CONSTANTINE: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization

Pietro Borrello
Sapienza University of Rome
borrello@diag.uniroma1.it

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Leonardo Querzoni
Sapienza University of Rome
querzoni@diag.uniroma1.it

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

ABSTRACT

In the era of microarchitectural side channels, vendors scramble to deploy mitigations for transient execution attacks, but leave traditional side-channel attacks against sensitive software (e.g., crypto programs) to be fixed by developers by means of *constant-time programming* (i.e., absence of secret-dependent code/data patterns). Unfortunately, writing constant-time code by hand is hard, as evidenced by the many flaws discovered in production side channel-resistant code. Prior efforts to automatically transform programs into constant-time equivalents offer limited security or compatibility guarantees, hindering their applicability to real-world software.

In this paper, we present CONSTANTINE, a compiler-based system to automatically harden programs against microarchitectural side channels. CONSTANTINE pursues a radical design point where secret-dependent control and data flows are *completely linearized* (i.e., all involved code/data accesses are always executed). This strategy provides strong security and compatibility guarantees by construction, but its natural implementation leads to state explosion in real-world programs. To address this challenge, CONSTANTINE relies on carefully designed optimizations such as just-in-time loop linearization and aggressive function cloning for fully context-sensitive points-to analysis, which not only address state explosion, but also lead to an efficient and compatible solution. CONSTANTINE yields overheads as low as 16% on standard benchmarks and can handle a fully-fledged component from the production wolfSSL library.

CCS CONCEPTS

• **Security and privacy** → **Side-channel analysis and countermeasures**; • **Software and its engineering** → *Compilers.*

KEYWORDS

Side channels; constant-time programming; compilers

ACM Reference Format:
Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. CONSTANTINE: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *Proceedings of ACM Conference on Computer and Communications Security (CCS '21).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Protecting the confidentiality of security-sensitive information is a key requirement of modern computer systems. Yet, despite advances in software security engineering, this requirement is more and more challenging to satisfy in face of increasingly sophisticated microarchitectural side-channel attacks. Such attacks allow adversaries to leak information from victim execution by observing changes in the microarchitectural state (e.g., cache eviction), typically via timing measurements (e.g., memory access latency).

Such attacks have been shown practical in the real world with or without the assistance of CPU bugs. Examples in the former category are transient execution attacks such as Spectre [38], Meltdown [40], L1TF [68], and MDS [15, 56, 70]. Examples in the latter category are traditional cache attacks (e.g., FLUSH+RELOAD [79] and PRIME+PROBE [51]) against security-sensitive software victims such as crypto libraries. While the former are the focus of many mitigation efforts by vendors, for the latter the burden of mitigation lies entirely on the shoulders of software developers [35].

In theory, this is feasible, as side-channel attacks leak secrets (e.g., crypto keys) by observing victim secret-dependent computations (e.g., branch taken or array indexed based on a crypto key bit) via microarchitectural measurements. Hence, eliminating explicit secret-dependent code/data accesses from software—a practice generally known as *constant-time programming* [13]—is a viable avenue for mitigation. In practice, removing side-channel vulnerabilities from software is a daunting and error-prone task even for skilled developers. Not surprisingly, even production side channel-resistant implementations are riddled with flaws [20, 61].

To address this problem, much prior work has proposed solutions to automatically transform programs into their constant-time equivalents or variations [27, 28, 34, 36, 41–43, 45, 49, 57, 63, 65, 71, 73, 75, 82–85]. Unfortunately, even the most recent solutions [52, 60, 78] offer limited security or compatibility guarantees, hindering their applicability to real-world programs.

In this paper, we introduce CONSTANTINE, a compiler-based system for the automatic elimination of side-channel vulnerabilities from programs. The key idea is to explore a radical design point based on *control and data flow linearization* (or CFL and DFL), where all the possible secret-dependent code/data memory accesses are always executed regardless of the particular secret value encountered. The advantage of this strategy is to provide strong security and compatibility guarantees by construction. The nontrivial challenge is to develop this strategy in a practical way, since a straightforward implementation would lead to *program state explosion*. For instance, naively linearizing secret-dependent branches that guard loop exits would lead to unbounded loop execution. Similarly, naively linearizing secret-dependent data accesses by touching all the possible memory locations would lead to an unscalable solution.

Our design is indeed inspired by radical and impractical-by-design obfuscation techniques such as the M/o/Vfuscator [3], which linearizes the control flow to collapse the program's control-flow graph into a single branchless code block with only data movement (i.e., x86 `mov`) instructions [37]. Each `mov` instruction uses an extra level of indirection to operate on real or dummy data depending on whether the code is running the intended side of a branch or not.

Revisiting such design point for side-channel protection faces several challenges. First, linearizing all the branches with `mov`-only code hinders efficient code generation in modern compilers and leads to enormous overheads. To address this challenge, Constantine only linearizes secret-dependent branches pinpointed by profiling information, allows arbitrary branchless instructions besides `mov`, and uses efficient indirect memory addressing to allow the compiler to generate efficient code. Second, the M/o/Vfuscator only linearizes the control flow and encodes branch decisions in new data flows, a strategy which would only multiply the number of secret-dependent data accesses. To address this challenge, Constantine couples CFL with DFL to also linearize all the secret-dependent data flows (generated by CFL or part of the original program).

Finally and above all, M/o/Vfuscator does not address state explosion. For example, it linearizes loop exits by means of invalid `mov` instructions, which generate exceptions and restart the program in dummy mode until the original loop code is reached. Other than being inefficient, this strategy originates new side channels (e.g., exception handling) that leak the number of loop iterations. To address state explosion, Constantine relies on carefully designed optimizations such as *just-in-time loop linearization* and *aggressive function cloning*. The former linearizes loops in the same way as regular branches, but adaptively bounds the number of iterations based on the original program behavior. The latter enables precise, context-sensitive points-to analysis which can strictly bound the number of possible targets at secret-dependent data accesses.

Collectively, our optimizations produce a scalable CFL and DFL strategy, while supporting all the common programming constructs in real-world software such as nested loops, indirect function calls, pointer-based accesses, etc. Our design not only addresses the state explosion problem, but also leads to a system that outperforms prior comprehensive solutions in terms of both performance and compatibility, while also providing stronger security guarantees. For example, we show Constantine yields overheads as low as 16% for cache-line attacks on standard benchmarks. Moreover, to show Constantine provides the first practical solution for automatic side-channel resistance for real-world software, we present a case study on the wolfSSL embedded TLS library. We show Constantine-protected wolfSSL can complete a modular multiplication of a ECDSA signature in 8 ms, which demonstrates Constantine's automated approach can effectively handle a fully-fledged real-word crypto library component for the very first time.

Contributions. To summarize, this paper proposes the following contributions:

- We introduce Constantine, a system for the protection of software from side channels.
- We show how Constantine can automatically analyze and transform a target program by efficiently applying control and data flow linearization techniques.
- We implement Constantine as a set of compiler transformations for the LLVM toolchain. Constantine is open source (available at https://github.com/pietroborrello/constantine).
- We evaluate Constantine on several standard benchmarks, evidencing its performance advantage against prior solutions. We also present a case study on the wolfSSL library to show its practical applicability on real-world software.

2 BACKGROUND

Microarchitectural side channels generally allow an adversary to infer *when* and *where* in memory a victim program performs specific code/data accesses. And by targeting secret-dependent accesses originating from secret-dependent control and data flows in the original program, an adversary can ultimately leak secret data. Constant-time programming is a promising solution to eliminate such explicit secret-dependent accesses from programs, but state-of-the-art automated solutions are severely limited in terms of security, performance, and/or compatibility.

Control Flow. Secret-dependent control flows (e.g., code branching on a crypto key bit) induce code accesses that microarchitectural attacks can observe to leak secrets. Early constant-time programming solutions only attempted to balance out secret-dependent branches with dummy instructions (e.g., with cross-copying [6]) to mitigate only simple execution time side-channel attacks [44]. Molnar et al. [49] made a leap forward with the *program counter security model* (PC-security), where the trace of secret-dependent executed instructions is the same for any secret value.

Prior work has explored two main avenues to PC-security. The first avenue is a form of *transactional execution* [52], which always executes both sides of every secret-dependent branch—hence a *real* and a *decoy* path—as-is, but uses a transaction-like mechanism to buffer and later discard changes to the program state from decoy paths. This approach provides limited security guarantees, as it introduces new side channels to observe decoy path execution and thus the secret. Indeed, one needs to at least mask exceptions from rogue operands of read/write instructions on decoy paths, introducing secret-dependent timing differences due to exception handling. Even when normalizing such differences, decoy paths may perform read/write accesses that real paths would not make, introducing new decoy data flows. An attacker can easily learn data-flow invariants on real paths (e.g., an array always accessed at the same offset range) and detect decoy path execution when the observed accesses reveal invariant violations. See Appendix A for concrete decoy path side channel examples. Also, this approach alone struggles with real-world software compatibility. For instance, it requires loops to be completely unrolled, which leads to code size explosion for nested loops and for those with large trip count.

Another avenue to PC-security is *predicated execution* [19], which similarly executes both real and decoy paths, but only allows the instructions from the real path to update the program state. Updates are controlled by a predicate that reflects the original program branch condition and take the form of a constant-time conditional assignment instruction (e.g., `cmov` on x86) [19]. When on a decoy path, read/write operations get rewired to a single (conditionally assigned) shadow address. However, such decoy (shadow) data flows can again introduce new side channels to leak the decoy nature of a path [18, 19]. Moreover, this form of predication hampers the optimization process of the compiler, forcing the use of pervasive cmov instructions and constraining code transformation and generation. Some more recent solutions attempt to generate more optimized code by allowing some [60] or all [78] accesses on unmodified addresses on decoy paths. However, this hybrid strategy mimics transactional execution behavior and is similarly vulnerable to side-channel attacks that detect data-flow invariant violations on decoy paths. In addition, existing solutions face the same compatibility issues of transactional solutions with real-world code.

Unlike prior solutions, CONSTANTINE's control-flow linearization (CFL) executes both real and decoy paths using an indirect memory addressing scheme to transparently target a shadow address along decoy paths. This strategy does not force the compiler to use cmov instructions and yields more efficient generated code. For instance, a CONSTANTINE-instrumented wolfSSL binary contains only 39% of cmov instructions (automatically emitted by the code generator, as appropriate) compared to predicated execution, resulting in a net CFL speedup of 32.9%. As shown later, the addition of data-flow linearization (DFL) allows CONSTANTINE to operate further optimizations, eliminating shadow address accesses altogether as well as the corresponding decoy data flows. CONSTANTINE is also compatible with all the common features in real-world programs, including variable-length loops bound by means of *just-in-time linearization* and indirect calls handled in tandem with DFL.

Data Flow. Data-dependent side channels have two leading causes on modern microarchitectures. Some originate from instructions that exhibit data operand-dependent latencies, e.g., integer division [19] and some floating-point instructions [10] on x86. Simple mitigations suffice here, including software emulation [11] (also adopted by CONSTANTINE), compensation code insertion [18], and leveraging hardware features to control latencies [19].

The other more fundamental cause stems from secret-dependent data flows (e.g., an array accessed at an offset based on a crypto key bit), which induce data accesses that microarchitectural attacks can observe to leak secrets. Hardware-based mitigations [74] do not readily apply to code running on commodity processors. To cope with such leaks, existing compiler-based solutions have explored code transformations [78] and software-based ORAM [52].

SC-Eliminator [78] transforms code to preload cache lines of security-sensitive lookup tables, so that subsequent lookup operations can result in always-hit cache accesses, and no secret-dependent time variance occurs. Unfortunately, since the preloading and the secret-dependent accesses are not atomic, a non-passive adversary may evict victim cache lines right after preloading and later observe the secret-dependent accesses with a cache attack. Cloak [32] adopts a similar mitigation approach, but enforces atomicity by means of Intel TSX transactions. Nonetheless, it requires manual code annotations and can only support short-lived computations. Moreover, these strategies are limited to standard cache attacks and do not consider other microarchitectural attacks, including those that operate at the subcacheline granularity [47, 80].

Raccoon [52] uses Path ORAM (Oblivious RAM) [63] as a shortcut to protect data flows from attacks. ORAMs let code conceal its data access patterns by reshuffling contents and accessing multiple cells at each retrieval [30]. Unfortunately, this strategy introduces substantial run-time overhead as each security-sensitive data access results in numerous ORAM-induced memory accesses.

Unlike prior solutions, CONSTANTINE's data-flow linearization (DFL) eliminates all the explicit secret-dependent data flows (generated by CFL or part of the original program) by forcing the corresponding read/write operations to touch all their target memory locations as computed by static points-to analysis. While such analyses are known to largely overapproximate target sets on real-world programs, CONSTANTINE relies on an *aggressive function cloning strategy* to enable precise, context-sensitive points-to analysis and strictly bound the number of possible targets. For instance, a CONSTANTINE-instrumented wolfSSL binary using state-of-the-art points-to analysis [66] yields an average number of target objects at secret-dependent data accesses of 6.29 and 1.08 before and after aggressive cloning (respectively), a net reduction of 83% resulting in precise and efficient DFL. Unlike prior solutions that are limited to array accesses, CONSTANTINE is also compatible with arbitrary pointer usage in real-world programs.

3 THREAT MODEL

We assume a strong adversary able to run arbitrary code on the target machine alongside the victim program, including on the same physical or logical core. The adversary has access to the source/binary of the program and seeks to leak secret-dependent computations via microarchitectural side channels. Other classes of side channels (e.g., power [39]), software (e.g., external libraries/OS [14]) or hardware (e.g., transient execution [16]) victims, and vulnerabilities (e.g., memory errors or other undefined behaviors [72]) are beyond the scope of constant-time programming and subject of orthogonal mitigations. We further make no restrictions on the microarchitectural side-channel attacks attempted by the adversary, ranging from classic cache attacks [51, 79] to recent contention-based attacks [8, 31]. With such attacks, we assume the adversary can observe the timing of arbitrary victim code/data accesses and their location at the cache line or even lower granularity.

4 CONSTANTINE

This section details the design and implementation of CONSTANTINE. We first outline its high-level workflow and building blocks.

4.1 Overview

CONSTANTINE is a compiler-based system to automatically harden programs against microarchitectural side channels. Our linearization design pushes constant-time programming to the extreme and embodies it in two flavors:

(1) **Control Flow Linearization** (CFL): we transform program regions influenced by secret data to yield secret-invariant instruction traces, with real and decoy parts controlled by a *dummy execution* abstraction opaque to the attacker;
(2) **Data Flow Linearization** (DFL): we transform every secret-dependent data access (including those performed by dummy execution) into an oblivious operation that touches all the locations such program point can possibly reference, leaving the attacker unable to guess the intended target.

CFL and DFL add a level of *indirection* around value computations. The CFL dummy execution abstraction uses it to implicitly nullify the effects of instructions that presently execute as decoy paths. DFL instead wraps load and store operations to induce memory accesses for the program that are secret-invariant, also ensuring real and decoy paths access the same collections of objects.

Linearizing control and data flows represents a radical design point with obvious scalability challenges. To address them, Constantine relies on carefully designed optimizations. For control flows, we rely on a M/o/Vfuscator-inspired *indirect memory addressing* scheme to legalize decoy paths while allowing the optimizer to see through our construction and generate efficient code. We also propose *just-in-time loop linearization* to efficiently support arbitrary loops in real-world programs and automatically bound their execution based on the behavior of the original program (i.e., automatically padding the number of iterations based on the maximum value observed on real paths).

For data flows, we devise *aggressive function cloning* to substantially boost the precision of static memory access analysis and minimize the extra accesses required by DFL. To further optimize DFL, we rely on an efficient object metadata management scheme and on hardware-optimized code sequences (e.g., AVX-512) to efficiently touch all the necessary memory locations at each secret-dependent data access. We also exploit synergies between control-flow and data-flow handling to (i) eliminate the need for shadow accesses on decoy paths (boosting performance and eradicating problematic decoy data flows altogether); (ii) handle challenging indirect control flows such as indirect function calls in real-world programs.

To automatically identify secret-dependent code and data accesses, we rely on *profiling* information obtained via dynamic information flow tracking and propagate the dependencies along the call graph. To analyze memory accesses, we consider a state-of-the-art Andersen-style points-to analysis implementation [66] and show how aggressive function cloning can greatly boost its precision thanks to newly added full context sensitivity.

From a security perspective, CFL ensures PC-security for all the instructions that operate on secret data or whose execution depends on it; in the process it also replaces variable-latency instructions with safe software implementations. DFL provides analogous guarantees for data: at each secret-dependent load or store operation, the transformed program obliviously accesses every potentially referenced location in the execution for that program point and is no longer susceptible to microarchitectural leaks by design.

Figure 1 provides a high-level view of the CFL, DFL, and support program analysis components behind Constantine. Our techniques are general and we implement them as analyses and transformation passes for the *intermediate representation* (IR) of LLVM.

4.2 Control Flow Linearization

With control flow linearization (CFL) we turn secret-dependent control flows into straight-line regions that meet PC-security requirements by construction [49], proposing just-in-time linearization for looping sequences. We also make provisions for instructions that may throw an exception because of rogue values along decoy paths, or yield variable latencies because of operand values.

***CFL**: The sequence of secret-dependent instructions that the CPU executes is constant for any initial input (PC-security) and data values do not affect the latency of each such instruction.*

With this invariant, only data access patterns can then influence execution time, and DFL will make them insensitive to secret input values. We assume that an oracle (the taint analysis of §4.4.1) enucleates which control-flow transfer decisions depend on secret data. Such information comprises if-else and loop constructs and indirect-call targets. For each involved code region, we push the linearization process in a recursive fashion to any nested control flows (i.e., if-else branches, loops, and function calls), visiting control-flow graphs (CFGs) and call graph edges in a post-order depth-first fashion. By doing so we avoid leaks from decoy paths when executing secret-independent inner branches in a protected region.

4.2.1 Dummy Execution. Each linearized region holds a *"taken"* predicate instance that determines if the original program would execute it (*real path*) or not (*decoy path*) under the current program state. We incrementally update the predicate with a new instance at every control-flow decision that guards the region in the original program, and let the compiler use the previous incoming instance upon leaving the region. The predicate backs a *dummy execution* indirection abstraction where we let decoy paths execute together with real paths, and use the *taken* predicate to prevent that visible effects from decoy paths may pollute the program state.

The key to correctness is that we can safely allow decoy paths to make local computations (i.e., assign to virtual registers in the IR), as long as their values do not flow into memory. For memory operations, each pointer expression computation selects an artificial $\perp$ value when in dummy execution. DFL primitives wrap every load and store instruction and make both real and decoy paths stride the same objects thanks to points-to metadata associated with the memory operation. Upon leaving a region, local values that the program may use later (i.e. *live* virtual registers) undergo a selection step to pick values from real paths at merge points.

The key to efficiency is using a selection primitive that is transparent for the optimizer thanks to indirection. As we observed in §2, the `cmov` selector typical of predicated execution constrains the behavior of the optimizer during code generation. We leverage the indirection on *taken* to design selection primitives based on arithmetic and logic operations that can instead favor optimizations.

Let us consider the pointer assignment $ptr = taken\ ?\ p : \perp$ of Figure 1. By modeling *taken* as an integer being 1 on real paths and 0 on decoy ones, and by using NULL to represent $\perp$ for DFL, the selection becomes $ptr = taken * p$. DFL helpers will prevent NULL accesses and deem them as from decoy paths: those cannot happen on real paths since, like prior literature [52], we work on error-free programs. This constant-time multiplication-based scheme unleashes many arithmetic optimizations (e.g., global value numbering [55], peephole [7]) at the IR and backend level, bringing a net CFL speedup of 32.9% in wolfSSL over using the `cmov` approach. Appendix B details other primitives that we evaluated.

Selection may be needed for (ϕ) compiler temporaries too, as we will detail in §4.2.3. Unlike memory addresses, both incoming values may be arbitrary, allowing for more limited optimization: for them we use the `select` IR instruction and let LLVM lower it branchlessly as it sees fit (including an x86 `cmov`).

Hereafter, we use `ct_select` to refer to a constant-time selection of any values, but we inline the logic in the IR in the implementation.

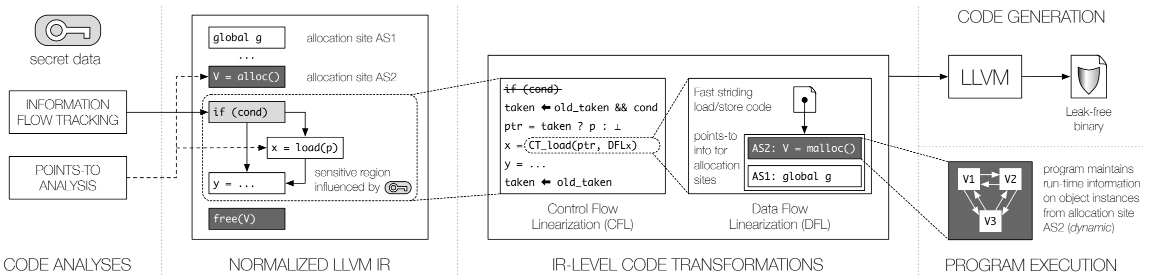
secret data
INFORMATION FLOW TRACKING
POINTS-TO ANALYSIS
CODE ANALYSES
global g
allocation site AS1
V = alloc()
allocation site AS2
if (cond)
x = load(p)
y = ...
sensitive region influenced by
free(V)
NORMALIZED LLVM IR
~~if (cond)~~
taken ⬅ old_taken && cond
ptr = taken ? p : ⊥
x = CT_load(ptr, DFLx)
y = ...
taken ⬅ old_taken
Control Flow Linearization (CFL)
Fast striding load/store code
points-to info for allocation sites
AS2: V = malloc()
AS1: global g
Data Flow Linearization (DFL)
IR-LEVEL CODE TRANSFORMATIONS
CODE GENERATION
LLVM
Leak-free binary
V1
V2
V3
program maintains run-time information on object instances from allocation site AS2 (dynamic)
PROGRAM EXECUTION

Figure 1: Architecture of Constantine: code analyses, CFL & DFL transformations, and run-time object metadata.

4.2.2 Compiler IR Normalization. Constantine takes as input the intermediate representation (IR) produced for the program by the language-specific compiler frontend. We assume that the IR comes in static single assignment (SSA) form [55] and that the CFG of every function containing regions to transform is reducible. The code can come in already-optimized form (e.g., `-O2`, `-O3` settings).

We apply a number of *normalization* passes that simplify later transformations with the ultimate goal of having *single-entry, single-exit regions* as unit of transformation, similarly to [78].

We use existing LLVM passes to lower switch constructs into if-else sequences, and to unify multiple function exit points into a single one (for abort-like sequences that do not fall through, we add artificial CFG edges to the exit node). As we work on error-free programs, we replace exception-aware `invoke` statements with normal calls. We also turn indirect calls into if-else sequences of direct calls using points-to information (§4.4.2), guarding each direct call with a pointer comparison on the target.

We then massage the CFG using standard compiler techniques [7] so that it results into a graph composed only of single-entry, single-exit regions: this will hold for all branches and loop constructs in the IR. This normalized IR is the input for the taint oracle of §4.4.1.

4.2.3 Branch Linearization. We can now detail how branch linearization operates and its orchestration with dummy execution. Under the single-entry, single-exit structural assumption from IR normalization, for a conditional construct of the likes *if (cond) then {A} else {B}*, we note that its exit CFG node post-dominates both the "then" and "else" regions of the branch, and is dominated by the entry node by construction. In SSA form, ϕ-nodes select incoming path-sensitive values. To linearize a conditional construct we:

(1) remove the conditional branch, unlinking blocks A and B;
(2) replace in A every pointer expression computation with a conditional assignment `ct_select(`*cond*, *ptr*, ⊥`)`;
(3) replace similarly in B, using the condition negated (!*cond*);
(4) wrap memory accesses with DFL `ct_{load, store}` primitives, supplying the DFL metadata for the operation (§4.3);
(5) replace each ϕ-node $v_0 = \phi(v_A, v_B)$ in the exit block (which assigns virtual register v_0 according to whether A or B executed) with a conditional assignment `ct_select(`*cond*, v_A, v_B`)`;
(6) merge ⟨entry, A, B, exit⟩ to form a single block, in this order.

We thus "sink" *cond* to conditionally assign pointers (⊥ for decoy paths) and virtual registers that outlive the region. Our transformation preserves the SSA form and can always be applied locally.

We can now add the dummy execution idea to the picture. Without loss of generality, let us consider two nested if-else statements that possibly take part in a larger linearized region as in Figure 2.

```
if (c_outer) {
  b_1 = v[2]
} else {
  if (c_inner) {
    b_2 = v[0]
  } else {
    b_3 = 0
  }
  b_inner = φ(b_2,b_3)
}
b_4 = φ(b_1,b_inner)
v[1] = b_4
```

(a) Original code

```
t_0 = <incoming 'taken' predicate>
t_1 = c_outer && t_0
  ptr_1 = ct_select(t_1, &v[2], ⊥)
  b_1 = ct_load(ptr_1, DFL_b1)
t_1-else = !c_outer && t_0
  t_2 = c_inner && t_1-else
    ptr_2 = ct_select(t_2, &v[0], ⊥)
    b_2 = ct_load(ptr_2, DFL_b2)
  t_2-else = !c_inner && t_1-else // unused
    b_3 = 0
  b_inner = ct_select(c_inner, b_2, b_3)
b_4 = ct_select(c_outer, b_1, b_inner)
ptr_3 = ct_select(t_0, &v[1], ⊥)
ct_store(ptr_3, b_4, DFL_store1)
```

(b) After linearization

Figure 2: Linearization and dummy execution.

When reaching the outer if construct, the program sees a *taken* predicate instance t_0 that determines whether the execution reached the construct as part of a real (*taken* = *true*) or decoy computation.

Inside a region, IR instructions that assign virtual registers do not need to know t_0. Path-sensitive assignments of live-out values from a region, such as b_{inner}, check the linearized conditions (c_{inner} in this case). Memory-related instructions see instead their pointer expressions conditionally assigned according to some t_i *taken* instance. Those instances are updated upon entering the enclosing code block in the (original) program to reflect the combination of control-flow conditions with the incoming *taken* predicate.

4.2.4 Loop Linearization. To cope with the practical requirements of real-world code, with Constantine we explore a *just-in-time* approach for the linearization of loops. Let us consider the following secret-sensitive fragment, taken from a wolfSSL function that computes $x/R == x \ (mod\ N)$ using a Montgomery reduction:

```
  _c   = c + pa;
  tmpm = a->dp;
  for (x = 0; x < pa+1; x++)
    *tmpm++ = *_c++;
  for (; x < oldused; x++)  // zero any excess digits on
    *tmpm++ = 0;  // destination that we didn't write to
```

The induction variable x depends on secret data pa, outlives the first loop, and dictates the trip count of the second loop. Prior solutions struggle with each of these aspects, as well as with continue/break statements we found in wolfSSL. For the secret-dependent trip count issue, some [78] try to infer a bound and pad the loop with decoy iterations, then unroll the loop completely. However, high trip counts seen at run time or inaccurate bound predictions make unrolling immediately impractical due to code bloat.

In Constantine we design a new approach to handle loops that avoids unrolling and supports full expressivity for the construct.

```
base:
  i_base = 0
body:
  i_cur = φ(base: i_base,
            body: i_body)
  [...]
  i_body= i_cur+1
  [...]
  cond = ... // exit loop?
  br cond, out, body
out:
  x = i_body
```

(a) Original code

```
base:
  i_base = 0
body:
  i_cur = φ(base: i_base, body: i_body)
  i_real = φ(base: undef, body: i_out)
  i_body = i_cur+1
  i_out= ct_select(taken, i_body, i_real)
  [...]
  cond = ... // exit loop?
  cfl_cond = ... // CFL override
  br cfl_cond, out, body
out:
  x = i_out
```

(b) After linearization

Figure 3: Linearization with local variables outliving loops.

The key idea is to flank the normal trip count of a loop with an own CFL induction variable—dubbed *c_idx* next—and let such variable dictate just-in-time how many times that loop should execute.

After IR normalization, a loop is a single-entry, single-exit region: its exit block checks some condition *cond* for whether the program should leave the loop or take the back-edge to the loop body. Note that break/continue statements are just branches to the exit node and we linearize them as in §4.2.3. Before entering the loop we set *c_idx* := 0, and modify the exit block in such a way that the program still makes the original *cond* computation, but uses instead the current *c_idx* value to decide whether to leave the loop.

Say that we expect the program to execute the loop no more than k times (we address loop profiling in §4.4.1). At every iteration our exiting decision procedure increments *c_idx* by 1 and faces:

(1) *taken = true* $\land$ *cond = false*. The program is on a real path and wishes to take the back-edge to the body: we allow it;
(2) *taken = true* $\land$ *cond = true*. The program is on a real path and wishes to exit the loop: if *c_idx* = k we allow it, otherwise we enter dummy mode (*taken* := *false*) and the program will perform next k − *c_idx* dummy iterations for PC-security;
(3) *taken = false*. We make the program leave the loop when *c_idx* = k, and take the back-edge otherwise.

Note that for (3) we do not use the value of *cond*, as it can go rogue along decoy paths, but we still read it for the sake of linearization. Additionally, during (1) we validate the prediction of the oracle: whenever real program paths wish to iterate more than k times, we adaptively update k allowing the loop to continue, and use the k' seen on loop exit as the new bound when the program reaches the loop again. The handling of this comparison is also linearized.

Nested loops or linearized branches in loop bodies pose no challenge: we incrementally update the taken predicate and restore it across regions as we did for nested branches in §4.2.3 and Figure 2.

Let us resume the discussion of the code fragment. As variable x outlives the first loop, we should prevent decoy paths from updating it for the sake of correctness. If the compiler places x in memory, the IR will manipulate it using load and store instructions, and the dummy execution abstraction guarantees that only real paths can modify it. If instead it uses a virtual register v for performance, we flank it with another register v' conditionally assigned according to *taken*, and replace all the uses of v as operand in the remainder of the CFG with v'. Figure 3 shows this transformation with $\mathtt{i}_{\mathtt{body}}$ and $\mathtt{i}_{\mathtt{out}}$: decoy paths keep modifying i_{body} for the sake of PC-security, but do not pollute the program state. Thanks to this design, we do not demote v to memory storage, which could harm performance especially for tight loops, nor we constrain the optimizer.

4.2.5 Operand Sanitization. As last step, we safeguard computations that could cause termination leaks from rogue values along decoy paths. In our design, this may happen only with divisions instructions receiving zero as divisor value. In §2 we noted that x86 integer division is also subject to variable latencies from operand values. We address both issues via software emulation, replacing `*div` and `*rem` LLVM instructions with subroutines that execute in constant-time, and for `*div` are also insensitive to rogue values.

4.2.6 Code Generation. Our CFL design poses no restrictions on code optimization as well as code generation operated in the backend. The optimizer can transform CFL-generated indirect memory references by means of optimizations such as common subexpression elimination and the code generator can lower such references using the most efficient patterns for the target architecture (including `cmov` instructions on occasion). However, we need to prevent the code generation process from inadvertently adding branches in branchless IR-level code. Indeed, this is not uncommon [20, 49]: luckily, modern compilers offer explicit support to preserve our constant-time invariants. In more detail, we use LLVM backend options (e.g., `-x86-cmov-converter=0` for branchless lowering on x86) to control this behavior. As discussed later, we have also experimentally validated Constantine-instrumented binaries preserve our security invariants by means of a dedicated verifier.

4.3 Data Flow Linearization

With data flow linearization (DFL), we devise a new abstraction for controlling the data access patterns influenced by secret data, so that arbitrarily different (secret) inputs will lead to the same observable program behavior for an attacker. As we discuss in our security evaluation of §5, this design hardens against side-channel attacks that prior solutions cannot handle and it does not suffer from leaks through data-flow invariants and memory safety violations as we saw for such solutions in §2. Furthermore, thanks to its combination with points-to analysis, DFL is the first solution that does not place restrictions on pointer and object types, supporting for instance pointer-to-pointer casts that occur in real-world crypto code.

***DFL**: For every program point that performs a memory load or store operation, DFL obliviously accesses all the locations that the original program can possibly reference for any initial input.*

To support this invariant we conduct a context-sensitive, field-sensitive points-to analysis (described in §4.4.2) to build DFL metadata for each use of a pointer expression in a sensitive load or store instruction. Such metadata describes the portions of the object(s) that the expression may reference each time the program evaluates it. We assume that only program-allocated memory can hold secret-dependent data (external library calls cannot leak from §3).

For dynamic storage, that is stack- and heap-allocated objects, we instrument the involved allocation sites in the program to keep track at run time of the object instances currently stemming from an allocation site of interest to DFL (rightmost part of Figure 1).

DFL uses indirection around incoming pointer values: it obliviously accesses all the candidate object portions identified by the

```
typedef struct dfl_obj_list {
    struct dfl_obj_list* next;
    struct dfl_obj_list* prev;
    struct dfl_obj_list** head_ptr;  // for fast removal from list
    unsigned long magic;   // to distinguish DFL heap objects
    unsigned char data[];  // contents of program object
} dfl_obj_list_t;
```

Figure 4: In-band metadata for data flow linearization.

points-to analysis, and retrieves or modifies the memory value only within the object instance (if any) corresponding to the incoming pointer value. We apply DFL to every memory load or store made in a code region linearized by CFL (where the operation will see an incoming ⊥ value when on a decoy path), and to memory operations that are outside input-dependent control flows but still secret-sensitive (e.g., array accesses with input-dependent index).

Unlike prior solutions, we do not need a shadow location for decoy paths (accessing it would leak the nature of such paths, §2), nor we let rogue pointers concur to memory accesses. Our design makes data accesses oblivious to secret dependencies and to the nature of control paths, and preserves memory safety in the process.

4.3.1 Load and Store Wrappers. For the linearization of the data flow of accessed locations, we use `ct_load` and `ct_store` primitives for DFL indirection and resort to different implementations optimized for the storage type and the size of the object instances to stride obliviously. As we discussed when presenting the CFL stage, we accompany each use of a pointer expression in a load or store with DFL metadata specific to the program point.

DFL metadata capture at compilation time the *points-to information* for all the allocation sites of possibly referenced objects. The analysis comprises stack allocations, objects in global memory, and heap allocation operations. For each site, we use field-accurate information to limit striding only to portions of an object, which as a whole may hold thousands of bytes in real-world code.

Depending on the scenario, the user can choose the granularity λ at which memory accesses should become oblivious to an adversary. One may only worry about cache attacks ($\lambda = 64$) if, say, only cross-core (cache) attacks are to be mitigated (e.g., with cloud vendors preventing core co-location across security domains by construction [5]). Or one may worry about arbitrary attacks if, say, core colocation across security domains is possible and attack vectors like MemJam ($\lambda = 4$) are at reach of the attacker.

Our wrapper implementations stride an object portion with a pointer expression incremented by λ bytes every time and which may match the incoming p input pointer from the program at most once. Depending on the object portion size, DFL picks between standard AVX instructions for striding, AVX2/AVX512 gather-scatter sequences to load many cache lines at once followed by custom selection masks, and a `cmov`-based sequence that we devise to avoid the AVX setup latency for small objects (details in Appendix C).

The DFL load and store wrappers inspect all the allocation sites from the metadata. For global storage only a single object instance exists; for stack and heap objects the instances may change during the execution, and the wrappers inspect the run-time metadata that the transformed program maintains (using doubly linked lists and optimizations that we describe in the next sections).

For a load operation, DFL strides all the object instances that the program point may reference and conditionally selects the value from the object portion matching the desired address. For decoy paths, no match is found and `ct_load` returns a default value.

For a store operation, DFL breaks it into a load followed by a store. The rationale is to write to every plausible program point's target, or the adversary may discover a secret-dependent write destination. For every object portion identified by DFL store metadata, we read the current value and replace it with the contents for the store only when the location matches its target, otherwise we write the current value back to memory. Decoy paths "refresh" the contents of each object; real paths do the same for all but the one they modify.

4.3.2 Object Lifetime. DFL metadata supplied at memory operations identify objects based on their allocation site and characteristics. While global storage is visible for the entire execution, stack and heap locations have a variable lifetime, and we need to maintain run-time metadata for their allocation sites.

We observe that real-world crypto code frequently allocates large structures on the stack and pointers seen at memory operations may reference more than one such structure. At the LLVM IR level, stack-allocated variables take the form of `alloca` instructions that return a pointer to one or more elements of the desired type. The compiler automatically releases such storage on function return.

We interpose on `alloca` to wrap the object with *in-band metadata* information depicted in Figure 4. Essentially, we prepend the originally allocated element with fields that optimize DFL operations and preserve stack alignment: the program element becomes the last field of a variable-sized `dfl_obj_list_t` structure. Then, we assign the virtual register meant to contain the v pointer from `alloca` with the address of $v.data$ (32-byte offset on x64).

This transformation is simple when operating at the compiler IR level: unlike binary rewriting scenarios [24], the compiler is free to modify the stack layout while preserving program semantics, including well-behaved pointer arithmetics. Upon `alloca` interposition, we make the program update the run-time allocation site information and a symmetric operation happens on function exit.

Heap variables see a similar treatment. We interpose on allocation operations to widen and prepend the desired object with in-band metadata, with the address of $v.data$ returned to the program instead of the allocation base v. The $v.magic$ field is pivotal for handling `free()` operations efficiently: when interposing on them, we may witness a `dfl_obj_list_t` structure or a "standard" object from other program parts. We needed an efficient means to distinguish the two cases, as `free()` operations take the allocation base as input: for DFL objects we have to subtract 32 from the input pointer argument. We leverage the fact that allocators like the standard libc allocator `ptmalloc` prepend objects with at least one pointer-sized field. Hence accessing a heap pointer p as $p - 8$ is valid: for DFL objects it would be the address of the *magic* field and we check its peculiar value to identify them.

4.3.3 Optimizations. One advantage of performing DFL at compiler IR level is that we can further optimize both the data layout to ease metadata retrieval and the insertion of our DFL wrappers.

We identify functions that do not take part in recursive patterns and promote to global variables their stack allocations that sensitive accesses may reference. The promotion is sound as such a function can see only one active stack frame instance at a time. The promotion saves DFL the overhead of run-time bookkeeping, with faster metadata retrieval for memory operations as we discuss next. To identify functions apt for promotion, we analyze the call graph of the program (made only of direct calls after IR normalization) to identify strongly connected components from recursion patterns [81] and exclude functions taking part in them.

We also partially inline DFL handlers, as object allocation sites are statically known from points-to analysis. For global storage, we also hard-code the involved address and striding information. For instance, a load operation from address *ptr* becomes in pseudo-code:

```
res = 0
res |= dfl_glob_load(ptr, glob1, stride_offset_g1, stride_size_g1)
res |= dfl_glob_load(ptr, glob2, stride_offset_g2, stride_size_g2)
res |= dfl_load(ptr, objs_as1, stride_offset_as1, stride_size_as2)
res |= dfl_load(ptr, objs_as2, stride_offset_as2, stride_size_as2)
res |= dfl_load(ptr, objs_as3, stride_offset_as2, stride_size_as2).
```

This is because the oracle determined that *ptr* may reference (portions of) global storage *glob1*, *glob2* or objects from allocation sites *as1*, *as2*, *as3*, where $objs\_as_i$ is the pointer to the data structure (a doubly linked list of objects, as with AS2 in Figure 1) maintained at run time for the allocation site (§4.3.2). With the OR operations we perform value selection, as each `dfl_` helper returns 0 unless the intended location *ptr* is met during striding. In other words, instead of maintaining points-to sets for memory operations as data, we inline their contents for performance (saving on retrieval time) and leave the LLVM optimizer free to perform further inlining of `dfl_` helpers code. The treatment of store operations is analogous.

Finally, we devise an effective (§7) striding optimization for loops. We encountered several loops where the induction variable flows in a pointer expression used to access an object, and from an analysis of its value (based on LLVM's *scalar evolution*) we could determine an invariant: the loop would be touching all the portions that require DFL striding and a distinct portion at each iteration. In other words, the code is "naturally" striding the object: we can avoid adding DFL striding and thus save on $n(n-1)$ unnecessary accesses.

4.4 Support Analyses

The compatibility of Constantine with real-world code stems also from two "oracles" as we tailor robust implementations of mainstream program analysis techniques to our context: an *information flow tracking* technique to identify program portions affected by a secret and a *points-to analysis* that we enhance with context sensitivity to obtain points-to sets as accurate as possible.

4.4.1 Identifying Sensitive Program Portions. Control and data flow linearization need to be applied only to regions affected by secret data, as protecting non-leaky code only hampers performance.

We assume the user has at their disposal a profiling suite to exercise the alternative control and data flow paths of the crypto functionality they seek to protect. Developers can resort to existing test suites for libraries, actual workloads, or program testing tools (e.g. generic [26] or specialized [33] fuzzers) to build one.

We then use DataFlowSanitizer (DFSan), a dynamic information flow tracking solution for LLVM, to profile the normalized IR of §4.2.2 over the profiling suite. DFSan comes with taint propagation rules for virtual registers and program memory and with APIs to define taint source and sink points. We write taint source configurations to automatically taint data that a program reads via I/O functions (e.g., a key file) and use as sink points conditionals, memory load/store operations, and `div`/`rem` instructions in the normalized IR. In the DFSan-transformed IR we then encode rules in the spirit of FlowTracker [54] to handle implicit flows among virtual registers, leaving those possibly taking place through memory to complementary tools like FTI [29].

We aggregate DFSan outputs to build a set of branches and memory accesses that are secret-dependent, feeding it to CFL and DFL. As we mentioned in §4.2, CFL will then push the hardening process to nested flows, linearizing their control and data flows. During the execution of the profiling suite we also profile loop trip counts that we later use as initial predictions for CFL (§4.2.4).

4.4.2 Points-to Analysis. Points-to analyses [58] determine the potential targets of pointers in a program. Nowadays they are available off-the-shelf in many compilation systems, with inclusion-based approaches in Andersen style [9] typically giving the most accurate results. In Constantine, we extend the Andersen-style analysis of the popular SVF library for LLVM [66]. For each pointer usage in the program, we use this analysis to build the *points-to set* of objects that it may reference at run time. Typically, points-to analyses collapse object instances from a dynamic allocation site into an abstract single object. Hence, points-to sets contain information on object allocation sites and static storage locations.

Points-to analyses are sound. However, they may overapproximate sets by including objects that the program would never access at run time. In a lively area of research, many solutions feature inclusion-based analyses as the approach is more accurate than the alternative, faster unification-based one [62]. Inclusion-based analyses could give even more accurate results if they were to scale to *context sensitivity*, i.e., they do not distinguish the uses of pointer expressions (and thus potentially involved objects) from different execution contexts. The context is typically intended as call-site sensitivity, while for object-oriented managed languages other definitions exist [46, 59]. To optimize the performance of DFL, we need as accurate points-to sets as possible, so in Constantine we try to restore context sensitivity in an effective manner for a sufficiently large codebase such as the one of a real-world crypto library.

Aggressive Cloning. We use function cloning to turn a context-insensitive analysis in a context-sensitive one. A calling context [22] can be modeled as an acyclic path on the call graph and one can create a function clone for every distinct calling context encountered during a graph walk. This approach can immediately spin out of control, as the number of acyclic paths is often intractable [23, 77].

Our scenario however is special, as we may clone only the functions identified as secret-dependent by the other oracle, along with their callees, recursively. We thus explore *aggressive cloning* along the maximal subtrees of the call graph having a sensitive function as root. The rationale is that we need maximum precision along the program regions that are secret-dependent, while we can settle for context-insensitive results for the remainder of the program, which normally dominates the codebase size.

Aggressive cloning turns out to be a key performance enabler, making DFL practical and saving on important overheads. As we discuss in §7, for wolfSSL we obtain points-to sets that are ~6x smaller than the default ones of SVF and very close to the run-time optimum. The price that we trade for such performance is an increase in code size: this choice is common in much compiler research, both in static [50] and dynamic [17] compilation settings, for lucrative optimizations such as value and type-based specialization.

Refined Field Sensitivity. A field-sensitive analysis can distinguish which portions of an object a pointer may reference. Real-world crypto code uses many-field, nested data structures of hundreds or thousands of bytes, and a load/store operation in the program typically references only a limited portion from them. Field-accurate information can make DFL striding cheaper: this factor motivated our practical enhancements to the field-sensitive part of SVF.

The reference implementation fails to recover field-precise information for about nine-tenths of the wolfSSL accesses that undergo DFL, especially when pointer arithmetics and optimizations are involved. We delay the moment when SVF falls back to field-insensitive abstract objects and try to reverse-engineer the structure of the addressing so to fit it into static type declarations of portions of the whole object. Our techniques are inspired by *duck typing* from compiler research; we cover them in Appendix D. Thanks to these refinements, we could recover field-sensitive information for pointers for 90% of the sensitive accesses in our case study.

Indirect Calls. Points-to analysis also reveals possible targets for indirect calls [66]. We use this information during IR normalization when promoting them to if-series of guarded direct calls (§4.2.2), so to remove leaks from variable targets. We refine the candidates found by SVF at call sites by matching function prototype information and eliminating unfeasible targets. Indirect call target information is also necessary for the aggressive cloning strategy.

4.5 Discussion

CONSTANTINE implements a compiler-based solution for eliminating microarchitectural side channels while coping with the needs of real-word code. We chose LLVM for its popularity and the availability of mature information-flow and points-to analyses. Nonetheless, our transformations are general and could be applied to other compilation toolchains. Similarly, we focus on x86/x64 architectures, but multiplexing conditional-assignment and SIMD striding instructions exist for others as well (e.g. ARM SVE [64] , RISC-V "V" [4]).

Moreover, operating at the compiler IR level allows us to efficiently add a level of indirection, with *taken* unleashing the optimizer and with DFL making memory accesses oblivious to incoming pointers. In addition, aggressive function cloning allows us to transform the codebase and unveil a significantly more accurate number of objects to stride. The IR also retains type information that we can leverage to support field sensitivity and refine striding.

The just-in-time strategy to linearize secret-dependent unbounded control flows (loops) allows us to dodge intractability with high bounds and code bloat with tractable instances [60]. For points-to set identification and indirect call promotion, our analyses yield very accurate results (i.e., closely matching the run-time accesses) on the programs we consider. We leave the exploration of a just-in-time flavor for them to future work, which may be helpful in non-cryptographic applications.

The main shortcoming of operating at the IR level is the inability to handle inline assembly sequences found in some crypto libraries. While snippets that break constant-time invariants are uncommon, they still need special handling, for instance with annotations or lifting. Verification-oriented lifting [53], in particular, seems a promising avenue as it can provide formally verified C equivalent representations that we could use during IR normalization.

As the programs we study do not exercise them, for space limitations we omit the treatment of recursion and multithreading. Appendix E details the required implementation extensions.

5 SECURITY ANALYSIS

This section presents a security analysis of our transformations. We start by arguing that instrumented programs are semantically correct and induce secret-oblivious code and data access traces. We then discuss how our design emerges unscathed by traditional passive and active attacks and examine the residual attack surface.

Correctness and Obliviousness. Correctness follows directly from our design, as all our transformations are semantics-preserving. In short, for control flows, real paths perform all and only the computations the original program would make. For data flows, values from decoy paths cannot flow into real paths and correctness properties such as memory safety are preserved. Appendix F provides informal proofs for these claims.

We now discuss how our linearization design yields *oblivious* code and data access traces. For code accesses, PC-security follows by CFL construction, as we removed conditional branches, loops see a fixed number of iterations (we discuss wrong trip count predictions later), and IR normalization handles abort-like sequences. For data accesses, we wrapped load and store operations with DFL machinery that strides portions of every abstract (i.e., by allocation site) object that the operation may access, *independently of the incoming pointer value.* For dynamic storage, for any two secrets, the program will see identical object collections to maintain at runtime: the composition of the lists can vary during the execution, but identically so for both secrets. Finally, for virtual registers that are spilled to memory by the backend, the CPU reads and writes them with the same instructions regardless of the current *taken* predicate value, so those accesses are also oblivious.

Security Properties. We build on the obliviousness claims above to show that both passive attacks (attackers only monitoring microarchitectural events) and active attacks (attackers also arbitrarily tampering with the microarchitectural state) are unsuccessful.

No instruction latency variance from secret-dependent operand values is possible, since we replace and sanitize instructions such as division (§4.2.5). Memory accesses may have variable latencies, but, thanks to the DFL indirection, those will only depend on non-secret code/data and external factors. Moreover, DFL wrappers do not leak secrets and do not introduce decoy paths side channels in terms of decoy data flows or exceptions. In §4.3.1, we explained how load and store helpers stride objects using safe [19, 52] `cmov` or SIMD instructions. As for decoy paths, *taken* can conditionally assign an incoming pointer with $\perp$: the adversary would need access to CPU registers or memory contents to leak the nature of a path (outside the threat model). Finally, helpers are memory-safe as points-to analysis is sound and we track object lifetimes.

Finally, an active attacker may perturb the execution to attempt Flush+Reload, Prime+Probe, and other microarchitectural attacks to observe cache line-sized or even word-sized victim accesses. With vulnerable code, they could alter for instance the access timing for a specific portion of memory, and observe timing differences to detect matching victim accesses. However, thanks to the obliviousness property of our approach, leaking victim accesses will have no value for the attacker, because we access all the possible secret-dependent code/data locations every time.

Residual Attack Surface. We now discuss the residual attack surface for Constantine. Design considerations aside, the correctness and obliviousness of the final instrumented binary are also subject to the correctness of our Constantine implementation. Any implementation bug may introduce an attack surface. To mitigate this concern, we have validated our correctness claims experimentally by running extensive benchmarks and test suites for the programs we considered in our evaluation. We have also validated our obliviousness claims experimentally by means of a verifier, as detailed later. Overall, our implementation has a relatively small trusted computing base (TCB) of around 11 KLOC (631 LOC for our profiler, 955 LOC for CFL, 2561 for DFL, and 7259 LOC for normalization and optimization passes), which provides confidence it is possible to attain correctness and obliviousness in practice.

Constantine's residual attack surface is also subject to the correctness of the required oracle information. The static points-to analysis we build on [66] is sound by design and our refinements preserve this property—barring again implementation bugs. Our information-flow tracking profiler, on the other hand, relies on the completeness of the original profiling suite to eliminate any attack surface. While this is a fundamental limitation of dynamic analysis, we found straightforward to obtain the required coverage for a target secret-dependent computation, especially in cryptographic software. Implementation bugs or limitations such as implicit flows (§4.4.1) also apply here. A way to produce a more sound-by-design oracle is to adopt static information-flow tracking, but this also introduces overtainting and hence higher overheads [52].

An incomplete suite might also underestimate a secret-dependent loop bound. Thanks to just-in-time linearization correctness is not affected, but every time the trip count is mispredicted (i.e., real-path loop execution yields a higher count than the oracle), the adversary may observe a one-off perturbation (given that the instrumentation quickly adapts the padding). This is insufficient to leak any kind of high-entropy secret, but one can always envision pathological cases. Similar considerations can be applied to recursive functions.

In principle, other than statically unbound secret-dependent control flows, one can also envision statically unbound secret-dependent data flows such as a secret-dependent heap-allocated object size. We have not encountered such cases in practice, but they can also be handled using just-in-time (data-flow) linearization—i.e., padding to the maximum allocation size encountered thus far during profiling/production runs with similar characteristics.

Part of the residual attack surface are all the code/data accesses fundamentally incompatible with linearization and constant-time programming in general. For instance, on the CFL front, one cannot linearize imbalanced if-else constructs that invoke system calls, or more generally secret-dependent code paths executing arbitrary library/system calls. Their execution must remain conditional. A way to reduce the attack surface is to allow linearization of idempotent library/system calls or even to include some external library/system code in the instrumentation. On the DFL front, one cannot similarly linearize secret-dependent data accesses with external side effects, for instance those to a volatile data structure backed by a memory mapped I/O region (e.g., a user-level ION region [69]). Again, we have not encountered any of these pathological cases in practice.

Similarly, Constantine shares the general limitations of constant-time programming on the compiler and microarchitectural optimization front. Specifically, without specific provisions, a compiler backend may operate optimizations that inadvertently break constant-time invariants at the source (classic constant-time programming) or IR (automated solutions like Constantine) level. Analogously, advanced microarchitectural optimizations may inadvertently re-introduce leaky patterns that break constant-time semantics. Some (e.g., hardware store elimination [25]) may originate new instructions with secret-dependent latencies and require additional wrappers (and overhead). Others (e.g., speculative execution [38]) are more fundamental and require orthogonal mitigations.

6 PERFORMANCE EVALUATION

This section evaluates Constantine with classic benchmarks from prior work to answer the following questions:

(1) What is the impact of our techniques on compilation time?
(2) How is binary size affected by linearization?
(3) What are the run-time overheads induced by CFL and DFL?

Methodology. We implemented Constantine on top of LLVM 9.0 and SVF 1.9 and tested it on a machine with an Intel i7-7800X CPU (Skylake X) and 16 GB of RAM running Ubuntu 18.04. We discuss two striding configurations to conceal memory access patterns with DFL: word size ($\lambda = 4$), reflecting (core colocation) scenarios where recent intra cache level attacks like MemJam [47] are possible, and cache line size ($\lambda = 64$), reflecting the common (cache attack) threat model of real-wold constant-time crypto implementations and also Constantine's default configuration. We use AVX512 instructions to stride over large objects. Complete experimental results when using AVX2 and the $\lambda = 1$ configuration (presently out of reach for attackers) are further detailed in Appendix G. We study:

- 23 realistic crypto modules manually extracted by the authors of SC-Eliminator [78] from a 19-KLOC codebase (SCE suite), used also in the evaluation of Soares et al. [60];
- 6 microbenchmarks used in the evaluation of Raccoon [52] (Raccoon suite)—all we could recover from the source code of prior efforts [41, 42]—using the same input sizes;
- 8 targets used in constant-time verification works: 5 modules of the pycrypto suite analyzed in [76] and 3 leaky functions of BearSSL and OpenSSL studied in Binsec/Rel [20].

For profiling, we divide an input space of 32K elements in 128 equal partitions and pick a random instance from each, producing a profiling input set of 256 inputs. We build both the baseline and the instrumented version of each program at (`-O3`). Table 1 presents our full experimental datasets with the SCE suite (first five blocks) and the Raccoon, pycrypto, and Binsec/Rel suites (one block each).

Validation. We validated the implementation for PC-security and memory access obliviousness with two verifiers. For code accesses, we use hardware counters for their total number and a cycle-accurate software simulation in GEM5. For data accesses, we use `cachegrind` for cache line accesses and write a DBI [21] tool to track what locations an instruction accesses, including predicated `cmov` ones visible at the microarchitectural level. We repeatedly tested the instrumented programs in our datasets with random

Table 1: Benchmark characteristics and overheads.

		IR constructs (sensitive/total)				performance		binary size	
	program	branches	loops	reads	writes	$\lambda = 4$	$\lambda = 64$	$\lambda = 4$	$\lambda = 64$
CHRONOS	aes	0/1	0/1	224/235	0/68	1.13x	1.08x	1.16x	1.16x
	des	0/1	0/1	318/362	0/36	1.19x	1.14x	1.37x	1.73x
	des3	0/3	0/3	861/1005	0/89	1.49x	1.36x	1.92x	2.84x
	anubis	0/1	0/1	776/1240	0/87	1.29x	1.12x	1.27x	1.27x
	cast5	0/1	0/1	333/372	0/36	1.13x	1.06x	1.16x	1.16x
	cast6	-	-	192/204	0/4	1.13x	1.08x	1.01x	1.01x
	fcrypt	-	-	64/74	0/18	1.04x	1.03x	1.01x	1.01x
	khazad	-	-	136/141	0/1	1.13x	1.09x	1.15x	1.15x
S-CP	aes_core	-	-	160/192	0/16	1.12x	1.06x	1.22x	1.22x
	cast-ssl	0/1	0/1	333/355	0/54	1.23x	1.10x	1.24x	1.24x
BOTAN	aes	0/12	0/6	452/525	0/153	1.05x	1.03x	1.36x	1.72x
	cast128	0/2	0/2	333/374	0/52	1.02x	1.01x	1.16x	1.16x
	des	0/1	0/1	136/185	0/24	1.01x	1.01x	1.16x	1.16x
	kasumi	0/7	0/7	96/174	0/18	1.01x	1.01x	1.29x	1.57x
	seed	0/6	0/6	320/360	0/41	1.02x	1.01x	1.18x	1.18x
	twofish	1/8	0/6	2402/2450	4/1084	1.14x	1.12x	1.45x	2.42x
APP-CR	3way	0/4	0/4	0/8	0/14	1.00x	1.00x	1.00x	1.00x
	des	2/10	0/6	134/182	2/17	1.24x	1.09x	1.23x	1.45x
	loki91	16/76	24/28	16/24	0/6	1.51x	1.43x	1.02x	1.02x
LIBGCRYPT	camellia	-	-	32/48	0/48	1.02x	1.01x	1.01x	1.01x
	des	0/2	0/2	144/195	0/12	1.06x	1.06x	1.29x	1.85x
	seed	0/4	0/1	200/265	0/18	1.18x	1.10x	1.22x	1.22x
	twofish	-	-	2574/2662	0/1080	1.97x	1.92x	1.43x	2.24x
RACCOON	binsearch	1/4	1/2	1/3	0/2	1.33x	1.18x	1.01x	1.01x
	dijkstra	3/15	0/5	5/10	3/7	3.45x	1.51x	1.01x	1.01x
	findmax	0/2	0/2	0/1	0/1	1.00x	1.00x	1.00x	1.00x
	histogram	0/2	0/2	1/2	1/1	2.66x	1.68x	1.01x	1.01x
	matmul	0/5	0/5	0/2	0/2	1.00x	1.00x	1.00x	1.00x
	rsort	0/9	4/6	6/8	4/4	1.87x	1.84x	1.30x	1.30x
PYCRYPTO	aes	0/11	0/5	96/223	0/59	1.13x	1.06x	1.19x	1.19x
	arc4	0/3	0/3	3/30	2/10	1.07x	1.03x	1.01x	1.01x
	blowfish	0/16	0/12	24/77	0/39	5.07x	3.17x	1.01x	1.01x
	cast	0/29	0/2	284/321	0/57	1.09x	1.04x	1.37x	1.37x
	des3	0/5	0/1	32/40	0/7	1.06x	1.04x	1.01x	1.01x
B/REL	tls-rempad-luk13	4/17	1/1	6/14	4/17	1.01x	1.01x	1.02x	1.02x
	aes_big	0/45	0/8	32/141	0/40	1.01x	1.01x	1.29x	1.29x
	des_tab	0/50	0/28	8/164	0/97	1.04x	1.02x	1.29x	1.29x
AVG (GEO)	SCE suite	-	-	-	-	1.16x	1.11x	1.22x	1.35x
	Raccoon suite	-	-	-	-	1.68x	1.33x	1.05x	1.05x
	pycrypto suite	-	-	-	-	1.48x	1.30x	1.10x	1.10x
	Binsec/Rel suite	-	-	-	-	1.02x	1.01x	1.19x	1.19x
	all programs	-	-	-	-	1.26x	1.16x	1.17x	1.25x

variations of the profiling input set and random samples of the remaining inputs. We found no visible variations.

Compilation Time. To measure Constantine-induced compilation time, we applied our instrumentation to all the programs in our datasets and report statistics in Table 1. The first four data columns report the sensitive program points identified with taint-based profiling over the randomly generated profiling input set. For the SCE programs, we protect the key scheduling and encryption stages. For brevity, we report figures after cloning and after secret-dependent pushing to nested flows (§4.2): the former affected `des3` and `loki91`, while the latter affected `applied-crypto/des`, `dijkstra`, `rsort`, and `tls-rempad-luk13`. Interestingly, for `3way`, the LLVM optimizer already transformed out a secret-sensitive branch that would be visible at the source level, while no leaky data flows are present in it (consistently with [78]).

Across all 37 programs, the average dynamic analysis time for taint tracking and loop profiling was 4s, with a peak of 31.6s on `libgcrypt/twofish` (~1 C KLOC). For static analysis (i.e., points-to), CFL/DFL transformations, and binary generation, the end-to-end average time per benchmark was 1.4s, with a peak of 23s on `botan/twofish` (567 C++ LOC). Our results confirm Constantine's instrumentation yields realistic compilation times.

Binary Size. Next, we study how our instrumentation impacts the final binary size. Two design factors are at play: cloning for the sake of accurate points-to information and DFL metadata inlining to avoid run-time lookups for static storage. Compared to prior solutions, however, we save instructions by avoiding loop unrolling.

During code generation, we leave the choice of inlining AVX striding sequences to the compiler, suggesting it for single accesses and for small stride sizes with the `cmov`-based method of Appendix C—we observed lower run-time overhead from such choice. When we use word-level striding ($\lambda = 4$), the binary size is typically smaller than for cache line-level striding (λ=64), as the AVX helpers for fast cache line accesses feature more complex logics.

As shown in Table 1, the average binary size increment on the SCE suite is around 1.35x in our default configuration ($\lambda = 64$) and 1.22x for $\lambda = 4$. For `des3`, we observe 1.92-2.84x increases mainly due to cloning combined with inlining. Smaller increases can be noted for the two `twofish` variants, due to DFL helpers inlined in the many sensitive read operations. The binary size increase for all the other programs is below 2x. The Raccoon programs see hardly noticeable differences with the exception of `rsort`, for which we observe a 1.3x increase. We note similar peak values in the two other suites, with a 1.37x increase for `cast` in pycrypto and 1.29x for `aes_big` and `des_tab` in Binsec/Rel. Our results confirm Constantine's instrumentation yields realistic binary sizes.

Run-time Performance. Finally, we study Constantine's run-time performance. To measure the slowdown induced by Constantine on our benchmarks, we measured the time to run each instrumented program by means of CPU cycles with thread-accurate CPU hardware counters (akin [78]). We repeated the experiments 1,000 times and report the mean normalized execution time compared against the baseline. Table 1 presents our results.

Constantine's default configuration produces realistic overheads across all our benchmarks, for instance with a geomean overhead of 11% on the SCE suite and 33% on the Raccoon programs. These numbers only increase to 16% and 68% for word-level protection. Our SCE suite numbers are comparable to those of SC-Eliminator [78] and Soares et al. [60] (which we confirmed using the artifacts publicly released with both papers, Appendix G), despite Constantine offering much stronger compatibility (i.e., real-world program support) and security (i.e., generic data-flow protection and no decoy path side channels) guarantees. On the Raccoon test suite, on the other hand, Raccoon reported two orders-of-magnitude slowdowns (up to 432x) on a number of benchmarks, while Constantine's worst-case slowdown in its default configuration is only 1.84x, despite Constantine again providing stronger compatibility and security guarantees (i.e., no decoy path side channels). Overall, Constantine significantly outperforms state-of-the-art solutions in the performance/security dimension on their own datasets, while providing much better compatibility with real-world programs. For the two other suites, we observe modest overheads with the exception of `blowfish`: its 3.17-5.07x slowdown originates in a hot tight loop making four secret-dependent accesses on four very large tables, a pathological case of leaky design for automatic repair.

7 CASE STUDY

The wolfSSL library is a portable SSL/TLS implementation written in C and compliant with the FIPS 140-2 criteria from the U.S. government. It makes for a compelling case study for several reasons.

From a technical perspective, it is representative of the common programming idioms in real-world programs and is a complex, stress test for any constant-time programming solution (which, in fact, none of the existing solutions can even partially support). As a by-product, it also allows us to showcase the benefits of our design.

Table 2: Characteristics and overheads for wolfSSL.

	baseline	w/o cloning	w/ cloning
functions	84	84	864
binary size (KB)	39	135 (3.5x)	638 (16.35x)
exec cycles (M)	2.6	200 (77x)	33 (12.7x)
accessed objs/point	1	6.29	1.08

	tainted	nested flows (w/o cloning)	w/ cloning nested (tainted)
branches	13	39	1046 (118)
loops	12	31	863 (139)
reads	33	138	2898 (52)
writes	1	91	1892 (2)

	time (ms)	cycles (M)	binary size (KB)
wolfSSL (W=4)	0.35	1.6	39
wolfSSL (W=1)	0.57	2.6	39
wolfSSL (const. time)	0.7	2.9	47
CONSTANTINE (W=1)	8	33	638

The library supports Elliptic Curve (EC) cryptography, which is appealing as it allows smaller keys for equivalent guarantees of non-EC designs (e.g. RSA) [67]. EC Digital Signature Algorithms (ECDSA) are among the most popular DSA schemes today, yet their implementations face pitfalls and vulnerabilities that threaten their security, as shown by recent attacks such as LadderLeak [12] (targeting the Montgomery ladder behind the EC scalar multiplication in ECDSA) and CopyCat [48] (targeting the vulnerable hand-crafted constant-time (CT) wolfSSL ECDSA code).

In this section, we harden with CONSTANTINE the `mulmod` modular multiplication procedure in ECDSA from the non-CT wolfSSL implementation. This procedure calculates a curve point $k \times G$, where k is a crypto-secure nonce and G is the EC base point. Leaks involving k bits have historically been abused in the wild for, e.g., stealing Bitcoin wallets [2] and hacking consoles [1].

Code Features and Analysis. The region to protect comprises 84 functions from the maximal tree that `mulmod` spans in the call graph. We generate a profiling set of 1024 random inputs with 256-bit key length and identify sensitive branches, loops, and memory accesses (Table 2). The analysis of loops is a good example of how unrolling is unpractical. We found an outer loop iterating over the key bits, then 1 inner loop at depth 1, 4 at depth 2, and 3 at depth 3 (all within the same outer loop). Every inner loop iterates up to 4 times, resulting in a nested structure—and potential unroll factor—of 61,440. And this calulation is entirely based on profiling information, the inner loops are actually unbounded from static analysis.

Similarly, cloning is crucial for the accuracy of DFL. We profiled the object sets accessed at each program point with our DBI tool (§6). With cloning, on average, a protected access over-strides (i.e., striding bytes that the original program would not touch) by as little as 8% of the intended storage. Without cloning, on the other hand, points-to sets are imprecise enough that DFL needs to make as many as 6.29x more accesses than strictly needed.

Overheads. Table 2 presents our run-time performance overhead results, measured and reported in the same way as our earlier benchmark experiments. As shown in the table, the slowdown compared to the original non-CT baseline of wolfSSL (using the compilation parameter W=1) is 12.7x, which allows the CONSTANTINE-instrumented version to complete a full run in 8 ms. The compilation parameter W allows the non-CT version to use different double-and-add interleavings over the key bits as part of its sliding window-based double-and-add approach to implement ECC multiplication. In brief, a higher W value trades run-time storage (growing exponentially with W) with steady-state throughput (increasing linearly with W), but also alters the code generated, due to snowball optimization from inlined constants. This choice turns out to be cost-effective in the non-CT world, but not for linearization.

For completeness, we also show results for the best configuration of the non-CT version (which we profiled to be W=4) and the hand-written CT version of wolfSSL. The non-CT code completes an ECC multiplication in 0.35 ms in its best-performing scenario, while the hand-written CT version completes in 0.7 ms. Our automatically hardened code completes in 8ms, that is within a 11.42x factor of the hand-written CT version, using 11.38x more CPU cycles, yet with strong security guarantees for both control and data flows from the articulate computation (i.e., 84 functions) involved.

In terms of binary size increase, with cloning we trade space usage for DFL performance. We obtain a 16.36x increase compared to the reference non-CT implementation, and 13.57x higher size than the CT version. The performance benefits from cloning are obvious (77x/12.7x=6.06x end-to-end speedup) and the size of the binary we produce is 638 KB, which, in absolute terms is acceptable, but amenable to further reduction. In particular, the nature of wolfSSL code is tortured from a cloning perspective: it comprises 36 arithmetic helper functions that we clone at multiple usage sites. We measured, however, that in several cases they are invoked in function instances (which now represents distinct calling contexts for the original program) that see the same points-to information. Hence, after cloning, one may attempt merging back functions from calling contexts that see the same points-to set, saving a relevant fraction of code boat without hampering DFL performance.

Other optimizations, such as our DFL loop optimization also yields important benefits, removing unnecessary striding in some loops—without it, the slowdown would more than double (27.1x). We conclude by reporting a few statistics on analysis and compilation time. The profiling stage took 10m34s, the points-to analysis 20s (~2s w/o cloning), and the end-to-end code transformation and compilation process 1m51s (31s for the non-CT reference).

Overall, our results confirm that CONSTANTINE can effectively handle a real-world crypto library for the first time, with no annotations to aid compatibility and with realistic compilation times, binary sizes, and run-time overheads. CONSTANTINE's end-to-end run-time overhead, in particular, is significantly (i.e., up to two orders of magnitude) lower than what prior comprehensive solutions like Raccoon [52] have reported on much simpler benchmarks.

8 CONCLUSION

We have presented CONSTANTINE, an automatic constant-time system to harden programs against microarchitectural side channels. Thanks to carefully designed compiler transformations and optimizations, we devised a radical design point—complete linearization of control and data flows—as an efficient and compatible solution that brings security by construction, and can handle for the very first time a production-ready crypto library component.

ACKNOWLEDGEMENTS

We thank our shepherd Qi Li and the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and 825377 (UNICORE), and by Intel Corporation through the Side Channel Vulnerability ISRA.

REFERENCES

[1] 2010. Console Hacking 2010. (Dec. 2010). https://fahrplan.events.ccc.de/congress/2010/Fahrplan/events/4087.en.html

[2] 2013. Bitcoin - Android Security Vulnerability. (Aug. 2013). https://bitcoin.org/en/alert/2013-08-11-android

[3] 2015. The M/o/Vfuscator. (Oct. 2015). https://github.com/xoreaxeaxeax/movfuscator

[4] 2019. RISC-V "V" Vector Extension. (Nov. 2019). https://riscv.github.io/documents/riscv-v-spec/riscv-v-spec.pdf

[5] 2020. Google Publishes Latest Linux Core Scheduling Patches So Only Trusted Tasks Share A Core. (Nov. 2020). https://www.phoronix.com/scan.php?page=news_item&px=Google-Core-Scheduling-v9#:~:text=Google%20engineer%20Joel%20Fernandes%20sent,against%20the%20possible%20security%20exploits

[6] Johan Agat. 2000. Transforming out Timing Leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA, USA) *(POPL '00)*. Association for Computing Machinery, New York, NY, USA, 40–53. https://doi.org/10.1145/325694.325702

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[8] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. 870–887. https://doi.org/10.1109/SP.2019.00066

[9] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation.

[10] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 623–639. https://doi.org/10.1109/SP.2015.44

[11] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. 2018. Towards Verified, Constant-Time Floating Point Operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1369–1382. https://doi.org/10.1145/3243734.3243766

[12] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. 2020. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3372297.3417268

[13] G. Barthe, B. Grégoire, and V. Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 328–343. https://doi.org/10.1109/CSF.2018.00031

[14] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *2016 IEEE Symposium on Security and Privacy (SP)*. 987–1004. https://doi.org/10.1109/SP.2016.63

[15] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-Resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 769–784. https://doi.org/10.1145/3319535.3363219

[16] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proc. of the 41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 913–926. https://doi.org/10.1145/3385412.3385970

[17] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. 2010. Optimizing Matlab through Just-In-Time Specialization. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 46–65.

[18] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. 2012. Compiler Mitigations for Time Attacks on Modern X86 Processors. *ACM Trans. Archit. Code Optim.* 8, 4, Article 23 (Jan. 2012), 20 pages. https://doi.org/10.1145/2086696.2086702

[19] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern X86 Processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, USA, 45–60. https://doi.org/10.1109/SP.2009.19

[20] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP '20)*. IEEE Computer Society.

[21] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proc. of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. ACM, 15–27. https://doi.org/10.1145/3321705.3329819

[22] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2011. Mining Hot Calling Contexts in Small Space. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 516–527. https://doi.org/10.1145/1993498.1993559

[23] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2016. Mining Hot Calling Contexts in Small Space. *Software: Practice and Experience* 46, 8 (2016), 1131–1152. https://doi.org/10.1002/spe.2348

[24] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. (2020).

[25] Travis Downs. 2020. Hardware Store Elimination. https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html.

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[27] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2012. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing* (Raleigh, North Carolina, USA) *(STC '12)*. Association for Computing Machinery, New York, NY, USA, 3–8. https://doi.org/10.1145/2382536.2382540

[28] Christopher W Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 213–224.

[29] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[30] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431?473. https://doi.org/10.1145/233551.233553

[31] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. https://doi.org/10.14722/ndss.2020.23018

[32] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, USA, 217–233.

[33] S. He, M. Emmi, and G. Ciocarlie. 2020. ct-fuzz: Fuzzing for Timing Leaks. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 466–471. https://doi.org/10.1109/ICST46399.2020.00063

[34] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 639–650.

[35] Intel. 2020. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. *Developer Zone - Secure Coding* (2020). https://software.intel.com/security-software-guidance/secure-coding/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations

[36] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. {STEALTHMEM}: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 189–204.

[37] Julian Kirsch, Clemens Jonischkeit, Thomas Kittel, Apostolis Zarras, and Claudia Eckert. 2017. Combating Control Flow Linearization. In *ICT Systems Security and Privacy Protection*, Sabrina De Capitani di Vimercati and Fabio Martinelli (Eds.). Springer International Publishing, Cham, 385–398.

[38] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[39] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *IEEE S&P*.

[40] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 973–990.

[41] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 87–101. https://doi.org/10.1145/2694344.2694385

[42] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium (CSF '13)*. IEEE Computer Society, USA, 51–65. https://doi.org/10.1109/CSF.2013.11

[43] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 311–324. https://doi.org/10.1145/2508859.2516692

[44] Heiko Mantel and Artem Starostin. 2015. Transforming Out Timing Leaks, More or Less. In *Proceedings, Part I, of the 20th European Symposium on Computer Security – ESORICS 2015 - Volume 9326*. Springer-Verlag, Berlin, Heidelberg, 447–467. https://doi.org/10.1007/978-3-319-24174-6_23

[45] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–129.

[46] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java *(ISSTA '02)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/566172.566174

[47] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations. *Int. J. Parallel Program.* 47, 4 (Aug. 2019), 538?570. https://doi.org/10.1007/s10766-018-0611-9

[48] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 469–486. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat

[49] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology* (Seoul, Korea) *(ICISC'05)*. Springer-Verlag, Berlin, Heidelberg, 156–168. https://doi.org/10.1007/11734727_14

[50] Robert Muth, Scott Watterson, and Saumya Debray. 2000. Code Specialization Based on Value Profiles. In *Static Analysis*, Jens Palsberg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 340–359.

[51] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (San Jose, CA) *(CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[52] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 431–446.

[53] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. 2019. Get Rid of Inline Assembly through Verification-Oriented Lifting. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 577–589. https://doi.org/10.1109/ASE.2019.00060

[54] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse Representation of Implicit Flows with Applications to Side-Channel Detection. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/2892208.2892230

[55] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[56] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 753–768. https://doi.org/10.1145/3319535.3354252

[57] Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((logN)3) Worst-Case Cost. In *In: Lee D.H., Wang X. (eds) Advances in Cryptology – ASIACRYPT 2011. Lecture Notes in Computer Science, vol 7073*. Springer Berlin Heidelberg, 197–214. https://doi.org/10.1007/978-3-642-25385-0_11

[58] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. and Trends in Prog. Lang.* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

[59] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

[60] Luigi Soares and Fernando Magno Quintao Pereira. 2021. Memory-Safe Elimination of Side Channels. In *(to appear) In Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2021)*.

[61] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1492–1504. https://doi.org/10.1145/2976749.2978411

[62] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

[63] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/2508859.2516660

[64] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (2017), 26–39. https://doi.org/10.1109/MM.2017.35

[65] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. 2003. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. 357–368.

[66] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[67] U.S. National Security Agency. 2016. Commercial National Security Algorithm Suite and Quantum Computing FAQ. (Jan. 2016).

[68] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient out-of-Order Execution. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 991–1008.

[69] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1675–1689. https://doi.org/10.1145/2976749.2978406

[70] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*. 88–105. https://doi.org/10.1109/SP.2019.00087

[71] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. 2011. Eliminating fine grained timers in Xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. 41–46.

[72] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 260–275. https://doi.org/10.1145/2517349.2522728

[73] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*. 494–505.

[74] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. Association for Computing Machinery, New York, NY, USA, 494–505. https://doi.org/10.1145/1250662.1250723

[75] Zhenghong Wang and Ruby B Lee. 2008. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 83–93.

[76] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. 2018. DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 603–620. https://www.usenix.org/conference/usenixsecurity18/presentation/weiser

[77] John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[78] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-Channel Leaks Using Program Repair. In *Proc. of the 27th ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, 15–26. https://doi.org/10.1145/3213846.3213851

[79] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proc. of the 23rd USENIX Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 719–732.

[80] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Cryptographic Hardware and Embedded Systems – CHES 2016*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 346–367.

[81] Ting Yu and Owen Kaser. 1997. A Note on "On the Conversion of Indirect to Direct Recursion". *ACM Trans. Program. Lang. Syst.* 19, 6 (Nov. 1997), 1085–1087. https://doi.org/10.1145/267959.269973

[82] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2011. Predictive Mitigation of Timing Channels in Interactive Systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. Association for Computing Machinery, 563–574. https://doi.org/10.1145/2046707.2046772

[83] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-Based Control and Mitigation of Timing Channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/2254064.2254078

[84] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. 2011. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on security and privacy.* IEEE, 313–328.

[85] Yinqian Zhang and Michael K. Reiter. 2013. DüPpel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, 827–838. https://doi.org/10.1145/2508859.2516741

A DECOY-PATH SIDE CHANNELS

We use the following snippet to show how existing constant-time protection solutions struggle to maintain both memory safety and real execution invariants along decoy paths, ultimately introducing new side channels for attackers to detect decoy paths.

```
1   char last_result;
2   char tableA[8192];
3   char tableB[4096];
4
5   char secret_hash(unsigned int secret) {
6     if (secret < 4096) {
7       register char tmp = tableB[secret];
8       tmp ^= tableA[secret];
9       last_result = tmp;
10    }
11    return last_result;
12  }
```

The if condition at line 6 guards the statement at lines 7-9 (two read operations followed by one write operation). Let us consider the case $4096 <= \text{secret} < 8192$. All the state-of-the-art solutions [19, 52, 60, 78] would also run the corresponding decoy path (statements inside the condition, normally executed only when secret < 4096), but with different code transformations. [19] rewires the memory accesses at lines 7-9 to touch a shadow address, therefore allowing an attacker to detect decoy path execution by observing (three) accesses to the shadow address. [78] preloads both tables before executing the branch, but executes the read/write operations at lines 7-9 with unmodified addresses, introducing a decoy out-of-bounds read at line 7. Such memory safety violation might cause an exception if the memory after tableB is unmapped, which, since the exception is left unmasked, would terminate the program and introduce a termination-based decoy path side channel. [52] closes such termination side channels by masking the exception, but this strategy also introduces an exception handling-based decoy path side channel. [60], on the other hand, replaces such an unsafe read access with an access to a shadow address, which however introduces the same decoy path side channel discussed for [19].

Finally, even assuming no exception is caused by the out-of-bounds read at line 7 and that we can even eliminate the out-of-bounds behavior altogether without introducing other side channels, an attacker can still trivially detect decoy path execution by side channeling the read at line 8. [19]'s shadow access would leak decoy path execution as discussed, but so will all the other solutions [52, 60, 78], which would allow an in-bound access at offset $4096 <= \text{secret} < 8192$ to tableA. Such access would never happen during real execution, breaking a program invariant on a decoy path and introducing a decoy data-flow side channel an attacker can use to again detect decoy path execution.

In contrast, Constantine's combined CFL/DFL strategy would instead ensure the very same data accesses during real or decoy execution, preserving program invariants and eliminating decoy path side channels by construction. Table 3 provides a detailed comparison between Constantine and prior solutions.

B CONDITIONAL SELECTION

The `ct_select` primitive of §4.2 can be instantiated in different ways. We studied how the LLVM compiler optimizes different schemes for constant-time conditional selection to pick the best possible alternative(s) in Constantine.

For the discussion we consider the pointer selection primitive that we use to differentiate decoy and real store operations (i.e., to conditionally select whether we should actually modify memory contents), evaluating the alternatives listed below:

Scheme	C equivalent	*taken* values	CFL overhead
1	`asm cmov`	`{0;1}`	9.6x
2	`ptr = taken ? ptr : (`**void**`*)NULL`	`{0;1}`	7.5x
3	`ptr = (`**void**`*)((size_t)ptr & (-taken))`	`(size_t) {0;1}`	7.3x
4	`ptr = (`**void**`*)((size_t)ptr & taken)`	`{0;0xff..ff}`	7.7x
5	`ptr = (`**void**`*)((size_t)ptr * taken)`	`{0;1}`	7.2x

We assume to operate on a `void* ptr` pointer given as input to the `ct_store` primitive, and *taken* values being 1 on real paths and 0 on decoy ones unless otherwise stated. Instead of reporting end-to-end overheads, we mask the slowdown from DFL by configuring CFL to use a single shadow variable as in the solution of Coppens et al. [19], then we compute the relative slowdowns of our protected `mulmod` wolfSSL version (§7) for the different `ct_select` schemes, using the default non-CT implementation (W=1) as baseline.

Scheme 1 forces the backend to emit `cmov` instructions at the assembly level, similarly to predicated execution mechanism we discussed in §2. As this choice constrains the optimizer's decisions, it turns out to be the worst performing alternative just as expected. Scheme 2 is essentially an LLVM IR `select` statement around the *taken* indirection predicate, on which the compiler can reason about and optimize, then after IR-level optimizations the backend for most of its occurrences emits a `cmov` instruction as in scheme 1, testing the value of the *taken* variable for conditional assignment.

Thus, we investigated different alternatives that could avoid relying on condition flags that, besides, get clobbered in the process and may require frequent recomputation. While mask-based schemes 3 and 4 seemed at first the most promising avenues, it turned out that the additional operation needed either to generate the *taken* mask from a boolean condition (scheme 3), or to maintain it at run-time and combining it with the boolean conditions coming from branch decisions (scheme 4), made these schemes suboptimal. Scheme 5 resulted in the most simple and most efficient one be-

Table 3: Technical, security, and compatibility features from state-of-the-art solutions vs. Constantine.

Feature	Coppens et al.	Raccoon	SC-Eliminator	Soares et al.	Constantine
control flows	predicated	transactional	hybrid	hybrid	linearization
data flows	-	Path ORAM	preloading	-	linearization
loop handling strategy	unroll	unroll	unroll	unroll	just-in-time
integration with compiler	backend	IR level	IR level	IR level	IR level
sensitive region identification	user annotations	annot. + static analysis	annot. + static analysis	user annotations	profiling (taint)
decoy-path side channels	shadow accesses	read/write accesses	read/write accesses	shadow, safe read/write accesses	no
fix variable-latency instructions (e.g., div)	no	sw emulation	no	no	sw emulation
threat model	code	code+data	code+data*	code	code+data
variable-length loops	no	no	decoy paths till bound	no	yes
indirect calls	no	-	-	-	yes
recursion	fixed-depth**	fixed-depth	-	-	yes
spatial safety preserved	yes	no	no	yes	yes
supported data pointers	-	arrays	arrays	arrays	no restrictions

** unimplemented * cache line with preloading

tween the solutions we tested, producing the lowest overhead as it unleashes several arithmetic optimizations (e.g. peephole, global value numbering) at IR and backend optimization levels.

C STRIDING

One of the key performance enablers that back our radical approach is the ability to stride over object fields efficiently. We thoroughly tested different possible implementations, and designed different solutions based on the size of the field that should be strode and the granularity λ at which the attacker could observe memory accesses. Several of these solutions leverage CPU SIMD Extensions: while we focused on AVX2 and AVX512 instructions for x86 architectures, the approach can easily be extended to other architectures supporting vectorization extensions (e.g., ARM SVE [64] , RISC-V "V" [4]). We group our solutions in three categories: simple object striding, vector gather/scatter operations, and vector bulk load/stores.

Simple Object Striding. Given an access on pointer ptr over some field f, the most simple solution is to just access linearly f at every λ-th location. We perform each access using a striding pointer s_ptr at the offset ptr % λ of the λ-th location, so that while striding a field s_ptr will match the target pointer ptr exactly once, on the location where the memory access should happen. For load operations we conditionally maintain the value loaded from memory, propagating only the real result over the striding, while for store operation we load every value we access, conditionally updating it at the right location (§4.3.1). We report a simplified[1] snippet of a striding load operation for a uint8_t, where the conditional assignment is eventually realized e.g. using a cmov operation:

```
uint8_t ct_load(uint8_t* field, uint8_t* field_end, uint8_t* ptr) {
  // Default result
  uint8_t res = 0;

  // Get the offset of the pointer with respect to LAMBDA
  uint64_t target_lambda_off = ((uint64_t) ptr) & (LAMBDA-1);

  // Iterate over the possible offsets
  for(uint8_t* s_ptr = field; s_ptr < field_end; s_ptr += LAMBDA) {
    // Compute the current ptr
    uint8_t* curr_ptr = s_ptr + target_lambda_off;

    // Always load the value
    uint8_t _res = *(volatile uint8_t*)curr_ptr;

    // If curr_ptr matches ptr, select the value
    res = (curr_ptr == ptr)? _res : res;
  }
    return res;
}
```

[1]Additional, constant-time logic is present in the implementation to deal with corner cases, so to avoid striding outside the object if not aligned correctly in memory.

Vector gather/scatter Operations. While for small fields the simple striding strategy presented above performs relatively well, larger sizes offer substantial room for improvement by leveraging SIMD extensions of commodity processors. Vectorization extensions offer instructions to gather (or scatter) multiple values in parallel from memory with a single operation. This allows us to design striding algorithms that touch in parallel N memory locations (up to 16 for x86 AVX512 on pointers that fit into 32 bits). Our algorithm maintains up to N indexes in parallel, from which to access memory from N different locations at the same time. We manage in parallel the multiple accessed values similarly to for simple object striding, with the N results being merged with an horizontal operation on the vector to produce a single value for loads.

Vector Bulk load/stores. Vectorization extensions allow us to load multiple values from memory at once, but a gather/scatter operation is in general costly for the processor to deal with. Depending on the value of λ (e.g., with $\lambda = 1$ or $\lambda = 4$) most of the values could lie on the same cache line. Therefore we further optimized DFL with a third option which simply uses SIMD extensions to access a whole cache line (or half of it with AVX2) with a single operation, thus touching all the bytes in that line. In case of loads, the accessed vector gets conditionally propagated with constant-time operations based on the real address to be retrieved, while for stores it gets conditionally updated, and always written back.

Sizing. Building on empirical measurements on Skylake X and Whiskey Lake microarchitectures, we came up with a simple decision procedure to choose the best possible handler by taking into account the *size* of the field to stride, and the granularity λ at which should be strode. Table 4 reports the average number clock cycles we measured for a striding handler given different values of *size* and λ. We list only the values for load operations for brevity, as the store handler incur in similar effects. We computed the number of cycles needed for each handler to stride over an object as the average of 1000 executions of the same handler, each measured using rdtscp instructions. The SIMD based measurements are based on AVX2.

We can immediately notice how bulk loads are the most effective for small λ values, as they allow for accessing whole cache lines in a single instruction, so this is the default choice for such values. The situation is more complex for higher λ values. We speculate that

Table 4: Clock cycles for different load striding handlers.

handler	λ	size	cycles
simple	64	64	4
gather	64	64	17
bulk	64	64	9
simple	64	512	17
gather	64	512	17
bulk	64	512	26
simple	64	1024	34
gather	64	1024	25
bulk	64	1024	44
simple	4	64	34
gather	4	64	22
bulk	4	64	11
simple	4	512	298
gather	4	512	116
bulk	4	512	44
simple	4	1024	586
gather	4	1024	226
bulk	4	1024	101

the AVX set-up operation for the CPU, and for the management of parallel values which should be merged together to produce the result, are too expensive to be amortized by the few iterations required to stride small objects. Therefore in this case we choose the simple object striding strategy. However, for bigger objects the gather operation is the clear winner, resulting in the minimum overhead. The decision algorithm in pseudocode reads as:

```
if (LAMBDA < 16) select bulk
else if ((striding_size / LAMBDA) < 8) select simple
else select gather
```

D FIELD SENSITIVITY

We improved the field-sensitivity of the Andersen points-to analysis of SVF in order to delay demotion to field-insensitivity and recover, partially or to a full extent, the intended object portions thanks to heuristic inspired by duck typing from programming language research. In short, our extension restricts the surface of the abstract object that can be dereferenced to further improve the field information precision. The extension is semantically sound for the programs we consider, and in most cases (90% for wolfSSL) could refine the SVF results up to the single desired field. We describe our extension using the following running example:

```
%struct.fp_int = type { i32, i32, [136 x i64] } ; size = 1096
%struct.ecc_point = type { [1 x %struct.fp_int], [1 x %struct.fp_int],
                           [1 x %struct.fp_int] } ; size = 3288
%struct.ecc_key = type { i32, i32, i32, i32, %struct.ecc_set_type*,
                i8*, %struct.ecc_point, %struct.fp_int }; size = 4416
```

These datatype declarations in LLVM IR describe the `fp_int`, `ecc_point`, and `ecc_key` structures of wolfSSL. An expression `i{8, 32, 64}` denotes an integer type of the desired bit width. LLVM IR uses pointer expressions for load and store operation that come from a GEP (`GetElementPtr`) instruction such as `%p` below:

```
%v = <some %struct.fp_int object>
%p = getelementptr %struct.fp_int, %struct.fp_int %v, %i32 0, %i32 1
```

The syntax of a GEP instruction is as follows. The first argument is the type, the second is the base address for the computation, and the subsequent ones are indices for operating on the elements of aggregate types (i.e. structures or arrays). The first index operates on the base address pointer, and any subsequent index would operate on the pointed-to expressions. Here `%p` takes the address of the second `i32` field of a `fp_int` structure. What happens with SVF is that it frequently reports a whole abstract object `ecc_key` in the points-to set for `%p`: this information if used as-is would require DFL to access all the 4416 object bytes during linearization.

Starting from this coarse-grained information, our technique identifies which portions of a large object could accommodate the pointer computation. In this simple example, we have that `ecc_key` can host one `fp_int` as outer member (last field), and three more through its `ecc_key` member (second-last). Hence we refine pointer metadata to set of each second `i32` field from these objects, and only four 4-byte locations now require access during DFL. In general, we follow the flow of pointer value computations and determine object portions suitability for such dereferencing as in duck typing.

E RECURSION AND THREAD SAFETY

Our implementation is lackluster in two respects that we could address with limited implementation effort, which we leave to future work as the programs we analyzed did not exercise them.

The first concerns handling recursive constructs. Direct recursion is straightforward: we may predict its maximal depth with profiling and apply the just-in-time linearization scheme seen for loops, padding recursive sequences with decoy calls for depths shorter than the prevision, using a global counter to track depths. For indirect recursion, we may start by identifying the functions involved in the sequence, as they would form a strongly connected component on the graph. Then we may apply standard compiler construction techniques, specifically the inlining approach of [81] to convert indirect recursion in direct recursion, and apply the just-in-time linearization scheme discussed above.

The second concerns multi-threading. As observed by the authors of Raccoon [52], programs must be free of data races for sensitive operations. The linearized code presently produced by CONSTANTINE is not re-entrant because of stack variable promotion (§4.3.3) and for the global variable we use to expose the current taken predicate to called functions. On the code transformation side, an implementation extension may be to avoid such promotion, then use thread-local storage for the predicate, and update the doubly linked lists for allocation sites atomically. As for DFL handlers, we may implement DFL handlers either using locking mechanisms, or moving to more efficient lockless implementations using atomic operations or, for non-small involved sizes (§C), TSX transactions.

F CORRECTNESS

We discuss an informal proof of correctness of the CONSTANTINE approach. As we anticipated in §5, to prove that our programs are semantically equivalent to their original representations, we break the claim into two parts: control-flow correctness and data-flow correctness For each part we assume that the other holds, so that the initial claim can hold by construction.

For control flows, we need to show that along real paths the transformed program performs all and only the computations that the original one would make. First, we rule out divergences from exceptional control flow since the original program is error-free and CFL sanitizes sequences that may throw (e.g., division) when in dummy execution. We then observe that by construction CFL forces the program to explore both outcomes of every branch, and the decision whether to treat each direction in dummy execution

Table 5: Run-time overhead analysis of different Constantine configurations (i.e., $\lambda = 1$, AVX2). The numbers for SC-Eliminator and Soares et al. were obtained from executing the publicly available artifact evaluation material for their papers.

	program	AVX512 ($\lambda = 1$)	AVX512 ($\lambda = 4$)	AVX512 ($\lambda = 64$)	AVX2 ($\lambda = 4$)	AVX2 ($\lambda = 64$)	SC-Eliminator	Soares et al.
CHRONOS	aes	1.13	1.13	1.08	1.22	1.08	1.11	1.02
	des	1.12	1.19	1.14	1.36	1.15	1.09	1.00
	des3	1.49	1.49	1.36	1.86	1.37	1.12	1.02
	anubis	1.29	1.29	1.12	1.55	1.22	1.06	1.00
	cast5	1.13	1.13	1.06	1.25	1.12	1.06	1.02
	cast6	1.13	1.13	1.08	1.13	1.07	1.05	1.00
	fcrypt	1.04	1.04	1.03	1.06	1.01	1.05	1.00
	khazad	1.13	1.13	1.09	1.23	1.07	1.30	1.00
S-CP	aes_core	1.12	1.12	1.06	1.01	1.05	1.06	1.04
	cast-ssl	1.23	1.23	1.10	1.38	1.15	1.17	1.01
BOTAN	aes	1.05	1.05	1.03	1.09	1.01	1.01	-
	cast128	1.02	1.02	1.01	1.03	1.01	1.05	-
	des	1.01	1.01	1.01	1.01	1.01	1.08	-
	kasumi	1.01	1.01	1.01	1.03	1.01	1.01	-
	seed	1.02	1.02	1.01	1.03	1.01	1.03	-
	twofish	1.14	1.14	1.12	1.21	1.15	1.04	-
APP-CR	3way	1.00	1.00	1.00	1.00	1.00	1.03	1.15
	des	1.24	1.24	1.09	1.27	1.06	1.08	1.11
	loki91	1.51	1.51	1.43	1.48	1.48	1.97	1.24
LIBGCRYPT	camellia	1.02	1.02	1.01	1.02	1.01	1.08	1.01
	des	1.06	1.06	1.06	1.05	1.09	1.03	1.01
	seed	1.18	1.18	1.10	1.21	1.01	1.15	1.01
	twofish	1.97	1.97	1.92	2.45	2.10	1.41	1.24
RACCOON	binsearch	1.33	1.33	1.18	1.30	1.16	-	-
	dijkstra	3.87	3.45	1.51	2.83	1.50	-	-
	findmax	1.00	1.00	1.00	1.00	1.00	-	-
	histogram	2.66	2.66	1.68	4.39	1.87	-	-
	matmul	1.00	1.00	1.00	1.00	1.00	-	-
	rsort	1.87	1.87	1.84	1.50	1.45	-	-
PYCRYPTO	aes	1.13	1.13	1.06	1.33	1.11	-	-
	arc4	1.07	1.07	1.03	1.08	1.03	-	-
	blowfish	5.07	5.07	3.17	10.58	3.23	-	-
	cast	1.09	1.09	1.04	1.16	1.08	-	-
	des3	1.06	1.06	1.04	1.08	1.05	-	-
B/Rel	tls-rempad-luk13	1.01	1.01	1.01	1.01	1.01	-	-
	aes_big	1.02	1.01	1.01	1.02	1.01	-	-
	des_tab	1.04	1.04	1.02	1.07	1.03	-	-
	geomean (total)	1.26	1.26	1.16	1.34	1.17	1.12	1.05

depends on the *taken* predicate value. CFL builds this predicate as the combination of the control-flow decisions that the (original) program takes on the program state, and from the data flow argument decoy paths have no effects on such state. All the linearized branch directions will be executed as many times and in the same interleaving observable in the original program; as for the special loop case, the amount of real and decoy iterations depends on the original loop condition and the *taken* predicate, so its real iterations closely match the original loop. This completes the argument.

For data flows, we need to show that values computed in dummy execution cannot flow into real paths, and that decoy paths preserve memory safety. The points-to metadata fed to DFL load and store wrappers make the program access the same memory objects along both real and decoy paths, and allocation metadata ensure that those objects are valid: memory safety is guaranteed. Also, only real paths can change memory contents during a store, hence only values written by real paths can affect data loads. Thus, we only need to reason about data flows from local variables assigned in dummy execution. LLVM IR hosts such variables in SSA virtual registers, and at any program point only one variable instance can be live [55]. For a top-level linearized branch, a `ct_select` statement chooses the incoming value from the real path (§4.2.1). For a nested branch (Figure 2) both directions may be part of dummy execution. Regardless of which value the inner `ct_select` will choose, the outer one eventually picks the value coming from the real path that did not contain the branch. Extending the argument to three or more nested branches is analogous: for a variable that outlives a linearized region, whenever such variable is later accessed on a real path, the value from a real path would assign to it (otherwise the original program would be reading an undefined value or control-flow correctness would be violated), while in decoy paths bogus value can freely flow. We discussed correctness for loops in §4.2.4.

G COMPLETE RUN-TIME OVERHEAD DATA

Table 5 shows the complete set of our performance-oriented experiments: we benchmarked Constantine with different λ values $(1, 4, 64)$ and SIMD capabilities (AVX2 and AVX512), and also ran the artifacts from [78] and [60] on the same setup used for §6. For the latter we did not try the Raccoon microbenchmarks, mostly due to compatibility problems and limitations of the artifacts, while the SCE suite was part of their original evaluation (Soares et al. leave the `botan` group out of the artifact evaluation harness). Both systems provide much weaker security properties than Constantine, yet the average overhead numbers we observe are similar. Also, the availability of AVX512 instructions brings benefits for the $\lambda = 64$ setting as they allow DFL to touch more cache lines at once over large object portions (§C). Interesting, protection for the presently unrealistic $\lambda = 1$ attack vector leads to overheads that are identical to the $\lambda = 4$ configuration for MemJam-like attacks, with `dijkstra` being the only exception (3.87x vs 3.45x).