFORK).

To further reduce execution time, DOCC parallelizes tests whenever possible. All tests, except heuristic ones that may lose precision under system load, can run in parallel up to the number of available cores and memory.

5 Evaluation

In the following, we evalaute DOCC across a wide range of constant-time building blocks and cryptographic libraries. The normalized results of each tests are shown in Figure 2, more detailled results are provided as part of our artifact.

5.1 Building-block Analysis

In this section, we apply DOCC to 9 common building blocks for data-oblivious code. We also test the non-constant time instruction rewriting pass and the debranching rewriting pass on these building blocks as stated in Section 3.1 Our study identifies 312 of 2525 test cases across 14 of 24 tested code snippets as non-data-oblivious. We test DOCC on building blocks suggested in the literature for writing data-oblivious code [66, 28, 5]. We provide the details about the tested building blocks in Appendix B. In Appendix A, we test the Constant Time Toolkit (CTTK) [50] library containing primitives for data-oblivious programming. Our analysis reveals that code appearing data-oblivious at the source level can be compiled into non-data-oblivious machine code and vice versa.

Takeaway Compiler options, e.g., optimizations, influence the data-obliviousness properties of the resulting binaries in arbitrary ways.

33 % of the samples produce a non-data-oblivious binary from ostensibly data-oblivious C code. In contrast, 56 % of the samples that are not data-oblivious on the source level result in a data-oblivious binary. The CTTK library remained data-oblivious independent of compilation options, showing that it is possible to create robust and portable data-oblivious code.

Testing small snippets helps evaluate the effectiveness and precision of various data-obliviousness-checking methods. However, each method can yield false results for different reasons. Simple heuristics may produce false positives due to measurement noise, misinterpreting errors as secret-dependent differences.

More sophisticated methods, such as taint analysis and symbolic execution, may also be overly sensitive, misclassifying branches that are never taken—like assertions—as leakage. For instance, in testing the CTTK library, a branch that skips non-secret whitespace was incorrectly flagged as a violation. To address these issues, we propose binary rewriting, as discussed in Section 3.1.

Takeaway Checkers disagree on data-obliviousness violations, but an ensemble of checkers can detect data-obliviousness violations reliably.

5.2 Cryptographic Library Analysis

This section analyzes cryptographic libraries as real-world case studies. We analyze implementations from the NIST lightweight cryptography competition,and 5 popular cryptographic libraries (OpenSSL, wolfSSL, Mbed TLS, BearSSL, and libsodium). Due to scalability issues these tests are all performed without the rewrtiting step.

NIST Lightweight Cryptography Finalists We test all 10 NIST lightweight cryptography finalists implementing authenticated encryption with associated data (AEAD). These algorithms should work on devices with little computational power while still providing sufficient security properties. We test multiple implementations per candidate, leading to 28 tested code snippets. Only one finalist variant is free of data-obliviousness violations.

Test Setup. We test the 10 finalist submissions, including the optimized implementation, using all compilers and checkers described in Section 4. Each submission is tested for both encryption and decryption routines.

Results. In 11 of the 28 tested implementations, data-obliviousness depends on the compiler or compiler option. We find that 16 schemes are implemented in a non-data-oblivious way independent of the compiler. For example, the romolus and photon-beetle reference implementations use lookup tables, always resulting in non-data-oblivious binaries. Lastly, ascon is the only finalist that is data-oblivious across all compilers and compiler options (excluding a false positive with the `zig` compiler).

Common Cryptograpic Libraries We evaluate DOCC on 5 commonly used cryptographic libraries: OpenSSL (3.1.1), wolfSSL (5.6.3-stable), Mbed TLS (v3.4.1), BearSSL (v0.6) and libsodium (1.0.19-stable). While previous work already focussed on manual and automatic testing of these libraries [61, 53], we focus on the compilation process and options. As our results show, for OpenSSL, the presence of optimization may cause the resulting library to violate data-obliviousness after compilation.

Tested Functions. We evaluate the AES Aria, and Camellia block ciphers, the ChaCha20 stream cipher and its AEAD counterpart Poly1305. We also test utility functions like base64 encoding. These functions are valid attack targets for cache and timing attacks, making their data-oblivious implementation crucial [69, 6].

Results. Across all libraries, Aria and Camellia use lookup tables, which intrinsically makes these ciphers violate data-obliviousness guarantees. We assume the reason for this is the limited use of those ciphers and their lower performance compared to AES when implemented without lookup tables. All analyzed cryptographic libraries provide a data-oblivious version of AES. However, the compilation flag `no-asm` of the OpenSSL library induces non-data-oblivious behavior in the compiled code. This behavior is only documented in the changelog of version 3.0.2. Our analysis on GitHub shows that multiple projects, including widespread ones such as nginx and the Monero GUI, use this flag. While

OpenSSL acknowledges our finding, they are not considering a change, as they consider this the responsibility of the project using OpenSSL.

ChaCha20 is data-oblivious across all tested libraries. Its design avoids lookup tables and secret-dependent control flow. Likewise, Poly1305, is data-oblivious in all libraries. DOCC verifies the data-obliviousness of ChaCha20 and Poly1305 in 4 of the 5 libraries, others timed out before verification.

The SHA hashing algorithm is either implemented in a data-oblivious way or leads to timeouts during the analysis. Manual inspection of the tests shows that the code is data-oblivious, as SHA does not require branches or lookup tables.

Base64 encoding is implemented data-oblivious independent of the used compiler optimizations for the tested utility functions. However, similar to Section 5.1, we encounter false positives with CTGRIND and PITCHFORK. They also stem from branches never taken on well-formed data and can be easily removed by debranching via binary rewriting.

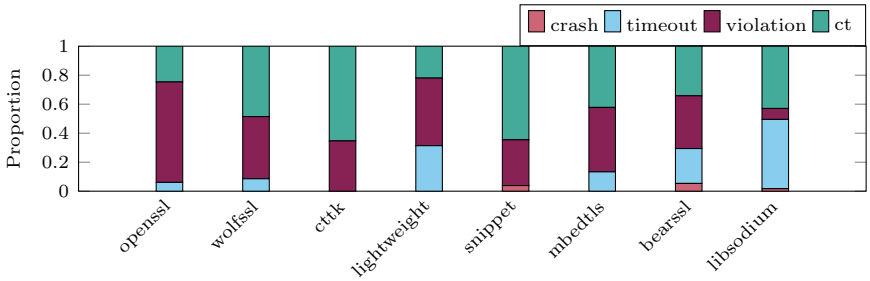


Fig. 2: Normalized data-obliviousness results for the accumulate function in all tested libraries. DOCC finds violations in all tested libraries.

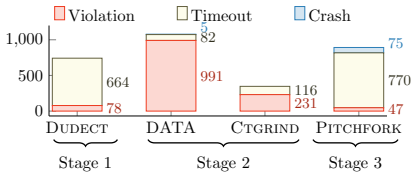
Takeaway Not all cryptographic primitives, even in the most-common cryptographic libraries, are data-oblivious, and libraries may fall back to unsafe implementations silently.

6 Evaluation Outcome

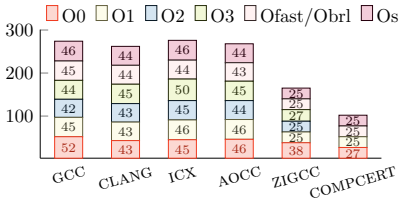
In the following we evaluate the efficacy and performance of DOCC across all tests from Section 5.

Efficacy. We illustrate the efficacy of DOCC in Figure 3a. Our pipeline can capture data-obliviousness violations in each stage, with the dynamic methods in Stage 2 being the most effective. Additionally, we observe that the third stage (symbolic execution) still discovers new data-obliviousness violations.

Figure 3b shows the number of violations observed for each compiler and optimization level. The significantly fewer violations in `compcert` and `zig` are due to the fewer code snippets they could compile. Overall, the relationship between optimization options and the number of violations varies by compiler.



(a) Snippets grouped by result and pipeline stage. If a violation is detected, later pipeline stages are excluded.



(b) Violations for each compiler and optimization level.

Fig. 3: Comparison of pipeline stages and compiler optimizations.

While `gcc` shows slightly more violations at lower optimization levels, LLVM-based compilers like `clang`, `icx`, and `aocc` exhibit slightly more violations at higher optimization levels.

Takeaway Pipelined checks of increasing complexity efficiently and reliably dectect data-obliviousness violations.

Performance. While high accuracy is desirable, it must be achievable within a practical timeframe for DOCC to be usable. We use an Intel Xeon Gold 6346 machine with 32 cores and 128 GB of memory running Ubuntu 22.04, with a test timeout of 2 min. In our tests, we find most violations within seconds, only requiring longer on branchy code or code that enters Stage 3. In addition, we parallelize testing, which scales with the number of cores on the test system. Running all tests in this paper takes around 100 h on our test system. With 12 917 tests overall, this results in an average time of 27 s per test.

As shown in Figure 3a, most timeouts occur in either Stage 1 (heuristic checks) or Stage 3 (symbolic execution). Still, Stage 1 is valuable to our pipeline, as it can capture secret-dependent instruction runtime differences.

7 Discussion

Potential Improvements. Input generation is the limiting factor for dynamic approaches. Fuzzing-based methods [25] with coverage-driven input generation could enhance coverage more quickly. Since DOCC manages input generation and provides coverage instrumentation, integrating this approach could improve efficiency without additional checker modifications.

Supporting more architectures in DOCC presents another avenue for future work. Different architectures come with unique instructions and optimizations, meaning C code that compiles to a data-oblivious binary on one architecture may not retain this property on another.

Related Work. Previous work manually analyzed the impact of compilation on data-obliviousness. Kaufmann [30] found that a seemingly data-oblivious implementation of curve25519 was vulnerable to a side-channel attack when compiled with the MSVC compiler. Additionally, Simon et al. [58] show that specific

cryptographic code snippets are transformed to non data-oblivious ones by the compiler and devise ways to prevent this transformation. Concurrent work [54] analyzed the impact of compiler optimizations on data-obliviousness in cryptographic libraries. However, our focus lies more on the choice of testing tool and compiler, where they test more cryptographic libraries and architectures.

Jancar et al. [29] surveyed data-oblivious programming and automated testing frameworks to check for data-obliviousness. They find that most developers working on cryptographic implementations are unaware of such tools or do not use them. Similarly, Geimer et al. [24] show that using a single checking tool can lead to unreliable results. These results closely relate to our work, as we not only provide more thorough checking of binaries but also unify multiple checking frameworks such that a testing harness only has to be written once.

Much work has gone into automatically rewriting programs into their data-oblivious counterparts [21, 22, 26, 31, 39–41, 52, 59]. FaCT [14] is a programming language that guarantees that the resulting executable satisfies constant-time properties. Similarly, Barthe et al. [9] propose a framework to compile code to constant-time versions that are then proven correct using the Coq proof assistant. In addition, Wu et al. [66] rewrite a program's LLVM code to a data-oblivious version. Program synthesis based approaches have been used to synthesize a constant time instructions from a safe subset of instructions [18]. A similar approach is taken by Borrello et al. [11], where code is dynamically benchmarked to ensure more precise rewriting. Additionally approaches that require a special runtime to enable data-obliviousness have been proposed [37, 56]. However, this approach trades provable data-obliviousness with performance and practicality. All compiler-based approaches require a modified compiler toolchain.

Work on extending the processor ISA [68, 39] aims to fix the data-obliviousness problem on the ISA level.

8 Conclusion

In this work, we proposed DOCC, an automated pipeline for checking data-obliviousness under different compiler optimizations. DOCC efficiently detects violations of data-obliviousness while at the same time being able to give strong guarantees. In 3 case studies, we evaluated the capabilities of DOCC, revealing data-obliviousness violations in 93 of the 127 tested algorithms and 1845 of the 12 917 test cases across cryptographic libraries and building blocks. We show that the choice of compiler and optimizations heavily influences the final binary's properties, making rigorous testing necessary to guarantee data-oblivious code.

Acknowledgment

We thank our reviewers and our shepherd for their valuable feedback. We thank Leon Trampert for fruitful discussions.

References

1. A. Abel, "Automatic generation of models of microarchitectures," 2020.
2. O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.
3. M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P*, 2015.
4. ARM, "mbed TLS," 2020. [Online]. Available: https:///tls.mbed.org
5. J.-P. Aumasson, "Cryptocoding," 2023. [Online]. Available: {https://github.com/veorq/cryptocoding}
6. D. Bae, J. Hwang, and J. Ha, "Flush+ reload cache side-channel attack on block cipher aria," *Journal of the Korea Institute of Information Security & Cryptology*, 2020.
7. R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *CSUR*, 2018.
8. G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe, "High-assurance cryptography in the spectre era," in *S&P*, 2021.
9. Barthe, Gilles and Grégoire, Benjamin and Laporte, Vincent, "Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time"," in *CSF*, 2018.
10. M. A. Ben Khadra, D. Stoffel, and W. Kunz, "Efficient binary-level coverage analysis," in *FSE*, 2020.
11. P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic side-channel resistance using efficient control and data flow linearization," in *SIGSAC*, 2021.
12. B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *ESORICS*, 2011.
13. C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security*, 2019, extended classification tree and PoCs at https://transient.fail/.
14. S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A flexible, constant-time programming language," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 69–76.
15. Cauligi, Sunjay and Disselkoen, Craig and Gleissenthall, Klaus v and Tullsen, Dean and Stefan, Deian and Rezk, Tamara and Barthe, Gilles, "Constant-time foundations for the new spectre era," in *SIGPLAN*, 2020.
16. L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *S&P*, 2020.
17. D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, 1976.
18. S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher, "Synthct: Towards portable constant-time code," in *NDSS*, 2022.
19. G. Dos Reis, B. Stroustrup, and A. Merideth, "Axioms: Semantics aspects of c++ concepts," *ISO/IEC JTC1/WG21 doc*, 2009.
20. G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *ACM SIGPLAN*, 2020.
21. C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *STC*, 2012.

22. C. W. Fletchery, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs," in *HPCA*, 2014.
23. Q. Framework, "quiling: A True Instrumentable Binary Emulation Framework ," 2024. [Online]. Available: https://github.com/qilingframework/qiling
24. A. Geimer, M. Vergnolle, F. Recoules, L.-A. Daniel, S. Bardin, and C. Maurice, "A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries," in *SIGSAC*, 2023.
25. S. He, M. Emmi, and G. Ciocarlie, "ct-fuzz: Fuzzing for timing leaks," in *ICST*, 2020.
26. C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *HPCA*, 2015.
27. Intel Corporation, "Pin - A Dynamic Binary Instrumentation Tool," 2012. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
28. ——, "Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations," 2020. [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
29. J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ""they're not that hard to mitigate": What cryptographic library developers think about timing attacks," in *SP*, 2022.
30. T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, "When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC ," in *CANS*, 2016.
31. T. Kim, M. Peinado, and G. Mainar-Ruiz, "{STEALTHMEM}:{System-Level} protection against {Cache-Based} side channel attacks in the cloud," in *USENIX*, 2012.
32. J. C. King, "Symbolic execution and program testing," *Commun. ACM*, 1976.
33. P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.
34. P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.
35. A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, "Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels," in *USENIX Security*, 2023.
36. A. Langley, "Checking that functions are constant time with Valgrind," 2023. [Online]. Available: {https://github.com/agl/ctgrind}
37. H. B. Lee, T. M. Jois, C. W. Fletcher, and C. A. Gunter, "Dove: A data-oblivious virtual environment," *arXiv preprint arXiv:2102.05195*, 2021.
38. libsodium, "libsodium," 2023. [Online]. Available: https://libsodium.org
39. C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," *SIGPLAN*, 2015.
40. C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," in *CSF*, 2013.
41. M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *SIGSAC*, 2013.

42. S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2008.
43. N. I. of Standards and Technology, "Lightweight cryptography," 2023. [Online]. Available: https://csrc.nist.gov/projects/lightweight-cryptography
44. O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing spectre-type vulnerabilities to the surface," in *USENIX Security Symposium*, 2020.
45. OpenSSL, "OpenSSL: The Open Source toolkit for SSL/TLS," 2019. [Online]. Available: http://www.openssl.org
46. ——, "OpenSSL RC5 implementation," 2024. [Online]. Available: https://github.com/openssl/openssl/tree/master/crypto/rc5
47. D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.
48. B. Pinkas and T. Reinman, "Oblivious ram revisited," in *CRYPTO*, 2010.
49. T. Pornin, "BearSSL: A smaller SSL/TLS library," 2022. [Online]. Available: https://www.bearssl.org
50. ——, "Constant-time toolkit," 2022. [Online]. Available: https://github.com/pornin/CTTK
51. ——, "Why Constant-Time Crypto?" 2022. [Online]. Available: https://www.bearssl.org/constanttime.html
52. A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital {Side-Channels} through obfuscated execution," in *USENIX*, 2015.
53. O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *DATE*, 2017.
54. M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun, "Breaking bad: How compilers break constant-time˜ implementations," *arXiv preprint arXiv:2410.13489*, 2024.
55. T. Schneider and A. Moradi, "Leakage assessment methodology: A clear roadmap for side-channel evaluations," in *CHES*, 2015.
56. F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
57. E. Shi, T. H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o ((log n) 3) worst-case cost," in *ASIACRYPT*, 2011.
58. L. Simon, D. Chisnall, and R. Anderson, "What you get is what you c: Controlling side effects in mainstream c compilers," in *EuroS&P*, 2018.
59. L. Soares and F. M. Q. Pereira, "Memory-safe elimination of side channels," in *CGO*, 2021.
60. D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, 2001.
61. 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 Symposium*, 2018.
62. M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," *CSUR*, 2019.
63. J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "MicroWalk: A Framework for Finding Side Channels in Binaries," in *ACSAC*, 2018.
64. J. Wichelmann, F. Sieck, A. Pätschke, and T. Eisenbarth, "Microwalk-ci: practical side-channel analysis for javascript applications," in *SIGSAC*, 2022.
65. wolfSSL, "wolfSSL: Embedded TLS Library," 2023. [Online]. Available: https://www.wolfssl.com/

66. M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*, 2018.
67. Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.
68. J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," *Cryptology ePrint Archive*, 2018.
69. X.-j. Zhao, T. Wang, and Y. Zheng, "Cache Timing Attacks on Camellia Block Cipher." 2009.

A Tested CTTK Library Functions

In this section, we discuss the tested CTTK primitives, including the testing results and challenges. CTTK is a small library that provides data-oblivious primitives for cryptographic applications.

De- and Encoding Functions. We test 2 standard functions to encode data, namely base64 and hex encoding.

Multiplication. Constant-time multiplication is necessary if the underlying processor does not provide a constant-time multiplication in its instruction set. Some hardware architectures [51] do not or only partially provide constant-time multiplications, so these operations must be emulated by software.

Oblivious RAM (ORAM). ORAM [48, 57] provides a way to perform memory accesses in a data-oblivious way. The implementation provided by CTTK accesses each element in memory when performing a single memory access while arithmetically masking the result.

B Tested Building Blocks

We discuss building blocks that stress compiler optimization and provide a ground truth for DOCC. We provide a data-oblivious and non-data-oblivious variant for each example. Previous work proposed simmilar examples [66].

Array Lookup. This building block performs a lookup into a `uint32_t` array using an index value that is considered secret. The data-oblivious version accesses every array cell and uses arithmetic to select a result. Array lookups are a widely used primitive (e.g., in T-table AES implementations).

String Comparison. This building block compares two-byte strings and returns whether they are equal. Again, string comparison is a widely used operation susceptible to timing attacks [53].

Conditional Select. This building block allows the selection of one of the operands depending on a condition. Such an operation can be implemented manually on systems where the `cmov` instruction or inline assembly is unavailable. It is widely used in cryptographic programming to avoid branches.

Maximum of Integers. This building block computes the maximum of 2 numbers. Like a conditional select, computing the maximum of 2 numbers is a common operation in cryptographic programming (e.g., `inner.h` in BearSSL).