ORIGAMI: Folding Data Structures to Reduce Timing Side-Channel Leakage

Eric Rothstein-Morris
ISTD
Singapore University of Technology and Design, Singapore
eric_rothstein@sutd.edu.sg

Jun Sun
School of Computing and Information Systems
SMU, Singapore
junsun@smu.edu.sg

Sudipta Chattopadhyay
ISTD
Singapore University of Technology and Design, Singapore
sudipta_chattopadhyay@sutd.edu.sg

***Abstract*—Timing channels in a program allow attackers to infer secret information being processed. To avoid introducing timing channels, programmers should follow Constant-Time Programming (CTP) guidelines or rely on repair tools that prevent leakage of information via timing channels. Existing repair tools prevent this leakage when programs have branches or loops whose behaviour depends on secrets; however, these repair tools do not efficiently prevent the leakage that occurs if the program accesses a data structure using secret indices. In this work, we present ORIGAMI, a set of repair rules to enforce constant read/write operations on fixed-size, multidimensional data structures so that accessing them via secret indices does not leak information. We implement ORIGAMI as a series of LLVM optimisation passes and evaluate ORIGAMI with programs from Tomcrypt and GDK libraries. Evaluation with the repaired programs using an accurate simulator (GEM5) confirms that our approach indeed repairs the timing channels in practice.**

I. Introduction

What are timing side-channel attacks?: Timing attacks are among the best known side-channel attacks [5] to ex-filtrate secret information from a program. Basic timing side-channel attacks aim to establish a relationship between inputs and execution time, which can be done by attackers who have a copy of the program. After running the program with different inputs, the attacker has a model of this input to execution-time relationship. Attackers then observe the total execution time of the same program run by the victim, and can infer the value of the secret input. More advanced timing attacks exploit micro-architectural features, e.g. caches, and they require the attacker be able to interact with these micro-architectural aspects in the machine where the victim executes the vulnerable program. Spectre style attacks [16], as discovered in 2018, are sophisticated attacks where the attacker exploits timing covert channels to ex-filtrate secret information loaded by speculative execution in the cache by reverse engineering the value of a secret input after observing cache hit/miss timings [8] or by computing the cache lines being accessed [23]. While Spectre attacks rely on speculative execution to load secrets into the cache, poorly implemented cryptographic software could directly leak secrets via memory access patterns (MAP), without relying on speculative execution [13].

This work is partially supported by the National Research Foundation, Singapore project 2019 ANR NRF 0092 and Ministry of Education Tier 2 project MOE2018-T2-1-098.

Repairing Leakage due to MAP.: If we assume that attackers can only infer information from the behaviour of the program counter, then the automatic repair of programs with timing side-channel vulnerabilities is a well understood problem: existing solutions [27], [31], [36] already close these timing side-channels while preserving functionality and preventing the appearance of undesired side-effects (e.g. unsafe memory accesses). However, once we empower the attacker to manipulate micro-architectural aspects, especially those that breach memory isolation like the ones used for Spectre [16] and Meltdown [21], the repairing of vulnerable programs remains an open problem [5]. In this work, we are particularly interested in repairing programs that leak information when a data structure is accessed using a secret value. These programs are vulnerable to an attacker that cannot read the contents of the cache directly but can manipulate and observe the state of the cache using attacks like Flush+Reload [37] or Prime+Probe [23] to infer secrets.

Consider the example program **P**, defined by

$$\text{if } A[s] \text{ then } x := 1 \text{ else } y := 0,$$

which reveals $A[s]$ under the *baseline leakage model*, because the attacker can infer the value of $A[s]$ by following the program counter. The baseline leakage model assumes that the valuations of branch conditions are exposed to attackers (see [2, §3, Example 1]), so the program **P** is unsafe if `A[s]` is a secret, but it is safe if `A[s]` is public (e.g. if `A[s]` is declassified data). Existing repair tools [27], [31], [36] offer strong security guarantees against the baseline leakage model, and can repair the program **P** with respect to this leakage model, yielding the linear-code program **T(P)**, defined by

$$x := CTSel(A[s], 1, x); y := CTSel(A[s], y, 0),$$

where $CTSel(c, a, b)$ is a constant-time selector which returns a if c is true, or b otherwise. The program **T(P)** is arguably safer with respect to the baseline leakage model, since the behaviour of the program counter no longer reveals the value of $A[s]$. Now, if we consider a leakage model which considers MAP, then the program **T(P)** leaks s, because the attacker can still infer s by probing the cache. Existing compiler-based repair tools [27], [31], [36] offer weak, inefficient, or no guarantees at all against this leakage model: Raccoon

[27] implements oblivious RAM (ORAM), which is quite taxing in terms of performance (it induces an overhead with geometric mean $\sim$ 16x); SC-ELIMINATOR [36] uses data structure preloading, but it is an unsound repair if the attacker can manipulate the cache; and the methodology presented in [31] does not repair against this leakage model. This lack of effective and efficient repair guarantees is the main motivation for our work.

Our Contributions: The authors of [29] describe a philosophy which aims to delegate the compiler the enforcement of timing side-channel freedom. Following this philosophy, we propose a set of repair rules to close the timing side-channel created by accessing fixed-size data structures indexed by secret information. Our repair rules offer strong security guarantees with respect to the leakage model that accounts for memory access patterns. In a nutshell, we replace each read access $y := A[x]$ and each write access $A[x] := y$ by a linear program that explores A, systematically loading it in memory, guaranteeing that $y = A[x]$ in the case of a read, and that $A[x] = y$ in the case of a write. These operations are similar to *fold* high-order functions, which is why we name our repair rules ORIGAMI.

We implement ORIGAMI as an LLVM opt pass to make it compatible with other target-independent compiler optimizations. The tool lets the compiler first perform optimisations, and then applies the ORIGAMI repair rules to the resulting intermediate representation code. Our pass ensures that the compiled programs have a constant memory footprint when indexing fixed-size array-like data structures using secrets. Our proposed solution only requires minimal annotation to the source code (i.e., to inform the compiler which variables and function arguments are secrets, and for loops whose bounds cannot be automatically derived by the compiler) and provides theoretical guarantees that the transformed code is secure under the memory access pattern leakage model.

There are a couple of clear limitations when repairing programs with ORIGAMI, (e.g. ORIGAMI can obfuscate a read access to a data structure whose values are secret pointers, but fails to protect the program if such resulting pointer is later used to load or store a value) which we discuss in Section IV-D. These limitations illustrate the impossibility of repairing every program with respect to the MAP leakage model while keeping the program functional, efficient, and secure.

This paper is structured as follows. We first provide a brief background in Section II. In Section III, we formally define timing side-channel freedom (TSCF) under the MAP leakage model, which is the property that we want to enforce. We present the ORIGAMI repair rules in Section IV, and we prove that they enforce TSCF under MAP for a language of while-programs; we also discuss the limitations of this enforcement. In section V, we evaluate an implementation of the ORIGAMI rules as LLVM optimization passes; we use these passes to enforce TSCF in LLVM-IR after all other compiler optimizations have taken place. We apply ORIGAMI to a small toy example and to real cryptographic ciphers from Tomcrypt [20] and to GDK library routines [32], [32], and we evaluate all the repaired programs using GEM5 – a cycle accurate simulator for x86 processor – to empirically show that all repaired programs satisfy TSCF with respect to MAP leakage. We then compare ORIGAMI against related work in Section VI, and we conclude in Section VII.

II. PRELIMINARIES

In this section, we provide the definitions and notation that we use through this work.

A. Timing Side Channel Freedom Enforcement

Why are timing side-channel vulnerabilities so hard to fix?: Both the programming languages and the computer security communities understand fairly well where timing differences could be introduced during the compilation and execution processes of software, and they tackle the problem of enforcing *timing side-channel freedom (TSCF)* using a layered approach. Unfortunately, timing differences can be (unintentionally) introduced at every step of the compilation process, and they propagate to the following stages.

For illustration purposes, let us consider a simplified version of the compilation and execution process. A program starts out as *source code*, which is then given to a compiler. The compiler often creates an *intermediate representation* (IR) of the program (e.g. a control flow graph (CFG)), which the compiler then optimises by using transformation rules that can be applied to *any* IR (e.g. dead-code elimination). We call the entity in charge of creation and optimisation of the IR the *front-end* of the compiler. The *back-end* of the compiler then compiles the optimised IR into a microarchitecture-dependent *low-level representation* (LLR), performs microarchitecture-dependent optimisations, and then creates the executable. Finally, the executable runs on the microarchitecture by following the sequence of instructions in the executable.

At the *source code* level, a developer who does not follow constant-time programming guidelines, e.g., CryptoCoding [3], can introduce timing differences by, e.g., using loops with input-dependent bounds, or by terminating early if a branch condition is satisfied, e.g. in a base case of a recursive functions. At the *IR* level, since compilers often optimise for performance, they may introduce timing differences via optimisations at the IR level, just like a programmer would at source level. To make matters more complicated, the compiler may even remove TSCF countermeasures introduced at the source code level if it deems them non-optimal, which is why developers of crypto algorithms disable compiler optimisations, or even choose to avoid compilers altogether and instead directly implement crypto routines in assembly [5].

Then, the back-end repeats the story of the front-end: it creates the LLR, optimises it and creates the executable, but its own optimisations may remove any TSCF enforcement introduced in the IR, and it may itself introduce timing differences. Finally, even if the back-end does not itself introduce timing differences or removes countermeasures added at the previous stages, the microarchitecture may manifest timing

differences during program execution; this may be because a micro-architectural instruction can vary its execution time depending on its parameters (e.g. multiplication), or due to out-of-order execution and speculative execution. In that sense, any TSCF enforcement introduced at early stages can be made irrelevant at later stages.

B. Leakage Models

We find the notion of leakage models used in `ct-verif` [2] particularly enlightening. Although their leakage models are defined based on LLVM rather than machine code, they argue in [2, §5] that "LLVM assembly code produced just before code generation [is] sufficiently similar to *any* target-machine's assembly code to provide a high level of confidence."

In the following, we provide the intuition behind three useful leakage models, what it means for a program to leak secrets with respect to them, and insights on what repairing a program with respect to each model entails.

Baseline Leakage Model: This leakage model reveals to the attacker the valuations of branch conditions. More precisely, the program $\texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2$ reveals the valuation of c, and the program $\texttt{while } c \texttt{ do } p$ reveals the valuation of c. This is the *baseline* leakage model because it is implied by all other leakage models.

A program leaks secrets with respect to this model if secrets influence the behaviour of the *program counter*, which is why it is also known as the *program counter security model* [24]. To repair a program with respect to this model, secret-dependent branches are linearised by replacing conditionals with constant-time selectors and loops are fully unrolled. This causes the behaviour of the program counter to be independent of value of secrets.

Memory Access Patterns Leakage Model (MAP): in addition to revealing the valuation of branch conditions, the MAP model reveals the indices used to access data structures. More precisely, the programs

$$A[x] := y, \quad \text{and} \quad y := A[x]$$

each reveals x because different indices may have different memory access patterns (e.g. when the cache lines for $A[x]$ and $A[x']$ are different), and the attacker can infer this information. Thus, a program leaks secrets under the MAP model if they are used to access data structures [2]. This leakage model is related to memory trace obliviousness [22], which requires constant behaviour of the memory for all public-equivalent traces. To avoid leakage under the MAP model, we must not use secrets when accessing data structures, and we must enforce a secret-independent behaviour on the program counter (to avoid leakage following the baseline model).

SC-ELIMINATOR proposes the use of preloading and must-hit analysis to repair programs so that they satisfy TSCF under the MAP leakage model. Unfortunately, this repair implicitly assumes that the state of the cache during must-hit analysis is the same as when the program executes, which is a problematic assumption if we consider that the attacker can also manipulate the cache using Prime+Probe and Flush+Reload attacks. Instead of preloading, we propose a new repair rule where instructions accessing data structures using secrets, i.e.$A[x] := y$ and $y := A[x]$, have the constant memory access patterns, thus preventing leaks under the MAP. The details of this solution are explained in detail in Section IV.

Operand Sensitive Leakage Model (OS): For completeness, we include the leakage model that distinguishes operations whose execution time are sensitive to inputs. The program $y := f(x)$ leaks its parameter x if its total execution time depends on x. This leakage model is the most general of the three models presented, as it implies the MAP and baseline models. Repairing programs with respect to the OS model at the source or compiler level is extremely challenging, because some operations offered by the micro-architecture leak their parameters (e.g. division and multiplication); thus, solutions for the OS model may need to be target dependent. Enforcing TSCF with respect to this leakage model is outside the scope of this work.

C. TSCF Under MAP Leakage - Informally

The MAP leakage model represents an attacker that is able to use the timing of hits and misses in the cache to indirectly obtain values from the cache, similar to what attackers relying on Spectre attacks do to recover the secrets loaded in memory. More precisely, if an array-like structure A is large enough to require several cache lines for it to be fully loaded in the cache, then caching $A[s]$ only fills the cache line that corresponds to the index s and its neighbouring values. An attacker can gain information about s by probing the cache, testing which parts of A result in a cache hits and which ones do not. For example, the program $\texttt{if } s < size_A \texttt{ then } x := B[A[s]]$ reveals s and $A[s]$ under the MAP leakage model, because the state of the cache is different for different values of s. This program does not reveal $B[A[s]]$, only the indices used to access it.

D. Guarded Kleene Algebra with Tests (GKAT)

Guarded Kleene Algebra with Tests (GKAT) is a modern formalism that offers a propositional abstraction of imperative while-programs with uninterpreted actions [30]. The specialty of GKAT is to enable reasoning about properties of programs *by merely looking at their structure and not at their (functional) semantics*. This makes GKAT interesting for modelling transformations at the compiler level [18], because a general-purpose compiler should not need know the exact semantics of a program to optimise it; in general, the compiler should look for structural patterns which enable optimisations, just as GKAT does for reasoning.

We use GKAT to provide a formal foundation in reasoning about TSCF for arbitrary programs. Specifically, the axioms and rules for GKAT expressions help us reason about the equivalence of programs. In this setting, the notion of semantic equivalence is defined over uninterpreted actions using languages of *guarded strings*, defined using *actions* and *tests*.

Actions, Tests and Expressions: Every GKAT is parametrised by a set of abstract *actions* Σ and a finite set of abstract *primitive tests* T. We assume T and Σ are disjoint and non-empty. A test $t \in T$ is an atomic proposition about the state of the program, and the execution of an action $p \in \Sigma$ can affect the state. We form *GKAT expressions* (GKATx) with the grammar presented in Figure 1.

Atoms: An *atom* is a truth assignment of all the tests in T. We denote atoms by α, β, and γ, and the set of atoms by At. For example, if $T = \{t_1, t_2\}$, the boolean expressions $\alpha = \overline{t_1} \cdot \overline{t_2}$, $\beta = \overline{t_1} \cdot t_2$, $\gamma = t_1 \cdot \overline{t_2}$, and $\delta = t_1 \cdot t_2$ are all the atoms, where $\overline{t_i}$ is the complement of t_i, for $i \in \{1, 2\}$.

Guarded strings are an intercalation of a logical atom and an action, which can be seen as the concatenation of $\{\texttt{pre}\}\{\texttt{action}\}\{\texttt{pos}\}$ elements, where `pre` and `pos` represent the precondition and postcondition of the `action` (any `pre` and any `pos` as we work with uninterpreted actions, but a `pos` and a `pre` need to be compatible to be concatenated).

Guarded Strings: A *guarded string* g is an element of the set $\mathrm{GS} := \mathrm{At} \cdot (\Sigma \cdot \mathrm{At})^*$, and it models a trace of an abstract program. To compose guarded strings, we use *fusion product* $\diamond\colon \mathrm{GS} \times \mathrm{GS} \to \mathrm{GS}$, a partial function defined by

$$w\alpha \diamond \beta v \triangleq \begin{cases} w\alpha v, & \text{if } \alpha = \beta; \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{1}$$

The fusion product of sets $L_1, L_2 \subseteq \mathrm{GS}$ is defined by

$$L_1 \diamond L_2 \triangleq \{g_1 \diamond g_2 \mid g_1 \in L_1, g_2 \in L_2, \text{ and } g_1 \diamond g_2 \text{ is defined}\}.$$

Language-based Semantics: The *language-based semantics* of a GKATx is a set of guarded strings (i.e., a language) defined by

$$\begin{aligned} [\![p]\!] &\triangleq \{\alpha p \beta \mid \alpha, \beta \in \mathrm{At}\}, \\ [\![b]\!] &\triangleq \{\alpha \mid \alpha \in \mathrm{At} \text{ and } \alpha \Rightarrow b\}, \\ [\![f \cdot g]\!] &\triangleq [\![f]\!] \diamond [\![g]\!], \\ [\![f +_b g]\!] &\triangleq ([\![b]\!] \diamond [\![f]\!]) \cup ((\mathrm{At} - [\![b]\!]) \diamond [\![g]\!]), \\ \left[\!\left[e^{(b)}\right]\!\right] &\triangleq \bigcup_{n \geq 0} ([\![b]\!] \diamond [\![e]\!])^n \diamond (\mathrm{At} - [\![b]\!]), \end{aligned}$$

where $L^0 \triangleq \mathrm{At}$ and $L^{n+1} \triangleq L^n \diamond L$, for $L \subseteq \mathrm{GS}$. Since actions in GKATx are uninterpreted, language-based semantics are an over-approximation of functional semantics.

Rules: A *GKAT rule* is an equivalence that lets us transform an arbitrary GKATx into a GKATx that is syntactically different, yet semantically equivalent; e.g., $b \cdot (e +_b f) \equiv b \cdot e$.

III. Formalising TSCF with MAP

Informally, TSCF means that all secret-dependent program traces have very similar *execution time*. If we model program traces using guarded strings, then the execution time of a program trace is the sum of the execution time required by each of its actions. To quantify the execution time of actions, we use a *time metric*.

$b, c, d \in \mathscr{B} ::=$		$e, f, g \in \mathrm{GKATx} ::=$	
0	**False**	$p \in \Sigma$	**do** p
1	**True**	$b \in \mathscr{B}$	**assert** b
$t \in T$	t	$e \cdot f$	e;f
$b \cdot c$	b **and** c	$f +_b g$	**if** b **then** f **else** g
$b + c$	b **or** c	$e^{(b)}$	**while** b **do** e
$\bar{b}$	**not** b		

Fig. 1: **Left:** boolean expressions. **Right:** GKAT expressions (from [30]).

A *time metric* associates each actions in a program trace with a time consumption. Under the MAP leakage model, an action p may consume different amounts of time depending on the state of the cache when p is executed: if a results in several cache hits then it consumes less time than it would if a resulted in several cache misses. Thus, the memory access pattern of a program trace directly impacts its execution time. We formalise this notion with the metric `mem`.

The `mem` Time Metric: Let $\mathscr{V}(p)$ be the variables used in the action p, let $\omega\colon \mathrm{GS} \times \mathscr{V} \to \mathbb{B}$ be a function where $\omega(g, v)$ states if the variable v is in the cache after executing g (or after executing nothing if $g \in \mathrm{At}$), let $\texttt{hit}_g(p) \triangleq \{v \in \mathscr{V}(p) | \omega(g, v)\}$ be the set of variables in p that are present in the cache, and let $\texttt{miss}_g(p) \triangleq \{v \notin \mathscr{V}(p) | \omega(g, v)\}$; we define $\texttt{mem}(g)(p)$, the *MAP of p (with respect to g)*, by

$$\texttt{mem}(g)(p) = \eta \times \texttt{miss}_g(p) + \mu \times \texttt{hit}_g(p)$$

where $\eta, \mu \in \mathbb{R}^+$ are constants with η being much greater than μ, respectively modelling the time to load a variable from memory and the time to load a variable from the cache.

Given a GS g, the *MAP of g*, denoted $\texttt{RC}_{\texttt{mem}}(g)$, is the accumulated MAP of its actions; formally,

$$\texttt{RC}_{\texttt{mem}}(g) \triangleq \begin{cases} 0, & \text{if } g \in \mathrm{At}; \\ \texttt{RC}_{\texttt{mem}}(h) + \texttt{mem}(h)(p), & \text{if } g = h \cdot p \cdot \alpha; \end{cases}$$

The metric `mem` is *causal*, since the resource consumption of actions depends on the history of the execution.

Now, consider two values for a secret s, say $s = 0$ and $s = n$; the `mem` values of $x := A[s]$ is

$$\begin{aligned} \texttt{mem}(1)(x := A[s]) &= \eta \times \{x, A[0], s\} + \mu \times \emptyset, \text{ if } s = 0, \\ \texttt{mem}(1)(x := A[s]) &= \eta \times \{x, A[n], s\} + \mu \times \emptyset, \text{ if } s = n. \end{aligned}$$

The secret s leaks because the MAP of the expression $x := A[s]$ depends on the value of s.When $A[0]$ and $A[n]$ are loaded into the cache, and they are placed in different regions of the cache; thus, the attacker can infer the value of s by looking at which regions corresponding to A are loaded.

Constant MAP.: Given a language of guarded strings $L \subseteq \mathrm{GS}$, we say that L has *constant resource consumption with respect to `mem`* if and only if $\texttt{RC}_{\texttt{mem}}(g_1) = \texttt{RC}_{\texttt{mem}}(g_2)$ for all $g_1, g_2 \in L$.

Constant resource consumption in its current form requires *all* GS in the language L to have the same resource consumption. However, a program that has no secret values trivially

satisfy this property, since there are no secrets that could be leaked. Thus, we need to enrich the current definition of constant resource consumption so that we only require traces that exclusively differ on secret values have the same resource consumption. As it is standard (see e.g., [2], [24]), we introduce a notion of *public equality*.

Let $\mathscr{V}$ be the set of public variables, and let $[\![\mathscr{V}]\!]$ be the set of valuation functions which map variables to values; we extend the notion of guarded strings so that they are now generated by the grammar

$$\text{GS} := ([\![\mathscr{V}]\!] \times \text{At}) \cdot (\Sigma \cdot ([\![\mathscr{V}]\!] \times \text{At}))^* .$$

Now, guarded strings satisfy one of the patterns (Γ, α) or $(\Gamma, \alpha) \cdot p \cdot g$, where Γ is a valuation of the public variables, α is an atom and g is a guarded string. Two guarded strings g_1 and g_2 are *equal on their public variables*, denoted $g_1 =_p g_2$, if and only if their *initial* valuations are equal on public variables.

We now define a formal notion of constant resource consumption, which we apply to the `mem` metric.

Definition 1 (TSCF with MAP)**.** *Given a language of guarded strings $L \subseteq GS$, we say that L has* secure constant time consumption guarantees with respect to `mem` *if and only if, for all $g_1, g_2 \in L$:*

$$g_1 =_p g_2 \Rightarrow RC_{mem}(g_1) = RC_{mem}(g_2).$$

Equivalently, we say that L satisfies TSCF.

This definition naturally extends to GKATx. A GKATx e satisfies TSCF if and only if $[\![e]\!]$ satisfies TSCF.

Attacker model: Definition 1 characterises an attacker model where the attacker that can choose all the values of public variable before execution of every GKATx and, at the end of the execution of the GKATx, can measure the total number of hits and misses to the cache. Additionally, the attacker can execute Prime+Probe [23] or Flush+Reload [37] attacks, either remotely or locally. More precisely, this attacker can see which addresses are loaded in the cache but cannot see their contents, and can flush the cache at will. This attacker is similar to the attacker which exploits a Spectre V1 vulnerability to extract secrets via the cache, but our attacker does not rely on speculative execution (speculative execution is beyond the scope of this work).

In the following sections, we provide a concrete GKAT for an enriched language of while-programs. Using this GKAT, we describe existing repair rules used by other repair tools to enforce TSCF under the baseline leakage model. We discuss why these rules are not enough to enforce TSCF under the MAP leakage model, and we propose our own set of rules to fill this gap.

IV. Repairing TSCF under MAP with ORIGAMI

Existing program repair solutions for TSCF [27], [31], [36] offer high-overhead, unsound guarantees, or no guarantees at all with respect to the MAP leakage model. Raccoon [27] implements ORAM to enforce TSCF in the MAP leakage model. While secure, the use of ORAM is quite taxing in terms of performance overhead. This overhead is unfortunate, which is why we look for other software/compiler-based alternatives to repair TSCF for the MAP model. SC-Eliminator [36] uses must-hit analysis and data structure preloading to enforce TSCF in the MAP leakage model. However, the solution presented is unsound when we consider that the attacker can manipulate the cache. More precisely, SC-Eliminator implicitly assumes that the state of the cache during the must-hit analysis is maintained through execution, which is not true since the attacker can flush the cache after data structures have been preloaded, rendering the must-hit assumptions invalid. Finally, the methodology presented in [31] only offers security guarantees with respect to the baseline leakage model, and not with respect to the MAP leakage model.

In the following, we present an enriched language of while-programs, and we study the possible causes for different MAP. First, we define a concrete GKAT and give more precise definitions about what their time consumption means. Then, we formally present the ORIGAMI program transformation rules, and we justify their soundness formally; i.e., we prove that they enforce TSCF for the MAP leakage model. Then, given that the MAP model implies the baseline model, we include for completeness the well-known repair rules used to repair branches and loops. We proceed to discuss limitations of enforcement, and finally we show a natural extension of ORIGAMI from arrays to multidimensional fixed-size data structures.

A. A Concrete GKAT

We want to keep the GKAT as abstract as possible, but we do need to define when a variable hits or misses the cache.

We start small by considering uni-dimensional arrays. Let $\mathscr{V}$ be the set of variables names; let $\mathscr{S}$ be the set of names of arrays. Let $\vec{S} \in \mathscr{S}, x \in \mathscr{V}$ and $n \in \mathbb{N}$; we define the set of *atomic expressions* $\mathscr{A}$ by the grammar

$$a \in \mathscr{A} ::= x \mid n \mid \vec{S}[x] \mid \vec{S}[n].$$

We define the set of *tests* T by the grammar

$$t \in T ::= \quad a_1 = a_2, \text{ with } a_1, a_2 \in \mathscr{A}.$$

Let $a_1, a_2, a_3 \in \mathscr{A}$, $b \in \mathscr{B}$; we define the set of *actions* Σ by the grammar

$$p \in \Sigma ::= a_1 := a_2 \mid a := \textbf{sel}(b, a_1, a_2)$$

The semantics of the language is standard, and we omit the details since they are largely irrelevant for the purposes of enforcing TSCF under the MAP leakage model. More precisely, we care when and where we load elements in the cache, independently of the functional semantics of an action. Nevertheless, we assume that the compiler discards expressions that are not well-formed (e.g. $1 := A[x]$). We consider nested expressions to be using syntactic sugar; e.g. $A[x] := B[C[x]]$ is equivalent to $x_1 := C[x] \cdot x_2 := B[x_1] \cdot A[x] := x_2$.

B. ORIGAMI Rules

It is impossible to repair programs that are not TSCF with traditional GKAT rules. Since GKAT rules preserve semantics, non-TSCF expressions are never transformed into TSCF expressions via these rules. Thus, for the purposes of enforcing TSCF, we assume two equivalences between actions and GKATx: the *read equivalence* $(y := A[x]) \equiv \mathscr{O}(y := A[x])$ and the *write equivalence* $(A[x] := y) \equiv \mathscr{O}(A[x] := y)$, where A has a fixed size of $n + 1$,

$$\begin{aligned}&\mathscr{O}(y := A[x]) \triangleq \\ &\quad acc := \textbf{sel}(x = 0, A[0], acc)\cdot \\ &\quad acc := \textbf{sel}(x = 1, A[1], acc)\cdot \\ &\quad \dots \\ &\quad acc := \textbf{sel}(x = n, A[n], acc)\cdot \\ &\quad y := acc,\end{aligned}$$

and

$$\begin{aligned}&\mathscr{O}(A[x] := y) \triangleq \\ &\quad A[0] := \textbf{sel}(x = 0, y, A[0])\cdot \\ &\quad A[1] := \textbf{sel}(x = 1, y, A[1])\cdot \\ &\quad \dots \\ &\quad A[n] := \textbf{sel}(x = n, y, A[n]).\end{aligned}$$

We respectively denote the *read ORIGAMI rule* and the *write ORIGAMI rule* by $(y := A[x]) \rightsquigarrow \mathscr{O}(y := A[x])$ and $(A[x] := y) \rightsquigarrow \mathscr{O}(A[x] := y)$. Functionally, these read and write rules are sound: $\mathscr{O}(y := A[x])$ implements an iteration over A, loading every value $A[i]$ but only storing it in acc if $i = x$, and $\mathscr{O}(A[x] := y)$ also implements an iteration, which loads $A[i]$ and stores it back at $A[i]$ if $i \neq x$, or loads $A[i]$ but stores y otherwise. These rules lift naturally to multidimensional arrays.

We now provide an argument for why these rules are sound with respect to TSCF. We assume that $0 \leq x \leq n$, which we consider a reasonable assumption since in many languages the semantics of the expression $A[x]$ where $x > n$ is undefined or triggers an exception. To show why $\mathscr{O}(y := A[x])$ is TSCF with respect to the MAP leakage, we do a proof by induction on the size of A.

Theorem 1. *Let A be an array-like data structure of size $n+1$ with $n \in \mathbb{N}$, then $[\![\mathscr{O}(y := A[x])]\!]$ satisfies TSCF with MAP.*

Proof. The intuition behind the proof is the following: this ORIGAMI rule expands a single access to the data structure A into a constant sequence of accesses to A with fixed parameters, folding a **sel** instruction to accumulate the value that would be computed by the access $A[x]$; since the new sequence of accesses to A is independent of x, the MAP of $\mathscr{O}(y := A[x])$ does not leak x. □

We use a similar argument to prove TSCF of $\mathscr{O}(A[x] := y)$ expressions.

Theorem 2. *Let A be an array-like data structure of size $n+1$ with $n \in \mathbb{N}$, then $[\![\mathscr{O}(A[x] := y)]\!]$ satisfies TSCF with MAP.*

Proof. This ORIGAMI rule also expands the single access $A[x]$ into a constant sequence of accesses to A with fixed parameters, but this time it loads from every position of A, and it stores a value chosen via a **sel**; again, since this new sequence of accesses to A is independent of x, the MAP of $\mathscr{O}(A[x] := y)$ does not leak x. □

C. Repair Rules for the Baseline Model

Since the MAP implies the baseline model, we must repair programs also with respect to the baseline model. The baseline repair rules are well known, but we briefly mention them here for completeness. Existing repair tools, including [27], [31], [36], are in general capable of repairing programs with respect to the baseline leakage model using the following repair rules, whose basic strategy consists of removing branches via predication and by unrolling loops once they have been fixed to have a constant number of iterations.

Branch predication: Given a program of the form

$$\text{if } b \text{ then } x := e_1 \text{ else } x := e_2$$

whose corresponding GKATx is $(x := e_1) +_b (x := e_2)$, the *predication repair rule* replaces the branch by the select instruction $x := \textbf{sel}(b, e_1, e_2)$. The tools [27], [31], [36] implement this rule to remove leaks due to branching. There are a couple of minor complications; repairing the GKATx $(a := b/c) +_{c\neq 0} 1$ could introduce runtime exceptions that were previously prevented by the conditional (e.g. division by zero or accessing an array out of bounds). However, it suffices to take a couple extra precautions before repair, as shown in [27] and [31].

Loop Unrolling: Given a program of the form

$$\text{while } b \text{ do } e$$

which corresponds to the GKATx $e^{(b)}$, the *loop unrolling rule* replaces the program with a sequence of linearisable branches

$$\underbrace{(e +_b 1) \cdot (e +_b 1) \cdot \ldots \cdot (e +_b 1)}_{k \text{ times}}.$$

where k is the maximum number of iterations that the loop could execute for all secrets.

The major complication to apply the loop unrolling rule is determining k. Raccoon does not protect information leaks from loop trip counts [27], so they cannot repair programs with loops whose trip count is originally secret dependent. `SC-eliminator` arbitrarily chooses a value from a list of fixed sizes (e.g. 64, 128, 256) depending on static analysis, [31] assumes that loops are already unrolled, and [27] do not protect against information leaks from loop trip counts.

Limitations of Predication and Unrolling: Neither predication nor unrolling repair programs with respect to the MAP leakage model. The loopless and branchless program $x := A[s]$ where s is a secret and A is a data structure has different memory access patterns for the different values of s; this limitation is the main motivation for ORIGAMI.

Vulnerable to OS Leakage: Since the MAP leakage model is weaker than the OS leakage model, ORIGAMI does not repair timing side-channel vulnerabilities that arise by the use of operations whose execution time varies depending on their parameters. We only repair leakage that occurs due to accesses to data structures using secret indices.

D. Limitations of ORIGAMI

Secret Pointers: While we can fold a data structure whose data are secret pointers without revealing the value of the resulting pointer or the value used to access it, once this resulting pointer is loaded, it is leaked to the attacker via the cache. Thus, ORIGAMI should not be used to repair programs which contain secret pointers that are naively accessed.

Secret Data Structure Sizes: The MAP of $\mathcal{O}(y := A[x])$ and of $\mathcal{O}(A[x] := y)$ depends on the size of A. This adds a caveat for using ORIGAMI to enforce TSCF under MAP: the size of data structures indexed by secrets must be public, since the size of A can be inferred via timing. We believe this caveat is acceptable, since several cryptographic algorithms fix the size of the data structures that hold secrets (e.g. AES has key sizes 128, 192 or 256 bits). We also believe this case is dual to trying to repair a loop by unrolling it if the loop bound depends on an unbounded secret: it is impossible to repair without compromising its functionality [31], [36].

Out-of-bounds Accesses: As a side-effect of applying the ORIGAMI rules, accessing a data structure A of size n using a sensitive index x such that $x > n$ no longer results in a runtime error. The repaired version of $y := A[x]$ simply sets y to the initial value of the accumulator acc if $x > n$. Similarly, a repaired version of $A[x] := y$ where $x > n$ never stores y; it simply loads all $A[i]$ and writes them back. Consequently, these new behaviours leak to the attacker the information $x > n$, assuming that n is public. This limitation is arguably artificial, since the original program possibly also leaks this information through a runtime error.

Vulnerabilities Beyond MAP: As we previously stated, ORIGAMI does not offer security guarantees for programs that are vulnerable due to speculative execution or that are vulnerable in the OS leakage model, since these have leakage and attacker models that are beyond the scope of this work.

V. Evaluation

Three research questions motivate the following empirical evaluation:

- RQ1: How effectively can we enforce TSCF?
- RQ2: How efficiently can we enforce TSCF?
- RQ3: How do our enforcement results vary when compiler optimizations are considered?

We divide the evaluation of ORIGAMI into two sections: 1), an example program written in C, vulnerable with respect to the baseline leakage model, which illustrates what to expect when repairing programs with ORIGAMI, and 2) a set of three test programs from real libraries (also written in C): we test the `rijndael_setup` function from the AES implementation in libtomcrypt [20], and the functions `gdk-keyname` and `gdk-keyuni` from the GDK Linux library.

A. Implementation

We implement the ORIGAMI repair rules presented in Section IV using LLVM (version 13) as compiler optimisation passes. The compilation chain is as follows: using `clang` with either the `-O0` or the `-O3` optimisation flag, we create an intermediate representation in LLVM-IR. This IR representation has been already optimised by `clang`, so we can apply the ORIGAMI rules without fearing them being removed by the optimization passes. Finally, to obtain an executable, we use `llc` with disabled optimizations (i.e., using the `-O0` flag) to compile and link the repaired LLVM-IR. The following experiments show that ORIGAMI rules are preserved by this final compilation step with disabled optimisations.

To mark which variables hold secrets, we rely on compiler annotations. Unfortunately, we did not find any reliable tool that performs taint propagation and analysis from the source language C to LLVM-IR, which is why we implement one. We provide the details of the tainting procedure in Section V-C.

B. Experimental Setup

We use *Gem5* [9] to evaluate the efficiency, effectiveness and compatibility with compiler optimisations of the ORIGAMI repair rules. Gem5 is a highly-configurable simulator which encompasses system-level architecture and processor micro-architecture. The setup for the simulated environment in Gem5 consists of a single TimingSimpleCPU with a 1GHz processor that possesses only a level one 2-set associative cache, which has a cache-line size of 32 bytes; the data cache has size 8kB and the instruction cache has size 16kB. Gem5 provides statistical data from the execution of binaries, including, among other metrics, the number of CPU cycles and the number of hits and misses on both data and instruction caches. We measure the variance of these metrics to evaluate the effectiveness of TSCF enforcement (RQ1). We use the growth factor $m_{repaired}/m_{original}$ of each metric m to evaluate the efficiency of TSCF enforcement (RQ2). Finally, to evaluate the compatibility of ORIGAMI with compiler optimisations (RQ3), we repair executables compiled with `O0` and with `O3` to see if there are any significant differences.

For each benchmark, we generate a set of five random secret inputs. We use these five instances of the secret to obtain the following metrics: the number of CPU cycles, and the number of hits and misses on both data and instruction caches. An implementation is *vulnerable with respect to the MAP leakage model* if there is variance in the numbers of cache hits or misses. Similarly, an implementation is *vulnerable with respect to the OS leakage model* if there is variance in the numbers

of CPU cycles. The ORIGAMI rules are effective at repairing programs with respect to the MAP leakage model if, for all original programs that are vulnerable with respect to MAP, they are no longer vulnerable with respect to MAP after their repair. Since ORIGAMI does not repair with respect to the OS leakage model, we do not expect the variance of CPU cycles in repaired programs to be zero, which means that they remain vulnerable with respect to OS.

C. Taint Analysis

We follow the philosophy presented by the authors of [29], where the programmer works together with the compiler to enforce TSCF. In our case, the programmer must provide the information that the compiler cannot derive on its own to apply the repair rules, i.e., which variables contain secrets, the maximal loop bounds of loops that depend on secrets, and which functions should not be inlined by the compiler to make the repair process more efficient. For that, the programmer should annotate the source code as follows: 1) annotate secret variables at their declaration, 2) annotate sensitive loops by giving them a maximum *constant integer* loop bound, and 3) annotate functions that manipulate secrets such that the compiler does not inline them.

We mark the variables that contain secrets with the annotation `__attribute__(annotate("secret"))`. We refer to these annotated variables as *taint sources*. Whenever ORIGAMI finds a loop whose bound depends on a secret and no bound has been provided, ORIGAMI rejects the program and asks the programmer to provide the annotation `__attribute__(annotate("bound=N"))` on the respective taint source, where N is the maximum number of times the loop could iterate given any secret. Finally, the programmer should the annotation `__attribute__((noinline))` at function declarations to prevent the inlining of functions that manipulate secrets, since that would unnecessarily increase the size of the repaired binaries (especially when loop unrolling is applied). AS a consequence, our taint analysis is not inter-procedural, and the programmers must annotate secret variables and parameters in each function.

To implement taint propagation from C to LLVM-IR, we identify the taint sources at the IR level, and we propagate the taint using both data and control flow dependencies, which we obtain from analysis passes included in LLVM.

D. Proof of Concept (POC)

Let $A[10][10]$ be a two-dimensional data structure of size 10×10 with randomly generated integer values; now, consider the program

```
int secret1 = N % 10, secret2 = M % 10;
for (int i=0; i<secret1; i++)
  for (int j=0; j<secret2; j++)
    A[j][i]=A[i][j];
```

where `N` and `M` are randomly generated secret integers. This program is vulnerable with respect to both the baseline leakage model and the MAP leakage model. The loop bound of the outer loop depends on `N` and the loop bound of the inner loop depends on `M`. The first access we perform in the data structure is `A[N][M]`, which leaks both `N` and `M` according to the MAP leakage model. ORIGAMI needs to fix both the dependence of loop bounds on secrets and the MAP for this program so that it is constant. We present the metrics of executing the repaired version of this program in Table I.

a) Results Analysis.: In both `O0` and `O3`, the original program displays variance in the metrics of the instructions and data cache. After applying ORIGAMI, the variance is zero in all metrics, which confirms that the repair tool is forcing constant accesses to A, and it is ensuring that no information leaks due to MAP. We remark that the number of CPU cycles (column `NumCycles`) has a variance of zero, meaning that the repaired version does not leak with respect to the OS leakage model; however, this is due to the repaired program not using CPU operations whose execution time depends on secrets (other benchmarks do not satisfy this condition).

Repairing programs which contain loops whose trip counts depend on secrets can exponentially increase the size and execution time of programs. The POC program uses secrets which range from 0 to 9, and loops iterate a different number of times depending on the value of the secrets. ORIGAMI enforces TSCF without sacrificing functionality by forcing the program to run the outer loop 10 times and the inner loop 10 times no matter which secrets are provided, obfuscating no-ops to preserve functionality with sel instructions. This exponential growth is unavoidable, otherwise the secrets would leak via the timing channel.

The overhead for repairing the POC program (column `Factor`) is high: 5.11x for `O0` and 30.2x for `O3`. To repair this program, ORIGAMI must first force both loops to have the same number of iterations independently of the secret given, so the program transforms from one with an average of 25 assignments per execution to a program which always performs 100 assignments per execution. Secondly, ORIGAMI folds each of those 100 assignments over A, forcing the MAP to be constant and preventing the secrets from leaking via the cache under the MAP leakage model. Then, by applying loop unrolling, ORIGAMI prevents secrets from leaking via the program counter.

The excessive overhead incurred by repairing this program may be discouraging at first, but it is mainly due to loop unrolling, which is not always required for sensitive programs. We note that Raccoon does not repair programs with sensitive loop trip counts like the POC, so we believe that their mean 16x performance overhead is only valid for programs which do not have sensitive loops. In the following, we compute the overhead of repairing programs that require little or no loop unrolling to be repaired.

E. ORIGAMI on AES and GDK

We now describe the other example programs that we used for benchmarking ORIGAMI, and we briefly explain the results of the experiments. We repair the `rijndael_setup`

TABLE I: Simulation Results for the POC

Metric	Optimization O0					Optimization O3				
	Average			Variance		Average			Variance	
	Orig.	Rep.	Factor	Orig.	Rep.	Orig.	Rep.	Factor	Orig.	Rep.
Size	16504	884856	53.615	0	0	16504	6082680	368.558	0	0
NumCycles	606469.2	3098976	5.11	1908459.2	0	604672.8	18269050	30.213	169281.2	0
Icache hits	114734.2	397059	3.461	197395.7	0	114214.2	1829849	16.021	13854.2	0
Icache misses	931	14535	15.612	0	0	930.6	95739	102.879	0.8	0
Dcache hits	114734.2	397059	3.461	197395.7	0	114214.2	1829849	16.021	13854.2	0
Dcache misses	931	14535	15.612	0	0	930.6	95739	102.879	0.8	0

TABLE II: Metrics for AES and GDK

Metric	O0					O3				
	Average			Variance		Average			Variance	
	Orig.	Rep.	Factor	Orig.	Rep.	Orig.	Rep.	Factor	Orig.	Rep.
Tomcrypt AES										
Size	33296	33296	1	0	0	33232	33232	1	0	0
NumCycles	904893.7	898276	0.993	148556.33	0	902331.9	895064	0.992	98294.94	0
Icache hits	113393	113002	0.997	0	0	112650	112428	0.998	0	0
Icache misses	1489	1479	0.993	0	0	1493	1476	0.989	0	0
Dcache hits	28246.9	28014	0.992	7.99	0	27809.35	27776	0.999	7.71	0
Dcache misses	3822.1	3785	0.99	7.99	0	3819.65	3777	0.989	7.71	0
gdk_unicode_to_keyval										
Size	22520	51192	2.273	0	0	20520	192552	9.384	0	0
NumCycles	885008	1042148	1.178	75162	0	882540.8	1626899.6	1.843	13443.2	179828.8
Icache hits	112674.2	122967	1.091	361.7	0	112553.8	159844	1.42	43.2	0
Icache misses	1474.4	2428	1.647	0.3	0	1469	6880	4.683	0	0
Dcache hits	27868.8	30970	1.111	64.7	0	27754.8	28742	1.036	0.2	0
Dcache misses	3779.6	3927	1.039	0.8	0	3782	3794	1.003	0	0
gdk_keyval_name										
Size	49456	53552	1.083	0	0	49080	368568	7.51	0	0
NumCycles	903146.4	907018	1.004	7840.8	0	900839.2	2269991.2	2.52	819.2	54915.2
Icache hits	112741.2	114045	1.012	24.2	0	112558.4	199660	1.774	0.8	0
Icache misses	1470	1604	1.091	0	0	1466	11481	7.832	0	0
Dcache hits	27719.6	28341	1.022	1.8	0	27600	28661	1.038	0	0
Dcache misses	3936	3774	0.959	0	0	3936	3950	1.004	0	0

function from the *Advanced Encryption Standard (AES)* implementation in libtomcrypt [20], and the functions `gdk_unicode_to_keyval` and `gdk_keyval_name` of the *Linux GDK library*.

AES is a specification for data encryption to establish secure communication, which receives a plaintext message and as an array of size 16 bytes as secret inputs (i.e., a key). The `rijndael_setup` function performs a series of memory accesses to setup the forward key. The function `gdk_unicode_to_keyval` converts a secret ISO10646 character to a key symbol [32]. The secret input is a single unsigned integer value. This function uses a binary search that depends on a secret input, and it loads from a structure using a secret index during that binary search. The function `gdk_keyval_name` converts a key value into a symbolic name [32]. The secret input k is a single unsigned integer value.

The metrics of the repaired version of these functions appear in Table II. We conclude that ORIGAMI successfully repairs all studied functions with respect to the MAP leakage model, because the variance in the number of hits and misses in both caches is zero for all repaired benchmarks. We also confirm that compiler optimisations can cause a program that was originally safe with respect to the OS leakage model to become vulnerable. Nevertheless, ORIGAMI is compatible with compiler optimisations, since the repairs can be applied after compiler optimisations, and preserved in the last steps of the compilation chain.

If we take the number of CPU cycles as a measure of time, then the overhead of applying ORIGAMI in these benchmarks has a geometric mean of 1.23x, which is significantly better than the 16x overhead incurred by Raccoon. The high variation in the number of CPU cycles that we observe in Table II in the repaired versions of `O3` optimised programs is due to the aggregation of operand-sensitive instructions (e.g., addition and multiplication); although these instructions are executed the same number of times, there is a direct relationship between the number of CPU cycles and the value of input parameters, which causes variance in the total number of CPU cycles for different values of the secrets.

In some cases, the repair slightly affects the program positively in terms of performance. We speculate that this is caused

by ORIGAMI forcing the control flow graph (CFG) to match the structure of a GKAT expression: during the transformation, the LLVM pass manager might optimise some parts of the code, compensating the overhead introduced by the repair and improving overall performance.

This evaluation reinforces the philosophy of [29]: compilers can be empowered to close timing-side channels automatically, and programmers need only worry about providing the right information to the compiler; independently of whether the programmer follows CryptoCoding guidelines or not.

VI. Related Work

The existing solutions for closing timing side-channels proposed by researchers in both computer security and programming languages can be classified into two complementary main categories: *verification and testing* solutions and *enforcement* solutions. Verification and testing solutions address the problem of checking whether a system has any TSC vulnerabilities. Enforcement solutions ensure that the resulting systems have no TSC vulnerabilities.

Verification and Testing Solutions.: These solutions help us determine whether a system has timing side-channel vulnerabilities, but they do not offer automated repair solutions. ORIGAMI is a static enforcement repair solution, so it does not fit the verification nor testing categories. However, we believe that ORIGAMI could be used in conjunction with verification solutions to check which programs effectively need repair before applying ORIGAMI unnecessarily.

There are verification solutions for source (e.g., ABPV13 [4],Low* [26] and [25]), intermediate (e.g., [2], [28]), assembly (e.g., [1], [10]) and binary (e.g., [14], [17]) levels. We refer the interested reader to Barbosa et al. [5, §IV], which contains an overview of tools for side-channel resistance. The verification procedure presented in [6] does not determine whether programs satisfy TSCF or not; instead, they verify whether compilers preserve the cryptographic constant-time properties of programs during compilation. This is a better guarantee that just trusting the compilers to do so. They do not address the problem of program repair.

Testing solutions like ct-fuzz [15] and [7] aim to provide counterexamples that violate TSCF. By a counterexample, we mean two sensitive inputs that could cause a program to display different execution times. However, these testing solutions do not repair programs in case that they find a counterexample. Similarly to verification solutions, we believe that ORIGAMI can be used in conjunction with these testing solutions.

Enforcement solutions.: Enforcement solutions are quite diverse, and can be further subdivided into two categories: *runtime enforcement* and *static enforcement*. Runtime enforcement solutions, like SCHMIT [33], alter the *execution* of programs to dynamically reduce the leakage of existing side channels. Static enforcement solutions modify the programs before execution, and do not monitor their execution. Static enforcement solutions can be further subdivided into two categories: *synthesis* and *repair*. *Synthesis* solutions apply a security-by-design principle, so systems generated by this tools satisfy TSCF. Among synthesis tools we find FaCT [12] and CompCert [19].

FaCT is a C-like DSL that always generates constant-time LLVM bitcode, and CompCert is a certified compiler that ensures that properties satisfied by the original program are preserved in the compiled program. We appreciate the security-by-design aspect offered by FaCT: if only secure programs can be written, then TSCF is automatically enforced. However, this guarantee only holds if optimization passes are disabled, since they might introduce TSC vulnerabilities. Moreover, FaCT simply reject programs that access data structures with secrets. Unlike FaCT, we let the compiler apply any optimizations it deems necessary, and then we use the ORIGAMI rules to ensure the resulting code satisfies TSCF under the MAP leakage model.

Program repair. Solutions like Raccoon [27], SC-ELIMINATOR [36] and [31] often modify programs at the IR level (i.e., they modify the front-end of the compiler). All static enforcement solutions assume that actors further ahead in the compilation and execution processes do not undo their modifications to the program. Other repair solutions, like oo7 [35] work at lower levels, but they protect programs against Spectre V1 attacks, and not against leakage caused directly by the program.

Our tool is mostly similar to Raccoon, SC-ELIMINATOR and the solution proposed in [31] because we do static enforcement; however, ORIGAMI offers an advantage over each of these three solutions. The tool in [31] does not repair programs with respect to the MAP leakage model (they repair with respect to the baseline model), the rules proposed by SC-ELIMINATOR are unsound in the context of the MAP leakage model, and while Raccoon does offer some protection against the MAP leakage model, the performance overhead has a geometric mean of 16x, while ORIGAMI has a performance overhead of 1.23x. Like Raccoon and SC-ELIMINATOR, ORIGAMI does not repair recursive calls, does not offer security guarantees over side-effects (e.g. I/O system calls), and does not repair programs whose timing side-channel vulnerability stems from the usage of functions whose execution time depends on their parameters (i.e., vulnerable in the OS leakage model).

Spectre Attacks: Spectre V1 attacks [16] rely on the speculative execution of a gadget of the form $x := B[A[s]]$ to leak secrets into the cache. Under the MAP leakage model, $x := B[A[s]]$ reveals $A[s]$ when the program **if** $s < size_A$ **then** $x := B[A[s]]$ executes, so if some secret value k is speculatively accessed via the $A[s']$, and then loaded in memory by $B[A[s']]$, the attacker can reverse engineer the value of $A[s']$, which is k. Although similar to the programs that ORIGAMI repairs, ORIGAMI does not defend against Spectre-style attacks, since its security guarantees only apply for non-speculative behaviours of the program. To repair programs that exhibit speculative leaks, we recommend to use a methodology that finds speculative leaks (e.g. [11]) or a methodology that prevents speculative leaks by cutting

dataflows from sensitive sources to speculative sinks (e.g. BLADE [34]).

VII. CONCLUSION

Although ORIGAMI offers a sound approach to enforcing timing side-channel freedom (TSCF) with respect to the memory access pattern (MAP) leakage model, there is still a long way to go until true TSCF is achieved. Most notably, ORIGAMI does not repair programs that are vulnerable with respect to the operand sensitive leakage model (OS), but adds another layer of security by repairing programs that are safe with respect to the baseline leakage model, but unsafe with respect to MAP leakage.

While we share the philosophy of [29], and we expect compilers to take over the task of enforcement of TSCF instead of the programmer, we believe that as long as hardware does not offer strong TSCF guarantees which can be depended on and rightfully used by compilers, any software based solution for TSCF enforcement, including ORIGAMI, will remain conditional, and thus not entirely reliable. Nevertheless, while this type of hardware is available, software-based solutions can mitigate timing side-channel vulnerabilities.

Our tool and all experimental data are readily available in an anonymous virtual machine for reproduction and further research: https://drive.google.com/file/d/1Y3xy_tXBMpRT95qw3zH_Nbx8LwLCmBy4/view?usp=sharing.

REFERENCES

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1807–1823, New York, NY, USA, 2017. Association for Computing Machinery.

[2] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, August 2016. USENIX Association.

[3] Jean-Philippe Aumasson. Cryptocoding, August 2019.

[4] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.*, 78(7):796–812, July 2013.

[5] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. *IACR Cryptol. ePrint Arch.*, 2019:1393, 2019.

[6] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018.

[7] Tiyash Basu, Kartik Aggarwal, Chundong Wang, and Sudipta Chattopadhyay. An exploration of effective fuzzing for side-channel cache leakage. *Software Testing, Verification and Reliability*, 30(1):e1718, 2020. e1718 stvr.1718.

[8] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), August 2011.

[10] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 917–934, Vancouver, BC, August 2017. USENIX Association.

[11] 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 *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 913–926, New York, NY, USA, 2020. Association for Computing Machinery.

[12] 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 *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 174–189, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying the information leakage in cache attacks via symbolic execution. *ACM Trans. Embed. Comput. Syst.*, 18(1), January 2019.

[14] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 431–446, Washington, D.C., August 2013. USENIX Association.

[15] S. He, M. Emmi, and G. Ciocarlie. ct-fuzz: Fuzzing for timing leaks. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 466–471, 2020.

[16] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[17] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 564–580, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[18] Dexter Kozen and Maria-Christina Patron. Certification of compiler optimizations using kleene algebra with tests. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 568–582, Berlin, Heidelberg, 2000. Springer-Verlag.

[19] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. Association for Computing Machinery.

[20] Libtom. libtomcrypt.

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[22] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 51–65, 2013.

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[24] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. Baltimore, MD, July 2005. USENIX Association.

[25] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728, 2017.

[26] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in f*. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[27] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., August 2015. USENIX Association.

[28] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference*

on Compiler Construction, CC 2016, pages 110–120, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 1–15, 2018.

[30] Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. Guarded kleene algebra with tests: Verification of uninterpreted programs in nearly linear time. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[31] Luigi Soares and Fernando Magno Quintão Pereira. Memory-safe elimination of side channels. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 200–210, 2021.

[32] The GNOME Project. gdk3 keyboard handling.

[33] Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi. Quantitative mitigation of timing side channels. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 140–160, Cham, 2019. Springer International Publishing.

[34] 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. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[35] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[36] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 15–26, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.