DECLASSIFLOW: A Static Analysis for Modeling Non-Speculative Knowledge to Relax Speculative Execution Security Measures

Rutvik Choudhary
rutvikc2@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Alan Wang
alanlw2@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Zirui Neil Zhao
ziruiz6@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Adam Morrison
mad@cs.tau.ac.il
Tel Aviv University
Tel Aviv, Israel

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

ABSTRACT

Speculative execution attacks undermine the security of constant-time programming, the standard technique used to prevent microarchitectural side channels in security-sensitive software such as cryptographic code. Constant-time code must therefore also deploy a defense against speculative execution attacks to prevent leakage of secret data stored in memory or the processor registers. Unfortunately, contemporary defenses, such as speculative load hardening (SLH), can only satisfy this strong security guarantee at a very high performance cost.

This paper proposes DECLASSIFLOW, a static program analysis and protection framework to *efficiently* protect constant-time code from speculative leakage. DECLASSIFLOW models "attacker knowledge"—data which is inherently transmitted (or, *implicitly* declassified) by the code's *non-speculative* execution—and statically removes protection on such data from points in the program where it is already guaranteed to leak non-speculatively. Overall, DECLASSIFLOW ensures that data which never leaks during the non-speculative execution does not leak during speculative execution, but with lower overhead than conservative protections like SLH.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

Speculative execution attacks, Static analysis, Software-based defense

ACM Reference Format:
Rutvik Choudhary, Alan Wang, Zirui Neil Zhao, Adam Morrison, and Christopher W. Fletcher. 2023. DECLASSIFLOW: A Static Analysis for Modeling Non-Speculative Knowledge to Relax Speculative Execution Security Measures.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623065

In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623065

1 INTRODUCTION

Security-sensitive programs, such as cryptographic software, perform computations over secret data (e.g., cipher keys and plaintext or personal information). Secure software must prevent its secrets from being "leaked" over microarchitectural side channels, which occur when secret data is passed as the operand to a *transmitter* instruction. A transmitter is an instruction whose execution creates operand-dependent hardware resource patterns that can potentially be observed ("received") by the attacker, allowing the attacker to learn information about the transmitter's operand. Classic examples of transmitters are load and branch instructions, whose execution makes operand-dependent changes to the cache state [31, 44] and instruction sequence. However, numerous other "variable time" instructions are also considered transmitters [4, 19, 24].

Traditionally, the standard technique for preventing secret leakage over microarchitectural side channels is to use *constant-time programming* (also called *data-oblivious programming*). Constant-time code performs its computation without passing secret-dependent data as arguments to transmitter instructions [4, 7, 8, 28, 35, 37].

Unfortunately, the discovery of speculative execution (or Spectre) attacks [6, 10, 29, 33, 43] undermines the constant-time approach [10, 14, 38, 45]. The problem is that constant-time guarantees are based on correct execution semantics and may not hold in an illegal mis-speculated execution created by a speculative execution attack. For example, misprediction of a loop branch [45], function return [14], indirect call [10], and so on can cause the processor to jump to a transmitter with the transmitter's operand holding secret data, even though the transmitter's operand would never contain secret data in a correct execution (see Figure 1).

Consequently, constant-time code must additionally deploy a defense against speculative execution attacks. Importantly, this defense must prevent speculative leakage not only of speculatively-accessed data (read from memory under mis-speculation) [47] but also of *non-speculatively-accessed data* that already exists in processor registers when mis-speculation begins (as in Figure 1). Contemporary defenses can only satisfy this strong security guarantee

```
for (int i = 0; i < NUM_ROUNDS; i++) {
  S = AES_Round(S, round_key[i]);
}
Tr(S);
```

Figure 1: Example of constant-time code breaking due to speculative execution [45]. Misprediction of the loop branch (`i<NUM_ROUNDS`) can cause an intermediate value of the AES state `S` to be passed to a transmitter, `Tr(·)`.

by blocking speculation of all transmitters, which incurs a high performance cost [36, 48]. We therefore ask: *how can constant-time code be efficiently protected from speculative execution attacks?*

We answer this question with DECLASSIFLOW, a static program analysis and protection framework that can relax speculative execution defenses. Our approach enforces the security property proposed by Speculative Privacy Tracking (SPT) [18]: data that never leaks during non-speculative execution does not leak during speculative execution. This property implies that data which gets *implicitly declassified*, due to being passed as the operand of a transmitter in the program's non-speculative execution, does not need to be protected during the program's speculative execution. This enables the safe removal of protection mechanisms and commensurately lower performance overhead.

Leveraging the SPT security property to reap performance benefits is non-trivial, however. SPT is only able to achieve gains by introducing hardware mechanisms for dynamically tracking non-speculative leakage and disabling protection at run-time. But SPT hardware is not available in current processors and its future adoption status is not clear. In this paper, we leverage the SPT security property *purely in software*. The resulting approach can be deployed to improve the performance of constant-time code today. DECLASSIFLOW can also identify protection relaxations that SPT hardware misses, because DECLASSIFLOW is a static program analysis that reasons about all possible program executions, whereas SPT hardware operates only based on the program's current execution.

In a nutshell, DECLASSIFLOW performs program analysis to determine the "attacker's knowledge" at every edge in the program's non-speculative control-flow graph, where "attacker knowledge" refers to the data guaranteed to be implicitly declassified (declared non-secret) by the non-speculative execution if said control transfer occurs at run-time. DECLASSIFLOW can thus identify data that is *guaranteed* to leak if execution reaches a program point (although it may not leak at that point, but only later in the execution).

We use DECLASSIFLOW's analysis to relax speculative execution protections, such as SLH, for several constant-time programs. By reasoning about attacker knowledge, DECLASSIFLOW is able to deduce that many transmitters (e.g., loads) leak information about the "same thing" (e.g., the base address of an array) and that this information is guaranteed to be known in the program's non-speculative execution. This enables DECLASSIFLOW to reduce overhead significantly; in some cases, replacing all protection instrumentation with a single mechanism (e.g., a barrier) that guarantees the program is entered non-speculatively.

To summarize, we make the following contributions.

(1) We propose an abstraction, *non-speculative knowledge*, for deducing a program region where a variable will "inevitably" be leaked in the program's non-speculative execution.
(2) We propose a novel program analysis that can calculate non-speculative attacker knowledge at each program edge, and strategies for placing protection primitives based on said non-speculative attacker knowledge.
(3) We evaluate the impact of our analysis on three constant-time benchmarks and demonstrate that our analysis leads to more efficiently protected programs.

Our analysis is open source, and can be found at https://github.com/FPSG-UIUC/declassiflow.

2 BACKGROUND

2.1 Programs, Executions, and Traces

2.1.1 The Building Blocks of a Program. We consider programs written in the LLVM assembly language [2], which uses static single assignment (SSA) (Section 2.2).

A *program* is a list of instructions that perform computations on variables. $\mathbb{V}$ denotes the set of all variables in the program, and $2^{\mathbb{V}}$ denotes the powerset of $\mathbb{V}$.

Instructions in a program are partitioned into smaller lists, *basic blocks* (or *blocks* for short), defined in the usual way. They are typically denoted as B, often with a subscript. We use $\mathbb{B}$ to denote the set of all blocks in the program.

A *control-flow edge* (or *edge* for short) is an ordered pair (B_i, B_j) for some $B_i, B_j \in \mathbb{B}$. A control-flow edge exists between B_i and B_j if it is theoretically possible to execute B_j immediately after B_i has been executed. Edges are denoted as e, often with a subscript. We use $\mathbb{E}$ to denote the set of all edges in the program. For any block B, we denote the set of its input edges and output edges as $E_{\text{in}}(B)$ and $E_{\text{out}}(B)$ respectively.

A *control-flow graph* is a directed graph where the nodes are given by $\mathbb{B}$ and the edges are given by $\mathbb{E}$. We define a node $\textsc{Entry} \in \mathbb{B}$ which is the singular point of entry for the program as well as a node $\textsc{Exit} \in \mathbb{B}$ which is the singular exit point of the program.[1] By definition, $E_{\text{in}}(\textsc{Entry}) = \emptyset$ and $E_{\text{out}}(\textsc{Exit}) = \emptyset$.

For any two blocks $B_i, B_j \in \mathbb{B}$, B_i *dominates* B_j, denoted $B_i \,\text{dom}\, B_j$, if every path from $\textsc{Entry}$ to B_j goes through B_i. B_j *post-dominates* B_i, denoted $B_j \,\text{pdom}\, B_i$, if every path from B_i to $\textsc{Exit}$ goes through B_j. B_i is a *predecessor* of B_j if $(B_i, B_j) \in \mathbb{E}$.

We say that an edge $e = (B_i, B_j)$ dominates a block B' if $B_j \,\text{dom}\, B'$. Similarly, B' dominates edge e if $B' \,\text{dom}\, B_i$. A variable x dominates edge e if the block in which x is defined, denoted as B_x, dominates B_i. Similarly, e dominates x if $B_j \,\text{dom}\, B_x$.

A *region*, typically denoted R, is a set of blocks such that: (1) there is one block in R, known as the *header*, that dominates all others; (2) for any two blocks $B_i \in \mathbb{B}$ and $B_j \in R$, if there is a path from B_i to B_j that doesn't contain the header, then B_i is in R [3].

Edge (B_i, B_j) is a *back edge* if $B_j \,\text{dom}\, B_i$. By convention, every block dominates itself, and so self edges (edges where $B_i = B_j$) are considered back edges. If there is a back edge in the control-flow graph, then there is a cycle. Cycles in the control-flow graph are typically created by using loop constructs (e.g. `for` and `while`).

2.1.2 Executions and Traces. A *non-speculative execution* of a program P on some input is the sequence of instructions P executes

[1] Not all paths through the control-flow graph will reach EXIT, e.g. in the case of non-terminating loops.

according to the semantics of the LLVM language [2]. We say that the k-th instruction in an execution occurs at *time* k. An execution *traverses* edge $e = (B_i, B_j)$ at time k if the k-th and $(k+1)$-th instructions are the last instruction in B_i and the first instruction in B_j, respectively.

A *trace*, typically denoted as t, is a sequence of edges. A trace t is *realizable* if, for some execution of the program, t is the sequence of edges traversed by the execution. Realizable traces thus model the control flow of executions. We refer to them interchangeably for brevity, understanding that every realizable trace is associated with an execution. The set of all realizable traces of the program is denoted $\mathbb{T}$. An edge e is *realizable* if $e \in t$ for some $t \in \mathbb{T}$.

We use $\mathbb{P}$ to denote all possible paths through the control-flow graph, including those that do not correspond to a realizable trace. By definition, $\mathbb{T} \subseteq \mathbb{P}$.

We now extend our execution semantics to capture *speculative executions*. We consider control-flow speculation of branches whose speculative target is consistent with the control-flow graph. That is, for $B_i \neq B_j$, a speculative execution executes instruction $I \in B_i$ followed by instruction $I' \in B_j$ only if I is the last instruction in B_i, I' is the first instruction in B_j, and $e = (B_i, B_j) \in \mathbb{E}$. The semantics of the LLVM assembly language [2] can be extended to model this form of speculation by adding microarchitectural events [2] for mispredicted control-flow instructions and eventual rollback of a mis-speculated sequence of instructions; this would be similar to various semantics developed in prior work [25–27, 40].

The implications of the above speculative semantics on our analysis are discussed in Section 3.

2.2 Single Static-Assignment Form

Our analysis works with code written in single static-assignment (SSA) form. This is a popular and useful abstraction that makes program data dependencies clear and variable identities unambiguous. Code is in SSA form when any usage of a variable is reached by exactly one definition of that variable [20].

In code with control flow, a variable's definition may depend on earlier control-flow decisions. In order to make such code SSA-compliant, control flow-dependent definitions at join points are assigned to by ϕ-functions [20] which return an input based on the control-flow decision. We define the semantics of the ϕ-function as follows: suppose in some block B we have an instruction I_ϕ that is a ϕ-function, $y = \phi(x_1, \ldots, x_N)$. By definition, there are N edges into B, denoted $e_1, \ldots, e_N$. The ϕ-function is defined such that at any point in any execution, if e_i is the last edge to reach I_ϕ, then $y = x_i$. An important consequence of the semantics of a ϕ-function is that every x_i must be defined *prior* to B. This is simply because it must be a usable name by the time the ϕ-function is encountered.

As an example, the following non-SSA code on the left is transformed to produce SSA code on the right.

```
x = 0;
if (...) {
  x = x + 1
}
print(x)
```

```
x1 = 0;
if (...) {
  x2 = x1 + 1
}
x3 = ϕ(x1,x2)
print(x3)
```

3 SETTING AND SECURITY GOAL

The goal of DECLASSIFLOW is to efficiently prevent speculative leakage of sensitive data while maintaining a useful security guarantee.

First and foremost, we define the points of potential information "leakage." A *transmitter* is any instruction whose execution exhibits operand-dependent hardware resource usage. Classic examples of transmitters are loads and branches [30]. Depending on the microarchitecture, there may be others [4, 19, 41, 45, 48]. We say that the operands (data) passed to a transmitter are *leaked*. Note that a transmitter may leak its operands fully or partially.

A *non-speculative transmitter* is one that appears in the program's non-speculative execution. A *speculative transmitter* appears in the program's speculative execution. That is, it appears as an operand-dependent microarchitectural event in the program's speculative semantics (Section 2.1.2), which may or may not correspond to an instruction that architecturally retires.

We assume the standard attacker used in the constant-time programming setting [18, 27, 40, 46, 47]. Here, the attacker knows the victim program. The attacker further sees a projection, or view, of the victim's non-speculative and speculative executions: namely, (1) the sequence of values taken by the program counter (PC), and (2) the sequence of values passed to transmitters.

With the above in mind, there are two main protection guarantees a speculative execution defense can have [47]. The first prevents speculatively-accessed data from being passed to speculative transmitters. Enforcing this policy satisfies "weak speculative non-interference" [27], and is sufficient to eliminate universal read gadgets and defend programs in sandbox settings [34]. As a result, there has been significant interest in both hardware [22, 46, 47] and software [23, 40] defenses that provide said guarantee.

Unfortunately, such defenses are not comprehensive as there are still important applications—namely constant-time cryptography—that *non-speculatively* read and compute on sensitive data but can still leak said data *speculatively* [18, 27, 32, 42, 47]. To protect these programs, one requires a defense with a broader protection guarantee: i.e., one that prevents both speculatively *and non-speculatively* accessed data from being passed to speculative transmitters. Defense mechanisms that meet this guarantee provide *complete* protection from speculative execution attacks but typically come at high performance overhead, i.e., they are tantamount to delaying every transmitter's execution until they become non-speculative or are squashed [36, 42, 48].

The aforementioned protections are (almost always) overly conservative because they implicitly treat *all* data as "secret" and deserving of protection. Yet, not all data is semantically secret. Software annotations could directly convey what is and is not secret, but there are major disadvantages to programmer intervention. For example, programmer intervention/expert labeling cannot be applied to legacy code already deployed. So, to determine what data

[2] A "microarchitectural event" can be thought of as an instruction that produces (possibly operand-dependent) microarchitectural changes but no architectural changes.

is secret without requiring expert labeling/intervention, we adopt a definition of "secret" proposed by SPT [18],

Definition 1. *Data x is secret if there is no data flow from it to an operand of a non-speculative transmitter, where data flow refers to flow through LLVM SSA variables and LLVM data memory.*

This definition is motivated by the constant-time programming model in which sensitive data is never passed to non-speculative transmitters. The contrapositive of this is that if any data is passed to a non-speculative transmitter, it is not sensitive, i.e. not "secret".

Definition 1 can be interpreted as enabling efficient implementations that satisfy *generalized constant-time* (GCT) [27]. Denote the non-speculative and speculative program semantics as S^{nspec} and S^{spec}, respectively. For brevity, we also assume these semantics encode the attacker's view, e.g., the set of transmitters. Given, a program P and a policy Π which defines what program variables are "high" (secret), P satisfies GCT w.r.t. S^{nspec}, S^{spec} and Π if the following requirements hold.

(1) Executions of P on S^{nspec} satisfy non-interference w.r.t. Π. That is, attacker observations of P's execution, given S^{nspec}, are independent of the values in Π.
(2) Executions of P on S^{nspec} that satisfy non-interference w.r.t. Π must also satisfy non-interference on S^{spec} w.r.t. Π.

Requirement 2 is referred to as *speculative non-interference* or SNI for short [26, 27].

Given this context, we can view Declassiflow as a function $P' = D(P; S^{\text{nspec}}, S^{\text{spec}})$ that takes a program P as input, produces a program P' as output, and is parameterized by S^{nspec} and S^{spec}. Suppose P satisfies Requirement 1; it need not satisfy Requirement 2. For a specified S^{nspec} and S^{spec}, D outputs a P' that is functionally equivalent to P and now (additionally) satisfies Requirement 2, i.e., now satisfies SNI and therefore GCT.

Importantly, D did not require Π as an input, but rather infers a policy Π_{Decl} which is sound w.r.t. Π. That is, $\Pi \subseteq \Pi_{\text{Decl}}$. This is possible because D has access to P, which already enforces Π. At the same time, Π_{Decl} will provide a basis for implementing efficient protection. That is, if Π_{All} denotes the policy (described above) that treats all data as secret, we have that $\Pi_{\text{Decl}} \subseteq \Pi_{\text{All}}$ in theory and $\Pi_{\text{Decl}} \subset \Pi_{\text{All}}$ in practice. This will allow us to more efficiently protect programs without additional programmer intervention or labeling, beyond the program being written to enforce non-speculative/vanilla constant-time execution.

3.1 Semantics and Transmitters

D is parameterized by S^{nspec} and S^{spec}, which encode the execution semantics and transmitters.

Semantics. For security, our analysis assumes the semantics set forth in Section 2.1.2, in particular that the speculative semantics is restricted to control-flow speculation that remains on the control-flow graph. This is sufficient to protect non-speculatively accessed data in the presence of direct branches (similar to those found in Spectre Variant 1). To block leakage due to other forms of speculation (e.g., indirect branches whose *targets* are predicted, as in Spectre variant 2), our analysis can adopt complementary defenses such as "retpoline".[3]

[3] See https://support.google.com/faqs/answer/7625886

Transmitters. Our analysis is flexible with respect to which instructions are considered transmitters. For the rest of the paper, we assume loads are transmitters that can execute speculatively. We assume that branches and stores are also transmitters, but only if they appear in the non-speculative execution. That is, we assume that branches and stores do not change microarchitectural state in an operand-dependent way until they become non-speculative. We note, this still allows for branch prediction; it just stipulates that said predicted branches only resolve (and redirect execution) when they become non-speculative. To reiterate: these choices were not fundamental, and the analysis can be modified to account for other transmitters and their speculative vs. non-speculative behavior.

4 ACHIEVING EFFICIENT PROTECTION

We now describe an analysis, dubbed Declassiflow, that enables low-overhead protection for "secrets" as given by Definition 1.

To understand our scheme's security and performance, we start by considering a secure but high-overhead software-based protection. We will use the abstraction proposed by Blade [40], which introduces a primitive called `protect(v)`. `protect` wraps a variable `v` and delays its usage until it becomes non-speculative (or *stable* [40]). Blade points out that, while current hardware does not support `protect`, `protect` can be emulated today by introducing control-flow-dependent data dependencies [23], speculation barriers, or a combination of the two. Regardless of how it is implemented, executing `protect` incurs overhead by delaying an instruction's (and its dependents') execution. This is especially pronounced when said instruction is on the critical path for instruction retirement (as is typical with loads).

As discussed in Section 3, we wish to protect non-speculatively accessed data. Thus, a secure baseline defense must protect the operands of all transmitters that can execute speculatively. We express this by wrapping said transmitters' operands with `protect` statements, placed immediately before each transmitter and in the same block.[4] This approach is tantamount to that of several recent defense proposals [36, 42, 48].

An example of our baseline is shown in Figure 2a, which depicts a program that contains a transmitter inside a loop as well as at the exit point. The transmitters are denoted `Tr(·)`. While this approach is secure, it is also expensive; the `protect(x)` statements can be encountered an unbounded number of times (depending on the semantics of the loop). Given this strict policy, which effectively prevents transmitters from executing speculatively, nothing more can be done to improve performance in this example.

However, if we instead consider Definition 1 and its implications, we can see that this program contains unnecessary protection. From the discussion surrounding that definition, we saw that data which does not meet the definition for "secret" does *not* need to be protected from speculative leakage. This is the manner in which we can reduce protection overhead. To aid in this process, we define a more useful concept that is core to our work,

Definition 2. *A variable x is considered known when it is guaranteed to be passed to a non-speculative transmitter (i.e., its value*

[4] We note that this protection scope is broader than Blade's (which only protects speculatively-accessed data), hence we don't compare to Blade further.

will inherently be revealed) or when its value can be inferred from other known variables.

We say that a variable can be "inferred" from other known variables if its value can be computed via a polynomial-time algorithm [5] from the values of said other variables. We consider the set of known variables over time to be the attacker's *non-speculative knowledge* (or just "knowledge" for short). One key addition made by Definition 2 is that a variable can be considered known not only when it is observed to be non-secret, but even when it is guaranteed to *eventually* become non-secret. That is, for the purposes of our analysis, inevitable non-secrecy is as good as knowledge.

Our goal is to use Definition 2 to derive a minimal set of locations at which to place `protect` statements. For this, we need to define one more concept: the *non-speculative knowledge frontier* (*knowledge frontier* for short) for each program variable. Intuitively, the knowledge frontier for a variable x represents the earliest points in the program such that, if the program's non-speculative execution "crosses" the knowledge frontier, x will be known. We define it more precisely as,

DEFINITION 3. *For any variable x, let $K_{\mathbb{B}}(x)$ denote the set of blocks in which x is known. The knowledge frontier of x, denoted $\mathcal{F}(x)$, is the smallest subset of $K_{\mathbb{B}}(x)$ such that for any B in $K_{\mathbb{B}}(x)$, there is no path from ENTRY to B that does not contain some B' from $\mathcal{F}(x)$.*

Note that by this definition, the knowledge frontier for a variable in a given program is *unique*. We can reframe what is required of a protection mechanism to enforce SNI (with respect to the policy implied by Definition 1) in terms of the knowledge frontier,

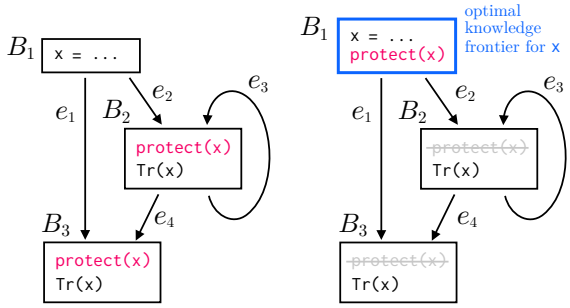
PROPERTY 1. *(Frontier Protection Property for x) A placement of* `protect(x)` *statements enforces SNI with respect to x if and only if no speculative execution can transmit a function of x before the non-speculative execution crosses the knowledge frontier for x.*

One straightforward strategy to satisfy Property 1 is to add `protect` statements only "along the knowledge frontier" for each given variable, as opposed to at the site of each transmitter.

Example. Look again at the program in Figure 2. As mentioned, a naive protection scheme places `protect(x)` statements in the same blocks as all transmitters. However, if the execution enters B_1 non-speculatively, it will necessarily (non-speculatively) enter either B_2 or B_3 next. Thus, the knowledge frontier for x is B_1. With this, we can re-instrument the code with a `protect(x)` inserted in B_1 only. Crucially, we have removed the `protect(x)` from the loop, which means that protection overhead will likely be amortized.

This is more aggressive than what is possible with the hardware-based defense SPT, on which Definition 1 is based. Since SPT doesn't know the program's structure, it doesn't know if there is a path from B_1 where x is not leaked non-speculatively, and hence it falls back to a baseline protection that is tantamount to Figure 2a. As mentioned, an interesting aspect of Definition 2 is that it allows for variables to be known "ahead of time" (i.e. before they are actually passed to a transmitter). This is a key difference between our approach and SPT's; knowledge that is inevitable in the future is knowledge that can be exploited "now." SPT on the other hand must wait to observe transmitters retire before it treats its operands as known.

[5] This is to admit computational assumptions. For example, knowing a plaintext and its corresponding AES ciphertext should not give one the ability to "infer" the key!



(a) Code protected via prior work **(b) Code protected via our work**

Figure 2: Making the protected code (left) more efficient (right).

5 MODELING ATTACKER KNOWLEDGE

We will now define non-speculative attacker knowledge, which will be used to compute the non-speculative knowledge frontier.

5.1 Non-Speculative Knowledge

We model the attacker's knowledge at the granularity of variables, and we treat knowledge in a binary fashion; the value of a variable is either "fully" known to an attacker or no function of the variable is known. Thus, an attacker's knowledge is a subset of $\mathbb{V}$, and the full space of the knowledge of an attacker is $2^{\mathbb{V}}$.

Per Definition 2, in any trace, a variable is considered known at the point when (and any time after) it is passed to a non-speculative transmitter (i.e., when its value is revealed), or at the point (and any time after) its value can be inferred from other known variables. Rather than keeping track of knowledge "temporally" by tracking it over time (per trace), we can instead capture knowledge "spatially" by mapping it onto the control-flow graph. We introduce the map $K_{\mathbb{E}} : \mathbb{E} \to 2^{\mathbb{V}}$ which represents the distribution of knowledge over edges. We precisely define $K_{\mathbb{E}}$ as follows,

DEFINITION 4. *Take any edge $e \in \mathbb{E}$. We have $x \in K_{\mathbb{E}}(e)$ if and only if for all traces $t \in \mathbb{T}$ such that $e \in t$, x is already known or is guaranteed to become known every time the execution corresponding to t traverses e. If $x \in K_{\mathbb{E}}(e)$, we say "x is known on edge e".*

There are two important clarifications we wish to make with respect to the above definition. First and foremost, $x \in K_{\mathbb{E}}(e)$ does *not* necessitate that e is dominated by the definition of x since the value of x may be inferable from other known variables, as discussed above. Second, the manner in which we define knowledge and the way we intend to use it leave open the possibility for *vacuous knowledge*. A variable is *vacuously known* if it is deduced to be known on a path on which it will never be defined. Crucially, since such a variable is not defined on this path, it cannot be used. Thus, for the purposes of optimizing a defense mechanism, this knowledge is inactionable. We will see in the next section that vacuous knowledge is an important concept, particularly when analyzing ϕ-functions.

5.2 Instructions as Equations

We now define a set of relations that describe knowledge $K_{\mathbb{E}}$ with respect to a concrete program.

5.2.1 Non-ϕ instructions. We first discuss how knowledge is propagated through non-ϕ instructions. An instruction I is said to be *deterministic* if it can be represented as an equation of the form $y = f(x_1, \ldots, x_N)$.[6] Control-flow instructions, by convention, don't have an output. Loads and stores are not considered deterministic since our current analysis does not model the contents of memory. We define $\text{out}(I) = y$ and $\text{in}(I) = \{x_1, \ldots, x_N\}$.

Deterministic instructions are of interest since knowing all but one of their operands/results enables deduction of the remaining one. We say an instruction I is *forward solvable* if, for all concrete assignments to $x_i \in \text{in}(I)$, we have a unique solution for y. We say I is *backward solvable* if for any $x_i \in \text{in}(I)$, for all concrete assignments to $\text{out}(I)$ and all $x_{i \neq j} \in \text{in}(I)$, we have a unique solution for x_i. All deterministic instructions (e.g., `add`, `sub`, `mul`) are forward solvable; not all are backwards solvable.[7] Exploiting the solvability of instructions is how we achieve propagation of knowledge through computations as motivated in Section 4.

An important consequence of working with programs expressed in SSA form is that the equations that define variables are *unique*. Crucially (and perhaps counter-intuitively) this implies the equations associated with instructions are exploitable *anywhere* in the control-flow graph, even if the associated instruction is not encountered in a given trace or is unreachable in general. When analyzing a non SSA-form program, there may be multiple definitions for any given variable, and thus you need to consider only the definitions that apply to the locale you are in. Attempting to analyze a non-SSA program while keeping track of which definitions apply where implicitly converts the program to SSA form. We will see the benefits of the global view of equations in the examples from Section 5.3.

5.2.2 ϕ-functions. We now discuss ϕ-functions. Before we begin, recall that the PC is public to the attacker due to assumptions made by the constant-time programming model (Section 3). *Since the PC is public, it can be considered known at all points in time in the non-speculative execution.*

Because the PC is known, we can treat ϕ-functions as being forward solvable. If at any point all of a ϕ-function's inputs are known, then regardless of which one gets assigned to the output, the output must be known as well. Consider a ϕ-function I_ϕ of the form $y = \phi(x_1, \ldots, x_N)$ in a block B with input edges e_i and output edges e_o. The semantics of I_ϕ are such that $y = x_i$ if e_i is taken to reach B. Now, suppose that $x_1, \ldots, x_N$, are known on some edge e. If e is in a trace containing I_ϕ, we know y because we know which x_i is assigned to y (because the PC is known) and we know each x_i. Note that if I_ϕ is *not* in a trace containing e, then knowledge of y is vacuous; it cannot be used in a meaningful way.

There is an additional, more subtle version of forward solvability when considering ϕ-functions. Consider again I_ϕ as defined before. If each input x_i to the ϕ-function is known on its respective edge e_i, y is known on all e_o. This is again due to the assumption that the PC is known; the input edge used to arrive at B is known and thus we will know which x_i is assigned to y.

Note that ϕ-functions are *not* backward solvable; knowing the output and all but one of the inputs doesn't necessarily reveal the last input. That said, there *is* a causal relationship we can exploit in the backwards direction. Using the definition/semantics of I_ϕ from before, suppose that y is known on all output edges e_o. Then every x_i is known on its respective edge e_i. The justification is as follows: suppose e_i is traversed. Then $y = x_i$ and we know for which x_i this holds. Now, we must leave B through some e_o, and y is known on every e_o. Thus, in this scenario, x_i is known.

5.2.3 Knowledge propagation theorems. We summarize the previous discussion with a series of theorems that describe relationships on knowledge. The proofs of these theorems can be found in the full version of the paper [17]. We start with theorems that describe the knowledge available to every edge in isolation,

Theorem 1. *Consider an instruction I of the form `Tr(x)` in some block B with output edges e_o. For all e_o, $x \in K_{\mathbb{E}}(e_o)$.*

Theorem 2. *Take any edge $e \in \mathbb{E}$. Consider a forward solvable instruction I from anywhere in the control-flow graph, and suppose it is of the form $y = f(x_1, \ldots, x_N)$. If $x_i \in K_{\mathbb{E}}(e)$ for all x_i, then $y \in K_{\mathbb{E}}(e)$.*

Theorem 3. *Take any edge $e \in \mathbb{E}$. Consider a backward solvable instruction I from anywhere in the control-flow graph, and suppose it is of the form $y = f(x_1, \ldots, x_N)$. For any $j \in \{1, \ldots, N\}$, suppose we have all $x_{i \neq j} \in K_{\mathbb{E}}(e)$ as well as $y \in K_{\mathbb{E}}(e)$. Then $x_j \in K_{\mathbb{E}}(e)$.*

The following theorems describe the relationship of knowledge between edges in the general case,

Theorem 4. *Consider a block B. If for some variable v we have $v \in \bigcap K_{\mathbb{E}}(e_i)$ for all realizable e_i in $E_{\text{in}}(B)$, then we have $v \in K_{\mathbb{E}}(e_o)$ for every $e_o \in E_{\text{out}}(B)$.*

Theorem 5. *Consider a block B. If for some variable v we have $v \in \bigcap K_{\mathbb{E}}(e_o)$ for all realizable e_o in $E_{\text{out}}(B)$, and if v is not defined in B, then we have $v \in K_{\mathbb{E}}(e_i)$ for every $e_i \in E_{\text{in}}(B)$.*

An important thing to notice about Theorem 5 is that we mandate v is not defined in B. This is because if v is *not* defined in B, then nothing in B can change v's status in terms of knowledge; this is not true if v is defined in B. That said, the theorem does *not* prevent variables from being known prior to their definition; they just need to be inferable (via the other theorems).

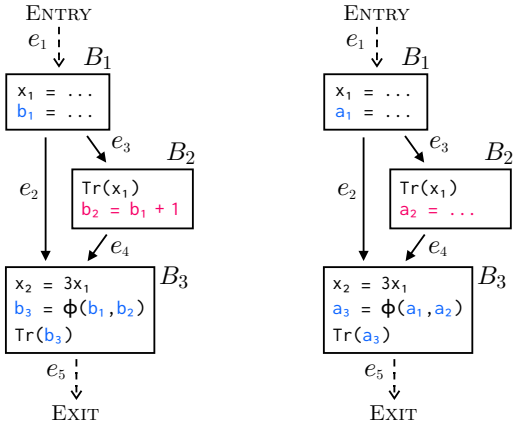
The following theorems describe the relationship of knowledge between edges in the special case that they are connected via a ϕ-function.

Theorem 6. *Let I_ϕ denote a ϕ-function of the form $y = \phi(x_1, \ldots, x_N)$ in block B with N input edges $e_1, \ldots, e_N$. The semantics of I_ϕ are such that $y = x_i$ if $e_i \in E_{\text{in}}(B)$ is traversed to reach B. If for all realizable e_i we have $x_i \in K_{\mathbb{E}}(e_i)$, then for every output edge e', $y \in K_{\mathbb{E}}(e')$.*

Theorem 7. *Let I_ϕ denote a ϕ-function of the form $y = \phi(x_1, \ldots, x_N)$ in a block B with input edges $e_1, \ldots, e_N$. The semantics of I_ϕ are the same as in Theorem 6. If for all realizable output edges e' we have $y \in K_{\mathbb{E}}(e')$, then for every e_i we have $x_i \in K_{\mathbb{E}}(e_i)$.*

[6] This can be generalized to instructions with multiple outputs by writing down a separate equation for each output.

[7] For example, $y = x_1 \times x_2$ is deterministic and forward solvable, but not backwards solvable; if one operand is 0, the output is 0 regardless of the other operand's value.



(a) Example 1 (b) Example 2

Figure 3: Examples used to illustrate knowledge over edges ($K_{\mathbb{E}}$).

Note that for Theorems 1–3, the realizability of an edge is not important to our model of knowledge. This is because the claims are about the *local* knowledge associated with each edge individually, and unrealizable edges only contain vacuous knowledge. On the other hand, Theorems 4–7 make use of the realizability of edges since we are considering the relationships *between* edges.

5.3 Examples Computing Knowledge/Frontiers

Figure 3 shows an example of how to compute knowledge (and then the knowledge frontier) with the relations from Section 5.2.

First consider Figure 3a. There are two paths through the control-flow graph; we'll assume they both correspond to realizable traces. Starting off, we have `Tr(b3)` in B_3 which leads to knowing b_3 on edge e_5 (Theorem 1) and subsequently b_1 on e_2 and b_2 on e_4 (Theorem 7). The equation $b_2 = b_1 + 1$ and Theorem 2 enable us to deduce b_2 everywhere b_1 is known (and vice versa by Theorem 3)—similarly for x_1, x_2 and $x_2 = 3x_1$. Since both b_1 and b_2 are known on e_2, e_3, and e_4, by Theorem 2, b_3 is also known on e_2, e_3, and e_4.

By continuing to apply the theorems, we get $K_{\mathbb{E}}(e_1) = \emptyset$, $K_{\mathbb{E}}(e_2) = \{b_1, b_2, b_3\}$, $K_{\mathbb{E}}(e_3) = K_{\mathbb{E}}(e_4) = \{b_1, b_2, b_3, x_1, x_2\}$, and $K_{\mathbb{E}}(e_5) = \{b_1, b_2, b_3\}$.

This provides a basis for the frontier of x_1 and x_2 to be $\{B_2\}$. More importantly, it means the frontier for b_1, b_2, b_3 is $\{B_1\}$. A program that non-speculatively enters B_1 will inherently transmit all three. (We detail more precisely how to compute the frontier given $K_{\mathbb{E}}$ in Section 6.3.) This will enable more efficient protection: to enforce Property 1, it is sufficient to add `protect` statements for b_1, b_2, b_3 solely in B_1.

Next consider Figure 3b, which is almost the same as Figure 3a except for two changes: first, every b_i is replaced with a_i for clarity; second (and most importantly), the equations for a_2 and b_2 differ. We use a_2 = ... to represent a non-deterministic instruction. Since we no longer have an equation relating a_2 to a_1 (as we did with $b_2 = b_1 + 1$ from before), the knowledge settles to $K_{\mathbb{E}}(e_1) = \emptyset$, $K_{\mathbb{E}}(e_2) = \{a_1\}$, $K_{\mathbb{E}}(e_3) = K_{\mathbb{E}}(e_4) = \{x_1, x_2, a_2\}$, and $K_{\mathbb{E}}(e_5) = \{a_3\}$. From this we can deduce that the frontier for a_1 is $\emptyset$; the frontier for x_1, x_2 and a_2 is $\{B_2\}$; the frontier for a_3 is $\{B_3\}$. This matches our security goal: it is unsafe to hoist `protect` statements above any variable's definition because knowing a_1 is not the same as knowing a_2 and hoisting a `protect` would enable an attacker to selectively learn both through `Tr(a3)`.

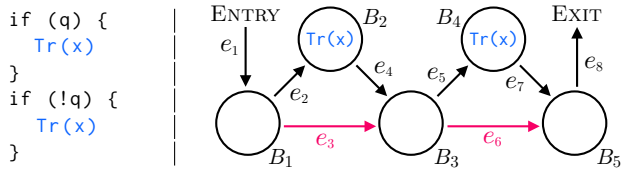


Figure 4: A program and its control-flow graph. The branches are anti-correlated, thus the trace $\{e_1, e_3, e_6, e_8\}$ is not realizable.

5.4 Approximating Non-Speculative Knowledge

Precisely computing $K_{\mathbb{E}}$ using the theorems from the previous section is generally intractable since it relies on knowing whether any given edge is realizable. We can instead attempt to compute an *approximation* of $K_{\mathbb{E}}$, denoted as $\widehat{K}_{\mathbb{E}}$. We consider our approximation *sound* if it does not over-estimate an attacker's true knowledge; we consider it *imprecise* if it under-estimates it. To compute $\widehat{K}_{\mathbb{E}}$, we make the assumption that any path through the control-flow graph corresponds to a realizable trace. More specifically, for any edge $e = (B_i, B_j)$, we assume that any edge $e' \in E_{\text{in}}(B_i)$ may have been traversed prior to it, and any edge $e'' \in E_{\text{out}}(B_j)$ may be traversed after it. Looking back at Definition 4, by increasing the number of traces we consider e to have been part of, we are (potentially) reducing the size of $\widehat{K}_{\mathbb{E}}(e)$; i.e. $\widehat{K}_{\mathbb{E}}(e) \subseteq K_{\mathbb{E}}(e)$. Thus, this approximation method (potentially) loses precision, but it maintains soundness.

Example. Look at Figure 4. The only possible traces are $t_1 = \{e_1, e_2, e_4, e_6, e_8\}$ and $t_2 = \{e_1, e_3, e_5, e_7, e_8\}$. (We've colored edges e_3 and e_6 to correspond with the figure.) The paths $\{e_1, e_3, e_6, e_8\}$ and $\{e_1, e_2, e_4, e_5, e_7, e_8\}$ do not correspond to realizable traces since they imply `q` is both true and false. Ideally then, we must have that x is known on e_3, i.e. $K_{\mathbb{E}}(e_3) = \{x\}$, since at this point the execution *must* take e_5 and encounter `Tr(x)`. Making the simplifying assumption (above) to compute $\widehat{K}_{\mathbb{E}}(e)$, however, we cannot assume whether we will take e_2 vs. e_3; nor can we assume that we will take e_5 vs. e_6 (for whichever of e_2 or e_3 we took). In other words, the analysis cannot conclude whether we will encounter `Tr(x)` and thus deduces that x is *not* known on e_3; i.e. $\widehat{K}_{\mathbb{E}}(e_3) = \emptyset$. See that $\widehat{K}_{\mathbb{E}}(e_3) \subset K_{\mathbb{E}}(e_3)$; we've lost precision in order to gain tractability, but we have not sacrificed soundness.

6 DECLASSIFLOW APPROACHES

In this section, we describe the approach used by DECLASSIFLOW to compute and utilize non-speculative attacker knowledge.

At a high level, our analysis first computes $\widehat{K}_{\mathbb{E}}(e)$ for efficiency reasons (Section 6.1), but invokes more sophisticated analyses to compute $K_{\mathbb{E}}(e)$ on specific edges when doing so is deemed profitable (Section 6.2). Later subsections then detail how to use the edge-based knowledge to compute the knowledge frontier (Section 6.3) and instrument protection (Section 6.4). Section 7 goes over lower-level details of all of the above.

6.1 Computing Knowledge via a Data-Flow Analysis

Our ideas from Section 5 map naturally to a *data-flow analysis* [3].

Data-flow analyses work by assigning *data-flow values* to every point in the control-flow graph. They iteratively apply local rules known as the *data-flow equations* to build up and combine these data-flow values. When this process has converged (i.e. the size of the data-flow values stagnates), what remains is a *data-flow solution*.

A data-flow analysis makes the assumption that any path through the control-flow graph is a potential path the execution may take; it approximates $\mathbb{T}$ with $\mathbb{P}$. Since the definition of $\widehat{K}_{\mathbb{E}}$ relies on the same assumption, we can formulate a data-flow analysis such that the solution is *exactly* $\widehat{K}_{\mathbb{E}}$. The specific data-flow equations we use are discussed in Section 7.1. By construction, the result of our ideas applied to Figure 4 (as discussed in Section 5.4) is precisely what our data-flow analysis would yield.

6.2 Improving Precision via Symbolic Execution

There are of course drawbacks to approximating $K_{\mathbb{E}}$ as done by the data-flow analysis (see Section 5.4). Namely, under-estimating an attacker's knowledge causes us to over-estimate the amount of protection needed. To avoid this, we need a way to analyze programs in a way that can deduce whether a trace is unrealizable.

To that end, we also selectively employ symbolic execution [5]. In a nutshell, symbolic execution involves executing a program with "symbolic" values; variables are mapped to symbolic expressions rather than concrete values (at least, in the cases when the concrete value cannot be deduced unambiguously). Expressions associated with all variables are derived by the instructions encountered during the symbolic execution. Upon reaching a branch, a symbolic execution engine will traverse both paths separately. Every path through will have associated with it "path constraints", which are a set of symbolic expressions implied by the set of taken branches.

Our framework uses symbolic execution to answer the following question,

Question 1. *Given some region R of the control-flow graph and given some variable x, does there exist a path through R (that corresponds to a realizable trace) upon which x is* not *transmitted?*

Recall that a data-flow analysis conservatively answers this question by assuming that if a path exists, it is part of a realizable trace. By considering the semantics of the instructions and the branch conditions, symbolic execution can try and answer the question less conservatively; though we stress that it does so in a sound manner since it needs to *prove* that the path cannot be taken.

We can look back at Figure 4 to see how symbolic execution can succeed where the data-flow analysis fails. The candidate region we consider is the entirety of the program. Recall that the problematic path was $p' = \{e_1, e_3, e_6, e_8\}$, which again does *not* correspond to a realizable trace. When the tool considers the path p', the path constraints will contain *both* q = false and $q \neq$ false. These contradictory statements mean that no non-speculative execution could ever traverse such a path. Thus, symbolic execution will conclude that `Tr(x)` is unavoidable (i.e. that the answer to Question 1 is "no" for this region and variable x) meaning x is guaranteed knowledge at (among other places) the program's entry point. Thus, we have achieved a more precise result than the data-flow analysis.

The details of how and when we utilize symbolic execution to answer Question 1 are given in Section 7.2.

Note that the symbolic execution cannot be used on its own; the results of the data-flow analysis are a prerequisite. For symbolic execution to work, we will need to instrument the code to indicate what variables are known at various points, and the data-flow analysis is precisely what provides this information. While it is certainly possible to formulate the entire analysis in terms of symbolic execution, this would certainly not scale to larger programs as well as a data-flow analysis would.

6.3 Computing the Knowledge Frontier

As discussed in Section 4, Property 1 is the key to finding a minimal yet sufficient set of `protect` statements needed to secure a program. To that end, we need to compute the knowledge frontier given the results of the data-flow analysis and/or symbolic execution.

The first step to computing the knowledge frontier is to map knowledge from edges to basic blocks; we want the map $\widehat{K}_{\mathbb{B}} : \mathbb{B} \rightarrow 2^{\mathbb{V}}$. For any block B, we define $\widehat{K}_{\mathbb{B}}(B) = \bigcap \widehat{K}_{\mathbb{E}}(e')$ for all $e' \in E_{\text{out}}(B)$. Recall that $\widehat{K}_{\mathbb{E}}$ is the result of the data-flow analysis. The results from the symbolic execution constitute additions to $\widehat{K}_{\mathbb{B}}$. Question 1 is associated with some candidate variable x and some candidate region R of the control-flow graph. If, when given these, the symbolic execution tool answers "no" to Question 1, then $x \in \widehat{K}_{\mathbb{B}}(B)$ for all $B \in R$.

With $\widehat{K}_{\mathbb{B}}$ in hand, we can compute the frontiers for all variables. For any variable x, we first over-estimate $\mathcal{F}(x)$ by adding all $B \in \mathbb{B}$ such that $x \in \widehat{K}_{\mathbb{B}}(B)$. Then, for any $B \in \mathcal{F}(x)$, if x is known in all of B's predecessor blocks, we remove B from $\mathcal{F}(x)$. This is done using another data-flow analysis. After this, $\mathcal{F}(x)$ represents the precise knowledge frontier for x.

Consider an arbitrary program function `f`. For any variable x, if its frontier $\mathcal{F}(x)$ is the entry block of `f`, we say x is *fully declassified*. If all variables transmitted by `f` are fully declassified, then `f` itself is fully declassified.

6.4 Adding Protection

Once we have the knowledge frontier $\mathcal{F}$, we can place protection using a simple strategy: for every variable x that is transmitted, we place `protect(x)` statements all along its frontier $\mathcal{F}(x)$. We refer to this approach as "enforcing the knowledge frontier". This is a straightforward method to satisfy Property 1. In terms of hoisting `protect` statements as high as possible, it is also optimal.

For simplicity, we perform our analysis at the granularity of functions. To that end, we now discuss two strategies—*callee enforcement* and *caller enforcement*—that an analysis can use to instrument protection when considering calls between functions. We use both of these in our final implementation.

Callee enforcement. Callee enforcement is a straightforward but potentially high overhead strategy. Suppose we have a program function `f`, which calls `g`. With callee enforcement, `f` and `g` are analyzed and instrumented with protections in isolation. That is, `g`

(the callee) is protected regardless of knowledge in the caller context f. This approach allows us to ignore the interactions between functions and have every function focus on enforcing its own frontier in isolation. While simple, this strategy may overprotect the callee. For example, all variables in g that require protection may be known before g is called.

Caller enforcement. To reduce overhead stemming from callee protection, we now consider an alternative approach called caller enforcement. The high-level idea is that, if certain conditions about the callee are met, the caller can abstractly view the callee as any other transmitter. We call these *pseudo transmitters*: program-level functions that leak (a subset of) their arguments but no internal data. More precisely, consider a function $f(x_1, \ldots, x_N)$ that non-speculatively leaks some non-empty subset of its arguments $x'_1, \ldots, x'_M$ (with $M \leq N$) (possibly) along with some other variables $v_1, \ldots, v_K$.

The function f is a *pseudo transmitter* if and only if: (1) $x'_1, \ldots, x'_M$ and $v_1, \ldots, v_K$ are all fully declassified in f; (2) knowledge of $x'_1, \ldots, x'_M$ is sufficient to infer every v_j; (3) all other functions called by f are themselves pseudo transmitters. The key insight is that the information leaked by a pseudo transmitter can be understood strictly in terms of its arguments, which means we can reason about its protection in its calling contexts.[8] Rather than protecting every function call, we can enforce the frontier of the *arguments* that are (non-speculatively) leaked by each function call. If calls to the same function have arguments that are connected via data flow, we can exploit this to protect their common frontier.

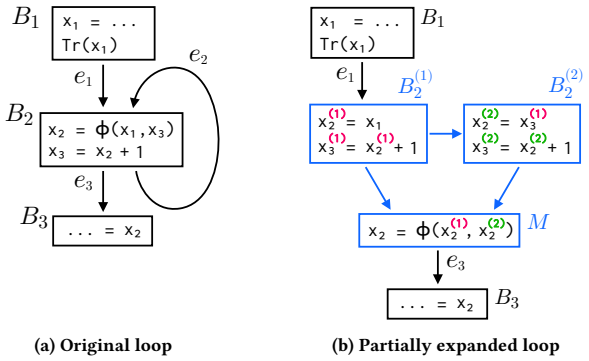
To see the benefits of caller enforcement, consider the following example. Suppose some program function f makes two calls to another function g(x) that is a pseudo transmitter which leaks its argument x and no internal variables. Suppose the first call to g is $g_1(x)$ and the second call is $g_2(x')$ where $x' = x+1$ (the subscripts are used to distinguish the calls). Suppose further that g_1 dominates g_2. Both calls leak their arguments, so naively, the frontier for x is the calling context of g_1 while the frontier of x' is the calling context of g_2. However, since x and x' are equivalent in terms of knowledge (due to the backward solvability of $x' = x + 1$), and since g_1 dominates g_2, we can promote the frontier of x' to that of x. Enforcing the frontier of x is sufficient to protect the call to $g_2(x')$. Thus one `protect` can cover two function calls. This is strictly better than callee enforcement which would have protected g_1 and g_2 separately. Note that being a pseudo transmitter is both sufficient and necessary for a function to be caller enforced.

7 IMPLEMENTATION DETAILS

We now give details for how the ideas from Sections 5 and 6 are implemented.

DECLASSIFLOW's main components follow the sketch given in Section 6, and we adopt the following strategy for combining the data-flow analysis, symbolic execution and protection steps together. First, we apply a data-flow analysis (Section 7.1). Second, if the data-flow analysis results are suboptimal (i.e. we have functions that are not fully declassified), we analyze those functions using the symbolic execution tool KLEE (Section 7.2). After running one or both analyses, we place protections (Section 7.3). We apply all 3 passes in that order to every function.

[8] We do not have this luxury with functions that are not pseudo transmitters even if they are fully declassified. Their internal leakages mandate that every call to the function creates its own personal frontier.



(a) Original loop (b) Partially expanded loop

Figure 5: On the left is a loop which creates an inductive relationship between x_1 and x_2. Partially expanding it as shown on the right allows our data-flow analysis to capture this relationship.

Our framework contains both intra- and inter-procedural aspects. By design, KLEE is both intra- and inter-procedural.[9] Thus, our symbolic execution pass adopts both these traits. The data-flow analysis and protection pass are primarily intra-procedural but are performed on functions in a specific order so as to pass non-speculative knowledge computed in a callee up the call graph. Specifically, we apply the analysis in the order of callees then callers, i.e., work up the call graph starting at the leaves.[10]

7.1 The Data-Flow Pass

As mentioned in Section 6.1, we can formulate a data-flow analysis that computes $\widehat{K}_{\mathbb{E}}$. We use control-flow edges as the program points of interest, and the data-flow value for a given edge e is precisely $\widehat{K}_{\mathbb{E}}(e)$. The data-flow rules are based on the theorems from Section 5.2.

Running the data-flow analysis is done in two steps. First, we perform an LLVM control-flow-graph-level transformation to ensure that loops are correctly modeled. Second, we initialize data-flow values along all program edges and iteratively apply the data-flow rules until $\widehat{K}_{\mathbb{E}}(e)$ is constructed.

Loop transformation. In the presence of loops, the data-flow rules can fail to capture inductive relationships (e.g., loop-carry dependencies). For example, in Figure 5a, they would be unable to deduce that knowledge of x_1 in B_1 implies knowledge of x_2 in B_3. To remedy this, prior to the data-flow analysis, we perform *partial loop expansion*. This procedure takes a control-flow graph with a loop and transforms it to be acyclic. We accomplish this by duplicating the loop body and removing the back edge. Unlike full loop unrolling, we only keep two cases: the initial case and the inductive case. We note that although this procedure destroys the "correctness" of the program in terms of the values computed, it

[9] That is, KLEE will symbolically execute the top level function we provide it as well as any callees, and so on recursively.

[10] This approach assumes that the call-graph is acyclic, i.e. there is no recursion. In cases where this does not hold, we can extend our method to repeatedly analyze functions and stop when no new information is derived. However, since we do not encounter recursion in our benchmarks, we omit implementation and further discussion.

preserves the relationships (between variables) that are relevant to modeling knowledge. This technique is crucial for analyzing the benchmarks in Section 8. We present more details about the partial loop expansion procedure in the full version [17].

Data-flow initialization and evaluation. Once the program control-flow graph is transformed to account for loops, we proceed to run the data-flow analysis.

We start by initializing data-flow values. We look at all the transmitters in the program. For all $B \in \mathbb{B}$, and for every $e' \in E_{\text{out}}(B)$, we initialize $\widehat{K}_{\mathbb{E}}(e')$ to be the set of all variables in B that are directly passed to a transmitter. This is a direct application of Theorem 1. We also utilize some inter-procedural initialization, but with a unidirectional flow of information.[11] Suppose we are analyzing a program function `f` that makes a call to some other function `g(x)` in block B. If previous analysis of `g` had concluded that the frontier of `x` within `g` was the entry block of `g`, then in our analysis of `f` we may add x to $\widehat{K}_{\mathbb{E}}(e)$ for all $e \in E_{\text{out}}(B)$. This step relies on `g` having been analyzed prior to `f`, which is why we must perform our analysis in the order of callees then callers as mentioned before.

We then apply Theorems 2-7 iteratively until we have achieved convergence; that is, until the size of $\widehat{K}_{\mathbb{E}}$ has reached a fixed point. Recall that the range of of our data-flow values is $2^{\mathbb{V}}$, which is a powerset lattice of finite height. Our data-flow rules never decrease the size of the data-flow value, i.e. the values can never move down the lattice. Thus, there is a limit to how much the data-flow values can grow before stagnating. This guarantees that our analysis will converge in finite time.

7.2 The Symbolic Execution Pass

We apply the symbolic execution pass on functions that are not fully declassified; our goal is to deduce additional attacker knowledge as detailed in Section 6.2. To that end, we invoke Klee [13] to answer Question 1 with respect to a region R and variable x. We are interested in analyzing regions that encompass all transmitters. Let $\text{Tx}(\mathbb{B})$ be the set of all blocks which contain at least one transmitter. Any region R such that $\text{Tx}(\mathbb{B}) \subseteq R$ is a candidate region for the symbolic execution pass. We can find these automatically by looking at all $B \in \mathbb{B}$ and keeping the ones that collectively dominate every transmitter. Then, the regions defined by each of these is a candidate region. Furthermore, any variable which is known in a block that contains a transmitter (even if that variable is itself not transmitted in that block) is a candidate variable.[12] That is, the set of all candidate variables is $\bigcup \widehat{K}_{\mathbb{E}}(B')$ for all $B' \in \text{Tx}(\mathbb{B})$.

We need to run Klee separately for every pair of candidate region and candidate variable. Thus, without loss of generality, we assume for the remainder of this discussion that R is the candidate region and x is the candidate variable. By definition, there is a unique block in R, which we'll denote as B_R, that dominates all other blocks R.

Suppose we are analyzing a program function `f`. For ease of instrumenting the analysis (below), we assume it has a single terminating block. If it does not, we modify the function such that it does by creating a new terminating block and redirecting all previous ones to it. To run Klee, we write a wrapper which serves as `main()` and calls `f` with symbolic arguments. Klee provides an interface for specifying constraints on the values arguments can take. Setting these constraints correctly is important. For example, if `b` in `f(int* a, int b)` denotes the length of an array pointed to by `a`, one should constrain `b` to be non-negative. Automating deriving argument semantics for this step would be useful, but we consider it out of scope for this paper.

"Asking" Klee if the answer to Question 1 is "yes" or "no" is done by using an `assert(...)` statement. The assertion is that the answer to Question 1 is "no": the paths upon which x is not transmitted (if any) are not realizable. To accomplish this, we instrument the program with flags. Let `L` denote the flag variable. For simplicity, assume `L` is not SSA, i.e., can be assigned more than once. We add the initialization `L = 0` in the entry block of the function `f`. We add the assignment `L = -1` in B_R. In every $B' \in \text{Tx}(\mathbb{B})$, we add the assignment `L = 1`. In the unique terminating block of the function, we add `assert(L ≠ -1)`. If Klee manages to find a counterexample to this assumption, then there is some path through R such that x is not transmitted. Thus, the answer to Question 1 is "yes". On the other hand, if Klee deduces that the assertion is provably true, then every realizable trace either circumvents the region R, or enters it and necessarily transmits x, rendering it known. Thus, x is known throughout R.[13]

7.3 The Protection Pass

We enforce Property 1 with speculation barriers, which we denote as `SPEC_BARR`. These barriers delay the execution of younger instructions until they are non-speculative. `SPEC_BARR` conceptually implements `protect(*)`; it indiscriminately applies protection for every variable. That means if the knowledge frontiers for two variables x and y both include some block B, only one `SPEC_BARR` needs to be added to B to enforce the frontier for both x and y. Since `SPEC_BARR` itself is relatively heavyweight, our main tactic to get speedup will be to determine that frontiers (and therefore `SPEC_BARR` placement) fall outside of critical loops. In that case, for sufficiently long loops, the cost of the `SPEC_BARR` will be amortized.

We now describe the procedure for placing `SPEC_BARR` for a program function `f`. We first clone `f` to produce `f'`. The latter is what we refer to as the *protected* version of `f`. Without loss of generality, suppose that `f` (and thus `f'`) only makes calls to one other function `g(x)` which leaks its argument. Assume a protected version of `g` exists, denoted `g'(x)`.

We now describe the procedure to protect the internals of `f'`. We compute the joint frontier of locally transmitted variables, i.e. $F_{\text{local}} = \bigcup \mathcal{F}(v)$ for all `v` in `f`.

We next need to get the joint frontier of all variables that are not necessarily locally transmitted, but are transmitted by calls to other

[11] In theory, there is benefit to augmenting the analysis to have callers send *and* receive information to/from callees. However, it complicates the analysis, and these opportunities don't arise in our benchmarks. Thus, we leave this for future work.

[12] If the data-flow analysis has deduced a candidate variable to be known throughout the candidate region, we don't need to run Klee. Note that we do not implement this optimization in our evaluation.

[13] One might worry that this method does not check whether the former is always true, i.e. whether R is even ever entered. We do not need to do so since if R is never entered, knowledge of x in R is vacuous and thus inactionable.

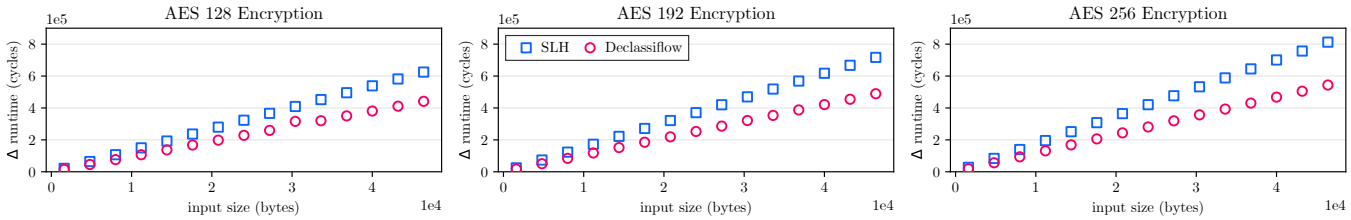


Figure 6: The results of running AES encryption with three different key sizes on inputs of various lengths. We show the raw difference between the runtime (in cycles) of the functions compiled with SLH and the runtime of the baseline (insecure) code. We show the same difference for the DECLASSIFLOW protected versions of the functions, i.e., after our analysis is applied. The overhead reduction from the SLH-enabled code to our protected code grows with the size of the input.

functions; we denote these as F_{func}. In this example, this would be from calls to g. We can replace calls to `g(x)` with calls to `g'(x)`.

The manner in which we enforce protection of `g'` depends on whether it is a pseudo transmitter or not. Suppose it is a pseudo transmitter and thus can be *caller* enforced. For purposes of disambiguation, let $\texttt{x}_\texttt{i}$ denote the argument to the i-th call site of `g'(x)`. We need to enforce the union of the frontiers for the leaked $\texttt{x}_\texttt{i}$. We compute $F_{\text{func}} = \bigcup \mathcal{F}(x_i)$.[14] If `g'` is *not* a pseudo transmitter, it will be *callee* enforced, and so it will be internally protected, and we would have $F_{\text{func}} = \emptyset$.

We now place SPEC_BARR's in `f'`. If `f'` itself is *not* a pseudo transmitter, then it must be *callee* enforced. In this case, a SPEC_BARR is placed in every block in $F_{\text{local}} \cup F_{\text{func}}$. However, if `f'` is a pseudo transmitter, and thus can be *caller* enforced, then a SPEC_BARR needs to placed in all those same blocks *except* the entry block of the function.[15] Note, if `f'` is a pseudo transmitter, then $F_{\text{local}} \cup F_{\text{func}}$ is precisely the entry block of `f'`. As a result, *no* SPEC_BARR's will be placed in `f'`. This leads to a useful optimization: if we have a chain of nested calls of pseudo transmitters, we need only a single SPEC_BARR at the top level to protect the full call chain.

This unidirectional inter-procedural strategy is a fairly greedy method for minimizing the total number of SPEC_BARR's in the program. Taking a global view of the relationship between functions will undoubtedly lead to better barrier placement strategies. However, estimating the cost of barriers will most likely rely on heuristics, and so we leave such a problem to future work.

8 EVALUATION

We implement our analysis as an LLVM pass that interoperates with KLEE. We will now evaluate the analysis' effectiveness in terms of how it can efficiently protect constant-time workloads.

Workloads. We evaluate 3 constant-time workloads: ① The AES block-cipher encryption from the `ctaes` repository under the Bitcoin Code organization found on Github [11]. ② The Djbsort constant-time[16] integer sorting algorithm [9, 21]. ③ The ChaCha20 stream cipher encryption from the BearSSL library [1]. For each, we applied DECLASSIFLOW as described in Sections 6 and 7.

[14] We can generalize our discussion to multiple functions being called with multiple arguments by expanding this term. If there are other functions called by `f'` that require caller enforcement, we can add the union the frontiers of their leaked arguments to this term. If any of the pseudo transmitters called leak multiple arguments, again, the union of the frontiers of those leaked arguments is added to this term.

[15] This is because the entry of `f'` will be protected at the call site. Note that if `f'` is a top-level function and has no callers, we must place the SPEC_BARR in the entry block.

[16] It is constant-time with respect to the *values* within the array, *not* the array size.

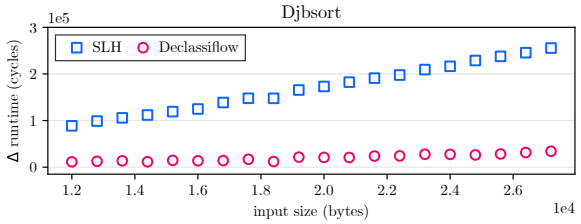


Figure 7: The results of running Djbsort on inputs of various lengths. The y-axis follows the same convention as in Figure 6.

Baselines. We compare against each benchmark unmodified and also to each benchmark compiled with Speculative Load Hardening (SLH) [23]. SLH is a Spectre mitigation deployed by LLVM that works by accumulating branch predicate state and using that state to prevent certain instructions from executing speculatively and/or to prevent certain data from being forwarded speculatively. The academic community has shown how SLH, when configured to delay the speculative execution of all transmitters, is sufficient to protect non-speculatively accessed data [36, 48]. We compare to a weaker variant, called "address SLH" or aSLH. aSLH was designed to protect speculatively-accessed data. It does this by delaying the execution of speculative loads, that have addresses only known at runtime, until they are non-speculative. That is, it considers loads to be instructions that access (return) sensitive data. Since loads are also transmitters, aSLH can be viewed as implementing a subset of the mechanisms required to protect non-speculatively accessed data. Thus, the security provided by DECLASSIFLOW is stronger than aSLH and aSLH's overhead will underestimate the true overhead of SLH in our setting.

All workloads compiled using DECLASSIFLOW maintain the branch predicate state (but do not use it to delay instructions/dataflows) needed to support SLH. We do this to provide a conservative overhead estimate: SLH requires this information be maintained across function calls, thus the benchmarks we protect with DECLASSIFLOW can interoperate with SLH implemented in a calling context, if needed.

Environment setup. We run our experiments on an x86-64 Intel Xeon Gold 6148 machine with Ubuntu 20.04, kernel version 5.4.0-146-generic. We compiled our benchmarks with `-O3` and ensured that all versions of the binary differ only in their protection mechanisms. The AES source files were compiled with LLVM version 16 while the Djbsort and ChaCha20 source files were compiled with

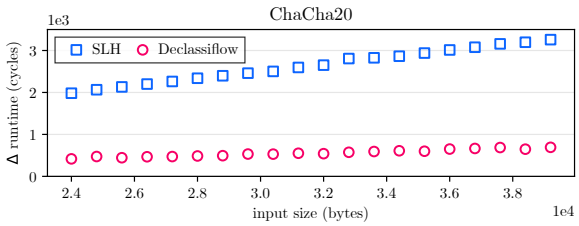


Figure 8: The results of running ChaCha20 on inputs of various lengths. The y-axis follows the same convention as in Figure 6.

LLVM version 11.[17] We run KLEE in a Docker container based on their official image.[18]

Analysis procedure. To produce protected code, we run the 3 phases of our analysis: (1) The data-flow analysis is run to determine $\widehat{K}_{\mathbb{E}}$. (2) KLEE is used to compute additions to $\widehat{K}_{\mathbb{E}}$. (3) The protection pass is run, which adds barriers. If KLEE is not needed to improve the precision of the result of phase (1), phase (2) may be skipped. For the benchmarks which do go through phase (2), we provide a static buffer of fixed size that contains symbolic values. We also provide a symbolic value that represents the buffer's *dynamic* length. We constrain this symbolic value to be as low as 0 and as high as the length of the static buffer. This static buffer coupled with the length represent the user input to the functions.

Benchmarking methodology. For constant-time AES encryption,[19] we benchmark the top-level functions `AES128_encrypt`, `AES192_encrypt`, and `AES256_encrypt`, which use key sizes of 128, 192, and 256 bits respectively. We benchmark each of these with inputs of various sizes. The Djbsort and ChaCha20 benchmarks only have one function each that performs the core workload. Thus, we benchmark those two functions, again with inputs of various sizes. Before every call to the function under test, we perform a small series of floating point computations which get their input from disparate locations in memory. The function under test is then only executed if the result is non-zero. This is meant to introduce a branch that is always taken and will be easily predicted as such. The induced speculation is meant to test the effects of the `SPEC_BARR`'s we place. We use `rdtscp` to measure timing. To amortize timer function overhead, we time the function calls in groups of 8 and then normalize the result. This batch-of-8 timing represents one "trial". For every data point, we run 800 trials and discard the first 100 to remove warmup effects of the cache and TLB. We then report the median of the 700 remaining trials.

8.1 Main Result

The full experimental results are presented in Figures 6, 7, and 8. We show the raw difference between the SLH and the DECLASSIFLOW versions of the functions with respect to the baseline. The geometric means of the relative overheads of SLH for the encryption functions are 16%, 16%, and 17% for each key size respectively. Meanwhile, the geometric means of the relative overheads of the DECLASSIFLOW versions of the functions are 12%, 12%, and 11% respectively. For the Djbsort benchmark, the geometric mean of the relative overhead of SLH is 24%, while the geometric mean of the relative overhead of the DECLASSIFLOW version is 3%. For the ChaCha20 benchmark, the geometric mean of the relative overhead of SLH is 7%, while the geometric mean of the relative overhead of the DECLASSIFLOW version is 1%.

All three benchmarks make heavy use of loops. Thus, the overhead of SLH increases with the size of the input because the SLH instrumentation is repeatedly encountered. Our analysis tries to prove that the knowledge frontier is outside of the inner loops (ideally, outside of all loops), hence decreasing the frequency that protection mechanisms are encountered. With AES, our analysis discovers that the frontier is outside of just the innermost loops. Thus, while the overhead of the DECLASSIFLOW versions grow with input size, they grow slower than that of the SLH-protected version. With Djbsort and ChaCha20, our analysis discovers the frontier is completely outside the main loop. Thus, the overhead of the DECLASSIFLOW versions are low and constant. In the DECLASSIFLOW version of each of the three benchmarks, only a *single* `SPEC_BARR` is statically inserted; though for AES the barrier is placed inside a loop, so it is encountered multiple times dynamically.

Analysis runtime. We now provide details on our analysis' runtime. Before we start, note that our analysis is *not* part of the typical code-compile-debug workflow programmers use during development. Instead, we expect it to be run a few times (e.g., once) as a post-processing step to add security on top of otherwise production-ready code. Thus, these overheads may amortize in practice.

The runtimes of all phases except (2) are given in Table 1. The runtime of phase (2) (running KLEE) is over two orders of magnitude higher than the runtimes of the other phases, making it the bottleneck. Table 1 also shows how the execution time of KLEE scales with respect to the complexity of the program. In this experiment, we vary the length of the static buffer mentioned previously, and along with it the constraints on the symbolic length. This essentially changes the *upper bound* on the size of the arrays that KLEE needs to reason about.

To understand better why KLEE runtime scales with buffer size, we now report how buffer size impacts KLEE's execution time. Recall from Sections 6.2 and 7.2 that we need to invoke KLEE for every candidate region R and candidate variable x for which we want to answer Question 1. For Djbsort, there are 72 possible candidate variables and 4 candidate regions; thus, we need $72 \times 4 = 288$ invocations of KLEE. For ChaCha20, there are 91 candidate variables and 3 candidate regions; thus we need $91 \times 3 = 273$ invocations of KLEE. For both benchmarks, we found that the number of invocations does not vary with static buffer size. Each invocation evaluates KLEE expressions, which generate what KLEE calls "query constructs."[20] For a given static buffer size, the number of query constructs generated is the same across all invocations. That said, the number of query constructs per invocation increases with the static buffer size. This is predictive of overhead; Figure 9 shows how the time taken for an individual invocation scales closely with the number of query constructs generated.

[17] These benchmarks needed to be run with KLEE, the Docker image for which uses LLVM version 11.

[18] `https://hub.docker.com/r/klee/klee/`

[19] We do not benchmark decryption since the structure (and thus the performance) is nearly identical to encryption.

[20] A query construct is a node in the tree created by a KLEE expression. See https://www.mail-archive.com/klee-dev@imperial.ac.uk/msg02341.html.

	DFA	Symbolic Execution				Protection
		$N = 4$	$N = 8$	$N = 16$	$N = 32$	
AES	1s	–	–	–	–	< 1s
Djbsort	35s	37m	48m	92m	266m	< 1s
ChaCha20	3s	35m	38m	44m	57m	< 1s

Table 1: The runtime of the data-flow analysis (DFA) phase; the protection phase; the symbolic execution phase (end-to-end runtime for various sizes for the symbolic input buffer). Since AES doesn't use KLEE, the data is omitted. Note the difference in units in the KLEE column versus the DFA and relaxation column. The numbers are the average of 5 runs rounded to the nearest unit.

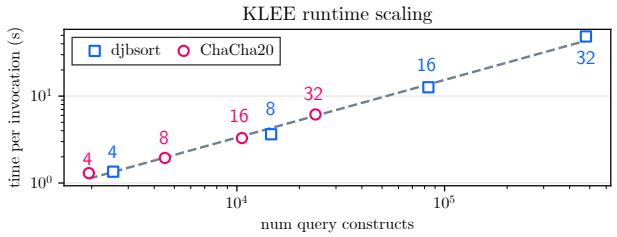
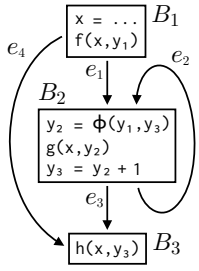


Figure 9: The time taken per invocation vs. the number of query constructs per invocation. Note, for any particular symbolic buffer size, every invocation of KLEE will have the same number of query constructs. We plot the median of the time measurements of those invocations. The size of the symbolic buffer associated with any particular datapoint (chosen to match Table 1) is indicated next to it. Note that in Table 1, we are reporting the end-to-end runtime (sum of) for all invocations.

```
encrypt(y1) {
  x = ...
  f(x,y1)
  for (...) {
    y2 = ϕ(y1,y3)
    g(x,y2)
    y3 = y2 + 1
  }
  h(x,y3)
}
```

(a) Original



(b) CFG

```
encrypt(y1) {
  SPEC_BARR
  x = ...
  f'(x,y1)
  for (...) {
    y2 = ϕ(y1,y3)
    g'(x,y2)
    y3 = y2 + 1
  }
  h'(x,y3)
}
```

(c) DECLASSIFLOW

Figure 10: Application of our analysis to `AES_encrypt`. Functions `f`, `g`, and `h` are pseudo transmitters that leak both their arguments. `f'`, `g'` and `h'` are their DECLASSIFLOW protected counterparts. NOTE: The figure depicts a *highly simplified* version of the function that still captures the salient details for our analysis. See the full version [17] for the details on the size of the benchmark.

We spend the remainder of Section 8 looking at some of the interesting aspects of each benchmark and explaining how the analysis responds to those aspects.

8.2 AES Encryption

The functions `AES128_encrypt`, `AES192_encrypt`, and `AES256_encrypt` are each given a plaintext input which can be composed of any number of data blocks.[21] Each function will call `AES_encrypt` internally for every provided block to encrypt it. `AES_encrypt` itself will run a specified number of rounds of encryption on the provided block. The various steps of a round of AES encryption are encapsulated in their own functions, which are called by `AES_encrypt`.

[21]Note that in this case, "blocks" refers to 16-byte chunks of data, *not* basic blocks.

The analysis' treatment of the function `AES_encrypt` is an interesting case study since it can be fully protected without the need for symbolic execution. Furthermore, it highlights the importance of our inter-procedural rules. An analogous and *highly simplified* form of the function (that still captures the salient details for our analysis) and its control-flow graph are depicted in Figures 10a and 10b, respectively. (See the full version [17] for the details on the size of the benchmark.) In the figure, `f`, `g`, and `h` represent encapsulations of various operations performed by AES encryption (e.g. mixing columns or adding in the round key). We point out that the variables `x` and y_1 do *not* correspond to secret data in the original code; their contents are the *addresses* of buffers.

Prior to analyzing `AES_encrypt`, `f`, `g`, and `h` will have already been analyzed and deduced to be pseudo transmitters that leak both of their arguments. From this, the data-flow analysis will be able to conclude that the frontier for every variable is B_1. The partial loop expansion mentioned in Section 7.1 is crucial to this deduction. Now `AES_encrypt` will be protected. There are no local transmitters to protect. The calls to `f`, `g`, and `h` can be replaced with their protected counterparts, `f'`, `g'`, and `h'`. Being pseudo transmitters, they need to be caller enforced, and so a `SPEC_BARR` is placed in B_1, the frontier of their collective arguments. The DECLASSIFLOW protected version of the code is shown in Figure 10c. Since `x` is leaked internally and cannot be deduced from the argument y_1, `AES_encrypt` is *not* a pseudo transmitter and thus cannot be caller enforced.

The crucial difference between the SLH and DECLASSIFLOW-protected versions of `AES_encrypt` is that under SLH, the protection of all the variables is present *inside* the loop. Since every loop iteration contains a branch that can be speculated, the hardening performed by SLH causes repeated delays. Thus, in the graph we see that the overhead of SLH increases as the size of the input increases. On the other hand, our protected version of `AES_encrypt` has a single `SPEC_BARR`, and within the loop body, we call `g'` and `h'` which do not have hardened loads. Thus, we pay the `SPEC_BARR` penalty once, but SLH pays a penalty for every load over and over again. This is why our protected function performs much better than its SLH counterpart.

We note that the higher-level functions which we benchmark call the protected version of `AES_encrypt` (Figure 10c) in a loop. Since `AES_encrypt` is callee enforced, the number of times the `SPEC_BARR` is encountered is linear with respect to the size of the input. That is why we see the overhead of the protected function increase with the size of the input as opposed to remaining fixed.

8.3 Djbsort

This benchmark is an interesting case study due to its heavy use of nested loops. The function takes two parameters, an array of integers `x` and the length of the array `N`. All transmitters in Djbsort arise from accessing `x` at various offsets. The core concept of the function is depicted in a *highly simplified* form (that still captures the salient details for our analysis) in Figure 11. (See the full version [17] for the details on the size of the benchmark.) Djbsort cannot be declassified using our data-flow analysis alone. The issue is by the semantics of `while` (and similarly `for`) loops, when the program is compiled, the control-flow graph will typically include

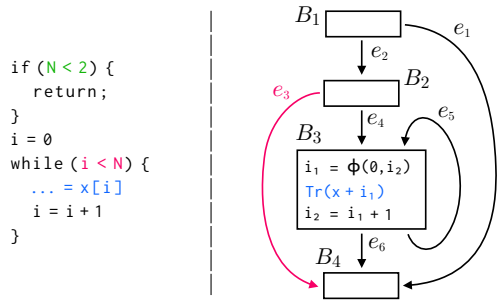


Figure 11: The core of the Djbsort benchmark from the perspective of our analysis. NOTE: The figure depicts a *highly simplified* version of the function that still captures the salient details for our analysis. See the full version [17] for the details on the size of the benchmark.

an edge that bypasses the loop body (e_3 from the figure).[22] Ideally, the frontier of `x` is B_2 since ① `Tr(x+i)` in B_2 is guaranteed to be encountered once the first branch is crossed, and ② `i` will be known at every point due to being an inductive variable with a known starting value. However, because the data-flow analysis assumes e_3 is traversable, the frontier can be lifted no higher than B_3. This problem is addressed by symbolic execution as described in Section 6.2. It can deduce that there is no possible execution of Djbsort in which the array `x` is not accessed at least once. After the symbolic execution pass, our analysis will correctly report that the frontier of `x` should be B_2.

The conditional `if (N<2)` at the top of the function is crucial for the symbolic execution tool to make the aforementioned deduction. Once the symbolic execution engine proceeds past this branch, it is armed with the path constraint $N \geq 2$. This constraint is necessary to deduce that the loop condition is always satisfied initially, and therefore that `x` is guaranteed to be leaked.

We note that although symbolic execution is needed to effectively relax Djbsort, it is *not* be able to do it on its own. Crucially, it is not `x` that is transmitted, but `x+i`. The data-flow analysis is needed to deduce that `i` is always known, and it will do so via partial loop expansion. Only then can the instrumentation for the symbolic execution pass treat `Tr(x+i)` as though it leaks `x`.[23]

Since `x` is only guaranteed to be known after it's non-speculatively confirmed that $N \geq 2$, a `SPEC_BARR` must be placed after the `if (N<2)`. After this point in the program, SLH can be disabled for the whole function.

8.4 ChaCha20

The ChaCha20 benchmark is similar to the Djbsort benchmark in that it involves a series of nested loops and that all transmitters are due to accesses of various arrays. One can intuit from the source code that if the length of the provided input is non-zero, then all arrays' addresses are guaranteed to be known. Klee must be used to prove this in an attempt to safely remove protections for the entirety of the function. One key difference between ChaCha20 and Djbsort is that in the high-level source code, there is no `if` statement that short-circuits the function if a zero-length input is provided. However, compilers will often add such checks in the form of a loop "preheader", and indeed this is what LLVM does for ChaCha20. Thus, from the point of view of our analysis (which is applied to the LLVM IR generated after compilation), the two benchmarks are quite similar in form. We thus omit any discussion of the analysis itself. A single `SPEC_BARR` is placed after the check for non-zero length but outside any loops, and SLH is disabled for the entire function.

[22] The compiler can indeed remove this edge if it can be deduced that the loop body will execute at least once. However, it is unlikely to happen if we have nested loops with complicated conditions, which is what we see in Djbsort.

[23] `Tr(x+i)` causes `x+i` to be known. Since `i` is known, we can exploit backward solvability to deduce that `x` is known.

9 RELATED WORK

Prior work Blade [40] and several recent SLH variants [36, 38, 48] share Declassiflow's goal of reducing overhead of speculative execution defenses for constant-time code. Blade protects only speculatively-accessed data: it statically constructs a data-flow graph from mis-speculated loads (which can return secrets) to transmitters and infers a minimal placement of protections that cuts off such data-flow. SSLH [36] and USLH [48] strengthen SLH to protect non-speculatively accessed data. As discussed in Section 8, these schemes will incur a higher performance penalty than Declassiflow while providing comparable security.

selSLH builds on programming language support for public/secret type information [15], which it uses to selectively apply SLH only on loads into public variables. This approach relies on typed variables, with the type system enforcing that a transmitter's operand is always typed public. However, this property isn't preserved on compilation: a machine register `rX` can hold public and secret values in different program contexts. Therefore, misspeculation from a context in which `rX` holds a secret to a context in which `rX` is public and is an operand to a transmitter can result in `rX` leaking [14, Section 4.2.2]. Overall, selSLH, like Blade, doesn't protect secret non-speculative register data. In contrast, Declassiflow protects both speculatively-accessed data (read from memory under speculation) and secret non-speculative register data, and makes no assumptions on the programming language used.

In Shivakumar et al. [39], the authors propose language primitives for writing cryptographic code that is protected from Spectre v1. These primitives can be used to implement traditional SLH along with selSLH, and a type system checks that the program meets a provided security definition. Inserting these primitives in code is not automatic and requires developer effort. In theory, their work is compatible with ours as we could use our analysis to relax code protected via their primitives. It could be that more fine-grained (and thus more beneficial) relaxations would be possible.

Finally, several works [14, 16, 25, 26] develop static analyses to detect violations of a formal notion of security against speculative leakage, which is based on the idea that a program's speculative execution should not leak more than its non-speculative execution. Declassiflow also leverages this type of security property, but its goal is to safely relax conservative protections while maintaining security.

10 CONCLUSION

This paper presented Declassiflow: a static analysis that reduces the amount of protection needed to "take speculative execution off the table" for constant-time programs. The key observation is

that as the program's non-speculative execution makes forward progress, various instructions that reveal their operands over side channels will *inevitably* execute. Such *inevitably-revealed* operands need not be protected in the program's speculative execution. This allows one to safely hoist, consolidate or even remove protection primitives, improving performance.

Longer term, an interesting question will be whether hardware-based schemes such as SPT and software-based schemes such as DECLASSIFLOW can be combined to further reduce overhead. These two approaches are complementary, in the sense that a hardware-based scheme can take advantage of fine-grain dynamic information (e.g., the current path taken by the program) while a software-based scheme can take advantage of global knowledge of the control data-flow graph. There are also opportunities to improve DECLASSIFLOW as a stand-alone analysis. For example, to automatically deduce protection placement, understand which program paths are realizable, and improve the fidelity in which the analysis understands knowledge (e.g., using ideas from QIF [12]).

Acknowledgments. We thank the anonymous reviewers and our (*truly superb*) shepherd for their valuable feedback. This research was partially funded by NSF grants 1954521, 1942888 and 2154183.

REFERENCES

[1] ChaCha20 (BearSSL). https://bearssl.org/.
[2] LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.
[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA, 2006.
[4] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Oakland'15*.
[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. In *ACM Comput. Surv. May*, 2018.
[6] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *Security'22*.
[7] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *PKC'06*.
[8] Daniel J. Bernstein. The Poly1305-AES message-authentication code. In *FSE'05*.
[9] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. Ntru prime: Reducing attack surface at low cost. In *SAC'17*.
[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS'19*.
[11] Bitcoin Code. ctaes. https://github.com/bitcoin-core/ctaes.
[12] Tevfik Bultan. Quantifying information leakage using model counting constraint solvers. In *VSTTE'20*.
[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*.
[14] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *PLDI'20*.
[15] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for timing-sensitive computation. In *PLDI'19*.
[16] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A Formal Approach to Secure Speculation. In *CSF'19*.
[17] Rutvik Choudhary, Alan Wang, Zirui Neil Zhao, Adam Morrison, and Christopher Fletcher. Declassiflow: A static analysis for modeling non-speculative knowledge to relax speculative execution security measures (full version). In *ArXiv*.
[18] Rutvik Choudhary, Jiyong Yu, Christopher W. Fletcher, and Adam Morrison. Speculative Privacy Tracking (SPT): Leaking information from speculative execution without compromising privacy. In *MICRO'21*.
[19] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Oakland'09*.
[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL'89*.
[21] Daniel J. Bernstein. djbsort. https://sorting.cr.yp.to.
[22] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *DAC'19*.
[23] Google/LLVM. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, Published 2018.
[24] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *ICISC'09*.
[25] Roberto Guanciale, Musard Balliu, and Mads Dam. InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *CCS'20*.
[26] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *Oakland'20*.
[27] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *Oakland'21*.
[28] Shay Gueron. Efficient software implementations of modular exponentiation. In *IACR ePrint'11*.
[29] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Oakland'19*.
[30] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO'96*.
[31] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Oakland'15*.
[32] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient non-observability. In *Security'21*.
[33] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *CCS'18*.
[34] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. In *ArXiv*.
[35] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. *IACR ePrint'05*.
[36] Marco Patrignani and Marco Guarnieri. Exorcising spectres with secure compilers. In *CCS'21*.
[37] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Security'15*.
[38] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: Reading from the right place at the wrong time. In *Oakland'23*.
[39] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing high-speed cryptography against spectre v1. In *IACR ePrint'22*.
[40] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with Blade. In *POPL'21*.
[41] Jose Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. Opening Pandora's box: A systematic study of new ways microarchitecture can leak private data. In *ISCA'21*.
[42] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *MICRO'19*.
[43] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *Security'22*.
[44] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *Security'14*.
[45] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *NDSS'19*.
[46] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *ISCA'20*.
[47] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *MICRO'19*.
[48] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *Security'23*.