# Structured Leakage and Applications to Cryptographic Constant-Time and Cost

Gilles Barthe
MPI-SP
Bochum, Germany
IMDEA Software Institute
Pozuelo de Alarcón, Spain
gbarthe@mpi-sp.org

Benjamin Grégoire
Université Côte d'Azur, Inria
Sophia Antipolis, France
benjamin.gregoire@inria.fr

Vincent Laporte
Université de Lorraine, CNRS, Inria, LORIA
F-54000 Nancy, France
vincent.laporte@inria.fr

Swarn Priya
Université Côte d'Azur, Inria
Sophia Antipolis, France
swarn.priya@inria.fr

## ABSTRACT

Many security properties of interest are captured by instrumented semantics that model the functional behavior and the leakage of programs. For several important properties, including cryptographic constant-time (CCT), leakage models are sufficiently abstract that one can define instrumented semantics for high-level and low-level programs. One important goal is then to relate leakage of source programs and leakage of their compilation—this can be used, e.g. to prove preservation of CCT. To simplify this task, we put forward the idea of structured leakage. In contrast to the usual modeling of leakage as a sequence of observations, structured leakage is tightly coupled with the operational semantics of programs. This coupling greatly simplifies the definition of leakage transformers that map the leakage of source programs to leakage of their compilation and yields more precise statements about the preservation of security properties. We illustrate our methods on the Jasmin compiler and prove preservation results for two policies of interest: CCT and cost.

## CCS CONCEPTS

• **Security and privacy** → **Logic and verification**; **Formal methods and theory of security**; • **Theory of computation** → **Semantics and reasoning**.

## KEYWORDS

Secure Compilation, Cryptographic Constant-Time, Cost

**ACM Reference Format:**
Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3484761

## 1 INTRODUCTION

Modern compilers are designed to carry out an aggressive program optimizations while respecting the input-output behavior of programs. In simple settings, where behaviors are modelled as execution traces, compiler correctness, is thus stated as an inclusion between the set of traces of the target program and the set of traces of source programs. However, this approach suffers from three shortcomings in a security context: first, many common security properties are hyperproperties [12], i.e. sets of sets of traces, rather than properties, i.e. sets of traces; in particular, information flow properties, which cover a broad range of applications are relational properties, i.e. sets of pairs of traces. Second, several security properties of interest, including popular notions of side-channel resistance are modelled by an instrumented semantics that collects (an abstraction of) the adversarially visible physical leakage. Third, the inclusion of instrumented traces fail for most common compiler optimizations, e.g. register allocation and dead code elimination that may add, modify or remove atomic leakages. These shortcomings are not purely theoretical, as documented by multiple security vulnerabilities caused by popular compilers; see, e.g. [14, 20]. To address these shortcomings; researchers have developed the foundations of secure compilation, where compilers are required to preserve both the functional behavior and the security of programs; these studies often consider broad classes of security properties and are not tied to specific compiler passes. In parallel, other works have explored compiler preservation [6, 7] and compiler-based mitigations [10] for cryptographic constant-time (CCT), a popular software-based countermeasure to protect cryptographic implementations against devastating cache-based timing side-channel attacks.

*Contributions.* The main contribution of this paper is a novel approach for proving preservation of non-functional properties, and in particular CCT. Our approach is based on the following ideas:

**Structured leakage** We model leakage using a dedicated data structure that collects atomic leakages. Our new data structure is closely aligned with the operational semantics of programs, a key benefit over the flat list structure used in prior work;

**Leakage transformers** We define a language of leakage transformers, that transform leakage of source programs into leakage of target programs. Although our language of leakage transformers is simple; yet we can define leakage transformers for many common optimizations. A key benefit of leakage transformers is that they yield an algorithm for computing the leakage of target programs from leakage of source programs.

Leakage transformers are naturally endowed with a rigorous definition of correctness: specifically, a leakage transformer $\tau$ is correct for a source program, if for every set of inputs, we have $[\![\tau]\!]^{\ell} = \ell'$, where $\ell$ represents the leakage of the source program on the chosen inputs, $[\![\tau]\!]^{\cdot}$ interprets the algorithmic description of the transformer $\tau$ as a function from leakage to leakage, and $\ell'$ represents the leakage of the compiled program on the chosen inputs. Surprisingly, many leakage transformers achieve this strong notion of correctness. This provides an effective method for proving preservation of CCT and other non-functional properties.

To illustrate the benefits of our method, we implement our approach on top of the Jasmin framework [2, 3] for high-assurance and high-speed cryptography. The Jasmin framework is a natural target for our approach for three reasons. First, Jasmin puts a strong emphasis on cryptographic constant-time and efficiency, two prime examples of non-functional properties. Second, verification of Jasmin programs (CCT, functional correctness, cryptographic strength, and cost) is carried out at source-level, to benefit from the verification-friendly nature of the Jasmin language. Third, the Jasmin compiler comes with a mechanized proof (in the Coq proof assistant) that generated assembly programs have the same behavior as their source programs, but there is no mechanized proof of preservation of CCT, and no prior study of the impact of compilation on cost. Using leakage transformers, we overcome these two shortcomings: we obtain a formal proof that the Jasmin compiler preserves CCT, and a certified algorithm to compute the cost of compiled assembly programs from the cost of source Jasmin programs[1]. A surprising aspect of our certified cost transformer is that it yields an exact cost rather than an upper bound for the generated programs (for all transformations except loop unrolling, where our transformer yields an upper bound); to our best knowledge, we are the first to provide formally verified exact cost transformers for a realistic compiler.

In summary, our main contributions include:

- the definition of structured leakage and leakage transformers;
- formal proofs of correctness of leakage transformers for all the passes of the Jasmin compiler;
- a proof that the Jasmin compiler preserves CCT;
- a certified algorithm for computing the cost of assembly programs from the cost of Jasmin programs.

All results presented in this paper have been formally verified using the Coq Proof Assistant. The complete development is provided as supplementary material[2].

[1]As shall be explained shortly, our cost model is abstract and only provides an estimate of the efficiency of the generated assembly. In particular, it does not provide cycle-accurate estimates of the program's true execution cost.

[2]Supplementary material is available for download at: https://github.com/jasmin-lang/jasmin/tree/constant-time

## 2 METHODOLOGY

This section outlines our methodology and its applications to cryptographic constant-time and cost.

### 2.1 Compiler correctness

Certified compilers are high-assurance compilers that come with a machine-checkable proof that the compiler is correct, i.e. preserves the behavior of programs. The statement of compiler correctness relies on operational semantics, which formalizes the execution of source and assembly programs. For the purpose of this section, we assume given big-step semantics for source and target programs; these semantics are expressed by judgments of the form $p : s \Downarrow s'$ (resp. $\overline{p} : s \Downarrow s'$ ), stating that execution of source program $p$ (resp. target program $\overline{p}$) on initial state $s$ terminates with final state $s'$. Using this notation, compiler correctness is informally stated as: for all source programs $p$ with compilation $\overline{p}$, and for all states $s$ and $s'$,

$$p : s \Downarrow s' \implies \overline{p} : s \Downarrow s'.$$

Note that our informal definition assumes that source and target programs operate over the same state space; the assumption simplifies the discussion, but our approach applies (and is formally verified) to the more general setting where source and target programs operate over different state spaces.

### 2.2 Instrumented semantics

Many properties of interest are expressed relative to an instrumented semantics, which tracks visible effects of program executions —since one main motivation of our work is protection against side-channel attacks, from now on we use the term leakage generically to refer to program's effects. The instrumented semantics is based on a leakage model describing what is leaked during program execution, leading to a judgment of the form $p : s \Downarrow_{\ell} s'$, stating that executing program $p$ on initial state $s$ yields a final state $s'$ and leaks $\ell$.

Unfortunately, compiler correctness does not readily extend to instrumented semantics. Indeed, for most leakage models and compilers of interest, source and target programs have different leakage. However, one can meaningfully extend the statement of compiler correctness by requiring the existence of a function $F$ that transforms leakage of $p$ into leakage of $\overline{p}$. There are many ways to exhibit such a function $F$; the approach taken in this paper is that the function $F$ is generated by the compiler. Under this approach, one can define instrumented compiler correctness as: for all source programs $p$ with compilation $\overline{p}$ and producing leakage transformer $F$, and for all states $s$ and $s'$,

$$p : s \Downarrow_{\ell} s' \implies \overline{p} : s \Downarrow_{F(\ell)} s'$$

Our notion assumes that $F$ does not depend on the initial state of the program. This holds for most common compiler passes. However, in some cases, the definition of the function $F$ may depend on the initial state, requiring additional steps. This shall be explained later.

### 2.3 Cryptographic constant-time

Cryptographic constant-time (CCT) is a software countermeasure against cache-based timing attacks, an effective class of side-channel attacks that exploit the latency between cache hits and cache misses to retrieve cryptographic keys and other secrets from program execution. The two rules of CCT programming are:

- do not branch on secrets;
- do not perform secret-dependent memory accesses.

These rules are very effective: in particular, they guarantee that a victim program is immune against cache-based timing attacks from a powerful low-level adversary with control over the cache and the scheduler, provided the victim program and the adversary execute in different processes, and memory isolation between processes are guaranteed [5]. Moreover, it has been proved experimentally that one can implement efficient cryptographic libraries that follow the CCT discipline.

CCT is an instance of observational non-interference [7], a general class of information flow policies that ensure that programs do not leak their secrets through observable leakage. As such, the CCT property is formalized using a leakage model such that control-flow instructions leak the branch in which they jump, and memory-accessing instructions leak the address (not the value) being accessed. In addition to the leakage model, the CCT property is stated relative to security declarations that tag the memory's public and private parts. The declarations induce an equivalence relation on states; it is denoted by $\sim$ and called indistinguishability: informally, two states are indistinguishable if they only differ in their private parts. The CCT property for a program $p$ is then stated as: for all initial states $s_1$ and $s_2$,

$$\left.\begin{array}{l} p : s_1 \Downarrow_{\ell_1} s_1' \\ p : s_2 \Downarrow_{\ell_2} s_2' \end{array}\right\} \implies s_1 \sim s_2 \implies \ell_1 = \ell_2.$$

Under this formalization, preservation of CCT for a program $p$ with compilation $\overline{p}$ is stated as: for all initial states $s_1$ and $s_2$,

$$\left.\begin{array}{l} p : s_1 \Downarrow_{\ell_1} s_1' \\ p : s_2 \Downarrow_{\ell_2} s_2' \\ \overline{p} : s_1 \Downarrow_{\overline{\ell}_1} s_1' \\ \overline{p} : s_2 \Downarrow_{\overline{\ell}_2} s_2' \end{array}\right\} \implies s_1 \sim s_2 \implies \ell_1 = \ell_2 \implies \overline{\ell}_1 = \overline{\ell}_2.$$

The definition readily extends to instrumented compilers that output leakage transformers.

Theorem 2.1 (Informal). *Any compiler that verifies instrumented correctness preserves constant-time.*

## 2.4 Cost

Programmers often rely, specially in the initial stages of development, on a cost model that provides a crude estimate of the efficiency of their code. Arguably one of the simplest cost models is the instruction counting model, which tracks how many times each instruction is executed in a program run. The instruction counting model is the basis of many approaches for computing upper bounds on the cost of the program. These approaches are generally developed for source programs. However, our framework offers a means to transfer the results of the analysis to target programs.

Specifically, note that for many cost models of interest, including the instruction counting cost model, it is possible to compute the cost of an execution as a function of its leakage, i.e. $\kappa = \text{tocost}\ (\ell)$, where $p : s \Downarrow_\ell s'$. Therefore, any function $F$ that correctly transforms the leakage of $p$ satisfies $\overline{\kappa} = \text{tocost}\ (F(\ell))$. Thanks to the explicit representation of $F$, it is, therefore, possible to compute the cost of a target Jasmin program from analyzing the source program. (Our description suggests a way to compute the cost of the generated program from the leakage of the source program; we later explain why the cost, rather than the leakage, of the source program, suffices for this purpose.)

Theorem 2.2 (Informal). *Any compiler that verifies instrumented correctness correctly transforms cost.*

Strikingly, this approach is precise, i.e. for most optimization passes, one can compute the cost of the target program from the cost of the source program. In other words, the cost of the target program is exact if the cost of the source program is exact. Moreover, the cost of the target program is a sound overapproximation if the cost of the source program is a sound approximation. Note that in all cases, the cost is understood in the context of the cost model rather than a concrete value based on the number of execution cycles. Nevertheless, and although this is beside the point of this paper, even a simple model like the instruction counting model gives a coarse but meaningful estimate of the number of execution cycles for the class of programs we consider.

## 3 BACKGROUND ON JASMIN

The Jasmin framework was introduced in [2] and further developed in [3]. Two of its main components are the Jasmin language and the Jasmin compiler, which we briefly describe below. Other main components are the verification tools for functional correctness, CCT, and cryptographic strength; however, they are not directly relevant to the work presented here, so we refer the interested reader to [2, 3].

The Jasmin language is a verification-friendly programming language that supports "assembly in the head". It combines high-level facilities —that simplify the writing, reading, analysis and verification of programs— with tight control over low-level details of the generated assembly —that empowers the programmers to construct aggressively optimized code. For instance, programmers choose whether values stay in registers or are stored in the stack memory (through the reg and stack annotations), decide which loops are unrolled at compile-time, and have direct access to assembly instructions through the use of intrinsics. High-level constructs include variables, functions —that are fully inlined—, structured control flow, compile-time computations, support for vectorized operations (SIMD), and arrays. Register arrays are particularly convenient in combination with unrolled loops and compile-time computations of array indices. Stack arrays dramatically simplify static analyses and manual verification of programs: since array addresses cannot be involved in pointer arithmetic, arrays with different names, neither alias nor overlap.

Source Jasmin programs compute over several kinds of values: boolean (written *tt* and *ff*), unbounded integers (for compile-time computations only), machine integers of various sizes (from 8 to 256-bit), and arrays of machine integers. Strikingly, arrays are a first-class value: functions can take them as arguments and return them as results. Arguments are passed by value: an array passed to a function is not modified unless it is also returned by this function. This considerably simplifies the reasoning about program behaviors. Of course, this is just a convenience for the programmers and

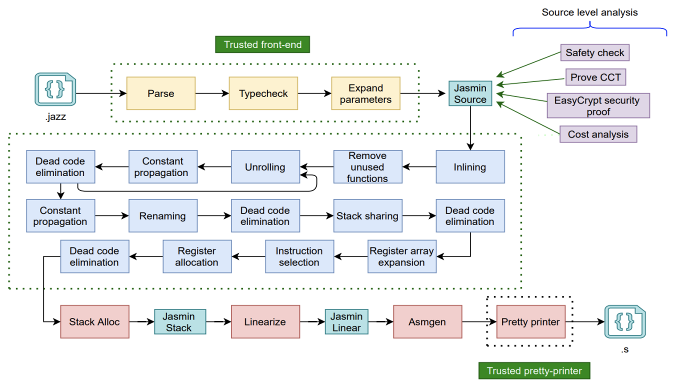

**Figure 1: Jasmin architecture**

| | | |
|---|---|---|
| $e \in$ Expr ::= | $x$ | variable |
| | $\mid\ c$ | constant |
| | $\mid\ a[e]$ | array access |
| | $\mid\ [e]$ | memory load |
| | $\mid\ \mathrm{op}(e,\ldots,e)$ | operator |
| $d \in$ Lval ::= | $x$ | variable |
| | $\mid\ a[e]$ | array store |
| | $\mid\ [e]$ | memory store |
| $i \in$ Instr ::= | $d := e$ | assignment |
| | $\mid$ if $e$ then $i$ else $i$ | conditional |
| | $\mid$ while $e$ do $i$ | while loop |
| | $\mid\ \{i;\ldots;i\}$ | sequencing |

$a \in \mathcal{A}$ ranges over array variables; $x \in \mathcal{X}$ ranges over scalar variables

**Figure 2: Syntax of programs**

verification tools: the compiler ensures that no copy happens at run-time.

The compiler produces efficient assembly code for x86_64 platforms (other platforms are under development). The overall compilation chain is presented in Figure 1. It is formally verified for correctness in Coq, except the front-end (parsing, type checking, and expansion of parameters) and the assembly pretty-printer that are trusted. Throughout the compilation, five different intermediate representations (IR) are used. At the highest-level, the Jasmin-source language is verification friendly: it is structured and has a clean semantics. Formal verification of Jasmin programs is done on this intermediate representation. The middle-end manipulates the Jasmin IR: it has the same syntax (presented in Figure 2) as Jasmin-source but a more flexible semantics that allows more optimizations. The last pass of the middle-end uses Jasmin-stack as output: this IR again has the same structured syntax but also features an explicit stack pointer. The back-end outputs unstructured IR: Jasmin-linear with labels and gotos after linearization, and assembly at the end. In this paper, we focus on the compiler middle-end; therefore, we only present the semantics of Jasmin and not of the lower languages. Of course, our implementation of leakage transformers carries all the way to assembly.

In order to remain predictable, the compiler does not perform optimizations that potentially affect efficiency, e.g. instruction scheduling or register spilling. For the latter, the Jasmin compiler instead performs a weaker form of register allocation that fails when too many registers are used. Nonetheless, the compilation chain is complex and features many passes that can dramatically impact leakage and cost; among them: inlining turns a program into a single function, unrolling fully unrolls all for loops, constant propagation simplifies expressions and conditionals, dead-code elimination removes some redundant computations, instruction selection replaces high-level operators by sequences of machine instructions, stack-allocation turns accesses to variables into memory operations, linearization fixes the program layout and introduces jumps.

## 4 ILLUSTRATIVE EXAMPLES

Theorems 2.1 and 2.2 highlight the benefits of instrumented compiler correctness. However, two challenges must be addressed in order to realize these benefits. First, one must define the leakage transformer $F$. Second, applications such as cost require an algorithmic description of $F$, in order to compute the cost of the target program. We address these challenges by using structured leakage and a syntax for describing leakage transformers. This section introduces deliberately simple examples that illustrate our representations of leakage and leakage transformers and their benefits. We first consider expressions, then turn to instructions.

*Expressions.* Figure 3 introduces two code snippets representing the source and target code for addition operations and their associated leakages. Figure 3 also presents the leakage transformer that will be produced during this transformation. The first addition operation adds 0 to the value present at index 0 in the array a and the second addition adds 1 to the result obtained from the first addition. The compiler knows statically that the result of the first operation will be $a[0]$; hence the target code is just one addition operation with operands $a[0]$ and 1. The leakage for $(0+a[0])+1$ is $((\bullet, (\bullet, [0])), \bullet)$ representing that evaluation of a constant produces no leakage and an array access leaks the index accessed. The leakage for the compiled expression $a[0] + 1$ is $((\bullet, [0]), \bullet)$. The compiler produces leakage transformer $(\pi_2, \mathrm{id})$ where $\pi_2$ projects the leakage at index 2 from the source leakage $(\bullet, (\bullet, [0]))$ and id preserves the leakage. If the leakage produced by the addition operation was represented as a concatenation of its sub-parts' leakages, it would be difficult to project at the corresponding index as concatenating with an empty leakage returns the original leakage. The flattened source leakage of the above example will be the concatenation of $\bullet$, $[0]$ and $\bullet$, which will get reduced to $[0]$. From the flattened list, it is hard to identify the leakages belonging to the sub-parts.

*Instructions.* Figure 4 presents a conditional instruction with a $tt$ guard, that is reduced to its then branch after compilation. This kind of transformation is carried out when compiler statically knows the value of boolean condition. The leakage for conditional instruction at the source level is $\mathrm{if}_{tt}\,(\bullet, (\bullet, [0]) := \bullet)$. The structure of the source leakage is closely aligned with the structure of the conditional instruction, with $tt$ indicating that the boolean condition is satisfied and $(\bullet, [0]) := \bullet$ indicating that the then branch is an assignment

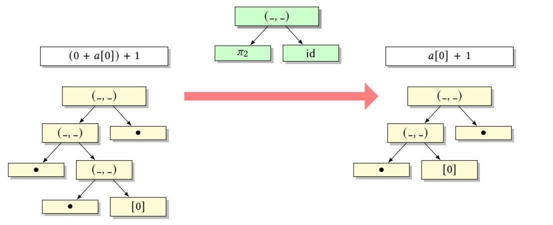

Figure 3: Example: Structured leakage for expression

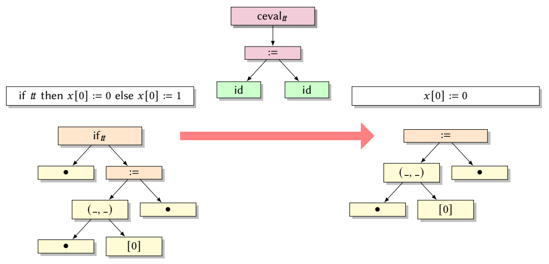

Figure 4: Example: Structured leakage for conditional

instruction. The target leakage is $(\bullet, [0]) := \bullet$, which gives us information, the conditional instruction is reduced to an assignment instruction. The leakage transformer produced during this transformation is ceval$_{tt}$ (id := id), where $tt$ indicates the branch taken and id := id transforms the leakage for the then branch from the source. Designing the leakage transformer was straightforward because of the structured notion of leakages. If the leakage generated from conditional instruction was a concatenation of its sub-parts then it would be hard to detect, which part in the list, belongs to the then or else branch.

## 5 INSTRUMENTED CORRECTNESS FOR JASMIN COMPILER

This section details how we apply our methodology for the Jasmin compiler. We present our representation of leakage and our instrumented operational semantics, and our syntax for leakage transformers. Finally, we discuss instrumented correctness for some of the Jasmin compiler passes. For the clarity of presentation, we provide a simplified treatment of the semantics of leakage transformers.

### 5.1 Instrumented semantics

In this section, we introduce the formal notion of structured leakages, and the instrumented semantics of a core fragment of the Jasmin language shown in Figure 2—our Coq formalization is for the full language, including other forms of loops and procedure calls.

We distinguish between leakage $\ell_e$ for expressions and leakage $\ell$ for instructions. Figure 5 presents the syntax of structured leakages. As the notations suggest, the leakage's syntax is closely related to the syntax of programs and their semantics. In the case of expressions, $\ell_e$ can be $\bullet$, an array index $[z]$, a memory address $*p$ or a tuple of leakage $(\ell_e^1, \ldots, \ell_e^n)$. In the case of instructions, there is one constructor per semantic rule.

| | | | | | |
|---|---|---|---|---|---|
| $\ell_e$ ::= | $\bullet$ | empty | $\ell$ ::= | $\ell_e := \ell_e$ | assignment |
| \| | $[z]$ | index | \| | $\text{if}_b(\ell_e, \ell)$ | conditional |
| \| | $*p$ | address | \| | $\text{while}_t(\ell_e, \ell, \ell)$ | iteration |
| \| | $(\ell_e, \ldots, \ell_e)$ | sub-leakage | \| | $\text{while}_f(\ell_e)$ | loop end |
| | | | \| | $\{\ell; \ldots; \ell\}$ | sequence |

Figure 5: Syntax of structured leakages

Expression semantics:

$$\frac{}{c \downarrow^s_\bullet c} \qquad \frac{v = s(x)}{x \downarrow^s_\bullet v} \qquad \frac{e_i \downarrow^s_{\ell_e^i} v_i \quad v = \text{op}(v_1, \ldots, v_n)}{\text{op}(e_1, \ldots, e_n) \downarrow^s_{(\ell_e^1, \ldots, \ell_e^n)} v}$$

$$\frac{e \downarrow^s_{\ell_e} z \quad s(a) = t}{a[e] \downarrow^s_{(\ell_e, [z])} t[z]} \qquad \frac{e \downarrow^s_{\ell_e} p}{[e] \downarrow^s_{(\ell_e, *p)} s[p]}$$

Assignment semantics:

$$\frac{}{x := v \downarrow^s_\bullet s\{x \leftarrow v\}}$$

$$\frac{e \downarrow^s_{\ell_e} z \quad s(a) = t \quad t' = t\{z \leftarrow v\}}{a[e] := v \downarrow^s_{(\ell_e, [z])} s\{a \leftarrow t'\}} \qquad \frac{e \downarrow^s_{\ell_e} p}{[e] := v \downarrow^s_{(\ell_e, *p)} s\{p \leftarrow v\}}$$

Instruction semantics:

$$\frac{}{\{\} : s \Downarrow_{\{\}} s} \qquad \frac{i : s \Downarrow_{\ell_i} s_1 \quad \{c\} : s_1 \Downarrow_{\{\ell_c\}} s_2}{\{i; c\} : s \Downarrow_{\{\ell_i; \ell_c\}} s_2}$$

$$\frac{e \downarrow^s_{\ell_e} v \quad d := v \downarrow^s_{\ell_d} s'}{d := e : s \Downarrow_{\ell_d := \ell_e} s'}$$

$$\frac{e \downarrow^s_{\ell_e} b \quad c_b : s \Downarrow_{\ell_c} s'}{\text{if } e \text{ then } c_{tt} \text{ else } c_{ff} : s \Downarrow_{\text{if}_b(\ell_e, \ell_c)} s'} \qquad \frac{e \downarrow^s_{\ell_e} ff}{\text{while } e \text{ do } c : s \Downarrow_{\text{while}_f(\ell_e)} s}$$

$$\frac{e \downarrow^s_{\ell_e} tt \quad c, s \Downarrow_{\ell_c} s_1 \quad \text{while } e \text{ do } c : s_1 \Downarrow_{\ell_w} s_2}{\text{while } e \text{ do } c : s \Downarrow_{\text{while}_t(\ell_e, \ell_c, \ell_w)} s_2}$$

Figure 6: Instrumented semantics.

The instrumented semantics is produced from the original semantics by annotating the judgments with leakage. The semantic uses three judgments. The first, $e \downarrow^s_{\ell_e} v$, provides the semantic of the expression $e$ in the state $s$, it produces a leakage $\ell_e$ and a value $v$. The second, $d := v \downarrow^s_{\ell_e} s'$ provides semantics of assigning a value $v$ to a destination $d$ in a state $s$; it generates a new state $s'$ and some leakage $\ell_e$ (leakage for assignments is a subset of the one for expressions). The last judgment, $i, s \Downarrow_\ell s'$, provides semantics of instructions; it takes an instruction $i$ and a state $s$ and returns an instruction's leakage $\ell$ and a state $s'$. The three judgments are presented in Figure 6.

Informally a state is a pair of a memory (a mapping from addresses to values), and a valuation for variables (a mapping from variables to values). $s(x)$ is the value associated to $x$ in $s$ (which can be an array). $s[p]$ loads the value stored in memory at an address $p$.

When a value $v$ is an array, $v[i]$ denotes the value at index $i$ in this array. $s\{x \leftarrow v\}$ updates the value associated to $x$ with $v$ and $s\{p \leftarrow v\}$ writes $v$ in memory at address $p$.

The instrumented semantics of expressions is presented in Figure 6. Variables leak $\bullet$, i.e. a mark indicating that a variable has been evaluated. The evaluation of an array access $a[e]$ leaks a pair $(\ell_e, [z])$ where $\ell_e$ is the leakage corresponding to the evaluation of the index $e$ and $z$ is the value of $e$. Memory accesses work in the same way. Operators leak the tuple composed by the leakages of their arguments. A destination can be either a variable, an array destination or a memory destination. The semantic of assignment also generates leakages due to memory and array stores. It follows the same pattern as for expressions.

Except for the leakage, the non-instrumented rules are mostly standard; hence we only discuss the parts related to leakage. An assignment instruction produces a leakage $\ell_d := \ell_e$ composed of the leakage generated during the evaluation of the expression $e$ and the one generated during the evaluation of the assignment $d := v$. The leakage of a sequence is composed of the leakage of each of its components. For conditionals, the leakage is $\mathsf{if}_b(\ell_e, \ell_{c_b})$, so it contains the leakage $\ell_e$ generated by the evaluation of the condition, the value $b$ of the condition and the leakage $\ell_{c_b}$ generated by the evaluation of the taken branch. We have two rules for the evaluation of loop instructions. If the condition evaluates to false, the loop exits and the leakage is $\mathsf{while}_\mathsf{f}(\ell_e)$. Otherwise the leakage is $\mathsf{while}_\mathsf{t}(\ell_e, \ell_c, \ell_w)$, where $\ell_c$ is the leakage generated by the body and $\ell_w$ is the leakage obtained by iterating the loop. The instrumented semantics is deterministic, both with respect to states and with respect to leakages.

## 5.2 Leakage transformers

In order to instrument all compilation passes of the Jasmin compiler from source to assembly, we have introduced three different languages of leakage transformers. The one that is presented in this paper covers all passes but two: the linearization and assembly generation passes use dedicated languages of leakage transformers, as the leakages for linear and assembly programs are fundamentally different than the leakage of structured Jasmin programs.

The (partial) syntax of leakage transformers is shown in Figure 7. Naturally, the syntax distinguishes between leakage transformers for expressions ($\tau_e$), instructions ($\tau$) and accessed addresses ($\tau_s$). Informally, leakage transformers are functions that take a source leakage and return a target leakage. This intuition is made formal using three interpretations $[\![\tau_e]\!]_e^{\ell_e}$ (for expressions), $[\![\tau]\!]^{\ell}$ (for instructions) and $[\![\tau_s]\!]_s^{v_{\mathrm{sp}}}$ (for accessed addresses) where $v_{\mathrm{sp}}$ is the value of stack pointer. Their formal definitions are provided Figure 8.

We start by explaining the syntax and semantics of leakage transformers for instructions. First, observe that many (but not all) compilation passes are structure-preserving and are defined recursively on the structure of the program. For such passes, the compilation of instructions consists of applying a transformation on its sub-expressions and sub-instructions. In this case, the leakage of the resulting instruction will also have the same structure as the source leakage, and only its sub-components will be modified. To account for these cases, the syntax of leakage transformer includes a constructor per instruction. This constructor will recursively traverse leakage without modifying its structure and only applying the transformation on the sub-leakages. The sub-transformations are themselves described using leakage transformers. For example, the leakage transformer $\tau_d := \tau_e$ will expect a leakage of the form $\ell_d := \ell_e$ and will apply its sub-transformers to the sub-leakages so that the resulting leakage will be of the form $[\![\tau_d]\!]_e^{\ell_d} := [\![\tau_e]\!]_e^{\ell_e}$. For conditional instructions, the leakage transformer $\mathsf{if}(\tau_e, \tau_{tt}, \tau_{ff})$ is built from leakage transformers for the condition and for each branch. Notice that only $\tau_{tt}$ or $\tau_{ff}$ will be used to transform the leakage (depending on which branch will be taken, but this cannot be known at compile-time). It is the interpretation of $[\![\mathsf{if}(\tau_e, \tau_{tt}, \tau_{ff})]\!]$ that selects which leakage transformer should be used. The transformers $\tau; \tau'$ (sequence) and $\mathsf{while}(\tau_e, \tau)$ (loop) work similarly.

In addition, we have leakage transformers corresponding to a change in the control flow of a program. The first transformer of this kind is remove which is used when an instruction is removed, e.g. in dead-code elimination. Assume that we have a program of the form $i; c$, let $c'$ and $\tau$ be the code and leakage transformer obtained by the compilation of $c$. Assume that the compiler is able to statically prove that the instruction $i$ is redundant, then the compiler will remove it and the compilation of $i; c$ will be $c'$. The leakage of the $i; c$ (resp. $c'$) is of the form $\ell_i; \ell_c$ (resp. $\ell_{c'}$). In this case, the leakage transformer for $i; c$ can be remove; $\tau$. The remove will throw away $\ell_i$ and then $\tau$ will be used to transform $\ell_c$ into $\ell_{c'}$.

The transformer $\mathsf{ceval}_b$ is used when a conditional instruction is replaced by one of its branches. This is typically used when the compiler replaces an instruction if $e$ then $c_{tt}$ else $c_{ff}$ by $c_b$ when the value of $e$ is statically known to be equal to $b$.

We also have leakage transformers corresponding to loop unrolling and inlining of function calls. They are not described here for space reasons. Another kind of leakage transformer is when the compiler replaces one instruction by a sequence of instructions, as in the instruction-selection pass of the Jasmin compiler. We will provide more explanation of them in section 5.4.

We now turn to leakage transformers for expressions. Broadly speaking, these transformers work similarly. We have a leakage transformer that is used for recursion $(\tau_e^1, \dots, \tau_e^n)$, $\bullet$ produces the constant leakage $\bullet$, and id is used when the compiler does not modify the expression, so its leakage remains the same. We also use two other leakage transformers $\pi_i$ and $\tau_e^1 \circ \tau_e^2$. The first allows access to sub-leakage, and the second allows to compose leakage transformers. The following table illustrates their usage with small examples based on constant propagation on expressions:

| Expression source | Expression target | Leakage transformer | Leakage source | Leakage target |
|---|---|---|---|---|
| $0 \times e$ | $0$ | $\bullet$ | $(\bullet, \ell)$ | $\bullet$ |
| $0 + e$ | $e$ | $\pi_2$ | $(\bullet, \ell)$ | $\ell$ |
| $e_1 + e_2$ | $e_1' + e_2'$ | $(\tau_e^1, \tau_e^2)$ | $(\ell_1, \ell_2)$ | $(\ell_1', \ell_2')$ |
| $e_1 + e_2$ | $e_2'$ | $\pi_2 \circ \tau_e^2$ | $(\ell_1, \ell_2)$ | $\ell_2'$ |

In the second line, the sub-expression $e$ is not modified, so the leakage transformer is a projection. In the third line, the addition is kept, and both sub-expressions are transformed, so we use a structural leakage transformer to combine the leakage transformers corresponding to the sub-expressions. In the fourth line, the addition

| | | |
|---|---|---|
| $\tau_e ::= \bullet$ | | empty |
| $\mid$ id | | identity |
| $\mid \tau_e \circ \tau_e$ | | composition |
| $\mid \pi_i$ | | projection |
| $\mid (\tau_e, \dots, \tau_e)$ | | map |
| $\mid$ rev | | reverse |
| $\mid (\tau_e; \dots; \tau_e)$ | | sequence |
| $\mid C(\tau_s)$ | | constant addr |
| $\mid I(x \mapsto \tau_s)$ | | indexed addr |

| | |
|---|---|
| $\tau_s ::= \mathsf{cst}(p)$ | constant |
| $\mid$ sp | stack pointer |
| $\mid \tau_s + \tau_s$ | addition |
| $\mid \tau_s \times \tau_s$ | multiplication |
| $\mid x$ | variable |
| $\tau ::= \tau_e := \tau_e$ | assign |
| $\mid \mathsf{if}(\tau_e, \tau, \tau)$ | cond |
| $\mid \mathsf{while}(\tau_e, \tau)$ | while |
| $\mid \tau; \tau$ | sequence |
| $\mid$ remove | remove |
| $\mid \mathsf{ceval}_b\ \tau$ | cond-eval |
| $\mid \dots$ | $\dots$ |

**Figure 7: Leakage Transformers**

is removed (as in the second line), and the second sub-expression is recursively transformed, so we compose a projection with the leakage transformer for the sub-expression.

## 5.3 Formal statement

The correctness proof of leakage transformers is stated as follows. For each source program $p$, if the compilation succeeds and produces target program $\overline{p}$ and leakage transformer $\tau$, then for every instrumented execution of the source producing a leakage $\ell$ then the instrumented execution of the target program is defined and produces a target leakage, which is equal to the leakage obtained by applying the leakage transformer to the source leakage.

Theorem 5.1 (Instrumented correctness).

$$p : s \Downarrow_\ell s' \implies \overline{p} : s \Downarrow_{[\![\tau]\!]^\ell} s'.$$

The Jasmin compiler and its correctness proof are structured as a sequence of passes —depicted on Figure 1. Each pass is independently verified and comes with its dedicated correctness theorem and proof: we update them all to turn them into instrumented correctness. The modification boils down to proving that the function $[\![.]\!]^{\cdot}_e$ and $[\![.]\!]^{\cdot}$ compute the correct leakage based on the source leakage.

These passes modify the compiled program in various ways: they may preserve its structure or change it; they may even completely remove some instructions. The leakage transformers are designed to cover all these kind of transformations. We concisely designed the leak transformers so that a single leakage transformer can serve the purpose of multiple compiler passes.

Most of the passes preserve or reduce the leakage; they can be justified using the leakage transformers presented in Section 5.2. Notable exceptions are the two final passes (*linearization* and *assembly generation*); they produce unstructured programs (that do not follow the syntax described in this paper) and therefore require a specific set of leakage transformers. Nonetheless, even though the correctness proofs of these passes are a bit tricky; extending them to the instrumented semantics is relatively easier.

In the rest of this section, we put the emphasis on two special cases: *instruction selection*[3] and *stack allocation*.

[3]This pass is also referred to as *lowering* in the Jasmin literature.

Leakage transformers for expressions:

$$\frac{}{[\![\bullet]\!]^{\ell_e}_e = \bullet} \qquad \frac{}{[\![id]\!]^{\ell_e}_e = \ell_e} \qquad \frac{}{[\![\mathsf{rev}]\!]^{(\ell_1,\dots,\ell_n)}_e = (\ell_n, \dots, \ell_1)}$$

$$\frac{}{[\![\pi_i]\!]^{(\ell_1,\dots,\ell_n)}_e = \ell_i} \qquad \frac{[\![\tau_i]\!]^{\ell}_e = \ell'_i}{[\![(\tau_1; \dots; \tau_n)]\!]^{\ell}_e = (\ell'_1, \dots, \ell'_n)}$$

$$\frac{[\![\tau_i]\!]^{\ell_i}_e = \ell'_i}{[\![(\tau_1, \dots, \tau_n)]\!]^{(\ell_1,\dots,\ell_n)}_e = (\ell'_1, \dots, \ell'_n)} \qquad \frac{[\![\tau_1]\!]^{\ell_e}_e = \ell'_e}{[\![\tau_1 \circ \tau_2]\!]^{\ell_e}_e = [\![\tau_2]\!]^{\ell'_e}_e}$$

$$\frac{[\![\tau_s]\!]^{v_{\mathsf{sp}}}_s = p}{[\![C(\tau_s)]\!]^{\ell_e}_e = *p} \qquad \frac{[\![\tau_s[i/x]]\!]^{v_{\mathsf{sp}}}_s = p}{[\![I(x \mapsto \tau_s)]\!]^{[i]}_e = *p}$$

Transformers creating address leakage:

$$\frac{}{[\![\mathsf{cst}(p)]\!]^{v_{\mathsf{sp}}}_s = p} \qquad \frac{}{[\![\tau_{s_1} + \tau_{s_2}]\!]^{v_{\mathsf{sp}}}_s = [\![\tau_{s_1}]\!]^{v_{\mathsf{sp}}}_s + [\![\tau_{s_2}]\!]^{v_{\mathsf{sp}}}_s}$$

$$\frac{}{[\![\mathsf{sp}]\!]^{v_{\mathsf{sp}}}_s = v_{\mathsf{sp}}} \qquad \frac{}{[\![\tau_{s_1} \times \tau_{s_2}]\!]^{v_{\mathsf{sp}}}_s = [\![\tau_{s_1}]\!]^{v_{\mathsf{sp}}}_s \times [\![\tau_{s_2}]\!]^{v_{\mathsf{sp}}}_s}$$

Leakage transformers for instructions:

$$\frac{}{[\![\mathsf{remove}]\!]^{\ell} = \{\}} \qquad \frac{[\![\tau_d]\!]^{\ell_d}_e = \ell'_d \qquad [\![\tau_e]\!]^{\ell_e}_e = \ell'_e}{[\![\tau_d := \tau_e]\!]^{\ell_d := \ell_e} = \ell'_d := \ell'_e}$$

$$\frac{[\![\tau_e]\!]^{\ell_e}_e = \ell'_e \qquad [\![\tau_b]\!]^{\ell_b} = \ell'_b}{[\![\mathsf{if}(\tau_e, \tau_{tt}, \tau_{ff})]\!]^{\mathsf{if}_b(\ell_e, \ell_b)} = \mathsf{if}_b(\ell'_e, \ell'_b)}$$

$$\frac{[\![\tau_e]\!]^{\ell_e}_e = \ell'_e \qquad [\![\tau]\!]^{\ell_c} = \ell'_c \qquad [\![\mathsf{while}(\tau_e, \tau)]\!]^{\ell_w} = \ell'_w}{[\![\mathsf{while}(\tau_e, \tau)]\!]^{\mathsf{while}_{\mathsf{t}}(\ell_e, \ell_c, \ell_w)} = \mathsf{while}_{\mathsf{t}}(\ell'_e, \ell'_c, \ell'_w)}$$

$$\frac{[\![\tau_e]\!]^{\ell_e}_e = \ell'_e}{[\![\mathsf{while}(\tau_e, \tau)]\!]^{\mathsf{while}_{\mathsf{f}}(\ell_e)} = \mathsf{while}_{\mathsf{f}}(\ell'_e)}$$

$$\frac{[\![\tau_i]\!]^{\ell_i} = \ell'_i}{[\![\tau_1; \dots; \tau_n]\!]^{\ell_1; \dots; \ell_n} = \ell'_1; \dots; \ell'_n} \qquad \frac{[\![\tau]\!]^{\ell_b} = \ell'_b}{[\![\mathsf{ceval}_b\ \tau]\!]^{\mathsf{if}_b(\ell_e, \ell_b)} = \ell'_b}$$

**Figure 8: Semantics for leakage transformers**

## 5.4 Focus on instruction selection

Instruction selection (a.k.a. lowering) replaces high-level constructions by low-level instructions that are closer to the assembly. For example, the instruction $x := x + y$ will be transformed into $(\dots, \mathsf{CF}, \dots, x) := \mathsf{ADD}(x, y)$, i.e. the $+$ operator is replaced by a low level instruction ADD that will perform the addition (but also computes extra data like the carry flag). The assignment of the flags will create extra $\bullet$ leakage that has to be justified.

Similarly the instruction if $x < y$ then $c_1$ else $c_2$ (where $<$ is the unsigned comparison) is transformed into the sequence:

$$\begin{array}{l} \dots, \mathsf{CF}, \dots := \mathsf{CMP}(x, y); \\ \mathsf{if\ CF\ then}\ c_1\ \mathsf{else}\ c_2 \end{array}$$

From the point of view of the leakage transformation, this means that the leakage generated by the expression $x < y$ needs to be used to create the leakage for the CMP instruction. Therefore this pass relies on leakage transformers that can, on one hand split leakages into smaller parts and, on the other hand, construct fresh leakages from these parts and from the • constant.

This pass also uses the rev transformer which reverses the order of a sequence of leakage. It is used for some instructions that require their arguments to be evaluated in a specific order.

### 5.5 Focus on stack allocation

Stack allocation allocates some variables into the stack memory and replaces the corresponding accesses (read and write) by memory operations (load and store). It may thus create new leakage.

Given a scalar variable $x$ that is allocated at constant offset $o_x$ in the stack, a read from this variable will be compiled into the memory load $[\mathrm{sp}+o_x]$ where sp is the register containing the value of the stack pointer. At the source level, the leakage of $x$ is •, and it becomes $*(v_{\mathrm{sp}} + o_x)$ at the target level[4] (where $v_{\mathrm{sp}}$ is the value of the stack pointer). So the target leakage depends on the constant $o_x$ and also on the dynamic value of sp.

The case of an array variable $a$ allocated at constant offset $o_a$ is similar. The expression $a[e]$ is compiled into $[\mathrm{sp} + o_a + n \times e]$ where $n$ is the size of an array element. At source level $a[e]$ leak $[v_e]$, where $v_e$ is the value of $e$, and at the target level the leakage is $*(v_{\mathrm{sp}} + o_a + n \times v_e)$. $o_a$ and $n$ are statically known values, which are provided to the leakage transformer. So in this transformation, the target leakage will further depend on the value $v_e$ that can be recovered from the source leakage and on the dynamic value of sp.

These two examples show that in contrast to the other passes, the target leakage cannot be computed using the source leakage only. The initial value of the stack pointer is needed. This will have some consequence for the preservation of constant-time that is explained in section 6.1: the interpretation of the leakage transformer is parameterized by the value of the stack pointer, which must be considered as a *public* input.

To capture these transformations, the syntax of leakage transformers includes transformers that can construct new leakages, and their semantics depends on the value of the stack pointer. Fresh memory access leakages are introduced by the transformers $C(\tau_s)$ and $I(x \mapsto \tau_s)$. They are made of an expression $\tau_s$ that denotes the accessed address. In the case of $I(x \mapsto \cdot)$, the expression has a free variable (implemented using higher-order abstract syntax) that denotes the actual value of the index. The formal definition of the semantics of these transformers is given in Figure 8 and relies on an auxiliary function $\llbracket \tau_s \rrbracket_s^{v_{\mathrm{sp}}}$ which evaluates the (closed) expression $\tau_s$ given the actual value $v_{\mathrm{sp}}$ of the stack pointer.

To conclude this section, a small example program (shown on the left of Figure 9) illustrates the leakage transformers introduced in this pass. It stores two literal values in an array, writes a literal into a variable, and finally reads from the array. The leakage corresponding to the execution of this program is therefore (see Figure 10): { (•, [0]) := •; (•, [1]) := •; • := •; • := ((•, •), [1]) }.

As requested by the programmer, the array is allocated into the stack memory, whereas the other local variables stay in registers. The output of this compilation pass is shown on the right of Figure 9. Note that this pass happens after instruction selection and

```
fn foo() ⟶ reg u64 {
    stack u64[2] t;
    reg u64 p r;
    t[0] = 0;
    t[1] = 1;
    p = 0;
    r = t[(int)(p + 1)];
    return r; }
```

```
fn foo() ⟶ u64 {
    stack: 16
    [RSP + 0] = MOV(0);
    [RSP + 8] = MOV(1);
    RAX = MOV(0);
    RAX = MOV([RSP + (8 * (RAX + 1) + 0)]);
    return RAX; }
```

**Figure 9: Example program: source and after stack-allocation**

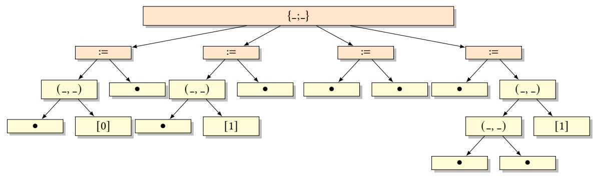

**Figure 10: Structured leakage for source code**

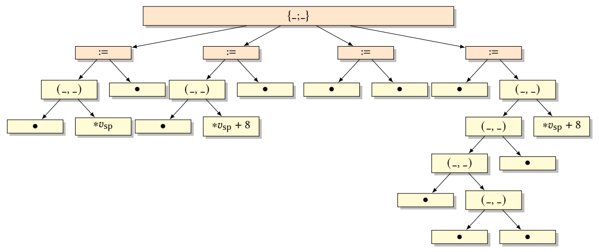

**Figure 11: Structured leakage for compiled code**

register allocation: variables p and r have been allocated to the RAX register and MOV instructions implement the assignments; for the sake of readability, a few type annotations have been hidden on the figure. The memory operations that are introduced by this pass compute addresses relative to the *stack pointer*, held in register RSP. Each index expression is transformed into a more complex address computation involving the actual index but also the stack pointer, constant offsets and scaling multiplications. The leakage corresponding to the evaluation of such an expression is transformed accordingly. The descriptions of these transformations, therefore, use the transformers producing address leakage that have been introduced above. They also use the vector of leakage transformers $(\tau_e; \ldots; \tau_e)$ that, applied to a single leakage, produces a vector of leakages, each component being the result of the application of the corresponding transformer.

The stack-allocation transformation applied to the example program yields the following leakage transformer[5]: $\{((\bullet, I(x \mapsto \mathrm{sp} + \mathrm{cst}\ x \times \mathrm{cst}\ 8))) := (\mathrm{id}); ((\bullet, I(x \mapsto \mathrm{sp} + \mathrm{cst}\ x \times \mathrm{cst}\ 8))) := (\mathrm{id}); (\mathrm{id}) := (\mathrm{id}); \mathrm{id} := ((\mathrm{id}, \mathrm{id}) \circ ((\bullet; \mathrm{id}); \bullet), I(x \mapsto \mathrm{sp} + \mathrm{cst}\ x \times \mathrm{cst}\ 8))\}$.

By instrumented correctness of this compilation pass (Theorem 5.1), the application of this transformer to the leakage shown above yields the leakage corresponding to the execution of the transformed program. Said leakage is as follows (see Figure 11): $\{(\bullet, *v_{\mathrm{sp}}) := \bullet; (\bullet, *(v_{\mathrm{sp}}+8)) := \bullet; \bullet := \bullet; \bullet := (((\bullet, (\bullet, \bullet)), \bullet), *(v_{\mathrm{sp}}+$

[4]The target leakage is in fact $(\bullet, *(v_{\mathrm{sp}} + o_x))$ to include the evaluation of the offset.

[5]This leakage transformer is represented as a tree in appendix A.

8))}, where $v_{\mathrm{sp}}$ is the value of the stack pointer. Note how the evaluation of the last address takes more (silent) steps than the evaluation of the corresponding array index and how the *index* leakages have been replaced by *address* leakages.

## 6 APPLICATIONS

This section describes how to leverage correct leakage transformers: we first, prove that the Jasmin compiler always preserves the constant-time property then show an application to cost analysis.

### 6.1 Constant-time preservation

The cryptographic constant-time (CCT) property is an effective counter-measure against practical side-channel attacks. It has been shown in practice that compilers do not always preserve this property and *introduce* vulnerabilities in otherwise secure programs. We formally prove that the Jasmin compiler is not subject to this issue: even though it transforms the control-flow and introduces memory accesses, it will neither remove counter-measures nor introduce sensitive information flows.

We have already described informally that preservation of CCT is a corollary of instrumented correctness (Theorem 5.1). However, that informal description elided a few practical issues that must be taken into account when implementing our methodology in the Jasmin compiler:

- some states are *unsafe*, i.e., the semantics of a program is a partial function;
- source and target languages (and states) are different;
- the interpretation of leakage transformers is parameterized by parts of the initial state (namely the value of the stack pointer);
- compiler correctness has side conditions (namely, there should be enough free memory in the initial target state to allocate the local variables).

We address these issues by using a definition of CCT that is meaningful even in the presence of unsafety and by suitably lifting the indistinguishability relation to target states.

As described in Section 2.3, the CCT property is parameterized by an indistinguishability relation on initial states and defined as follows: the leakage is the same for all indistinguishable states. We strengthen this definition to imply that indistinguishable states are safe to execute in constant-time programs.

*Definition 6.1 (Cryptographic constant-time).* A program $p$ is *cryptographic constant-time* w.r.t. the indistinguishability relation $\cdot \sim \cdot$ when the following holds:

$$\forall s_1\ s_2, s_1 \sim s_2 \implies \exists s_1'\ s_2'\ \ell, p : s_1 \Downarrow_\ell s_1' \land p : s_2 \Downarrow_\ell s_2'.$$

Moreover, in order to state and prove preservation of the CCT property, we must define the indistinguishability relation between target states. In the case of the Jasmin compiler, an initial source state is made of a memory $m$ and a list of values $\vec{v}$ (the arguments of the main function), whereas a target state is made of a memory $m$ and a register bank $r$. Fortunately, the source state can be computed from the target state: the memory is kept, and the values of the arguments are read in the appropriate registers. This computation is consistent with the compiler correctness statement. Therefore we can relate target states by relating the corresponding source states. The target leakage usually depends on the initial value of the stack pointer: we must thus require that this value is public, i.e., equal in indistinguishable states. Finally, to ensure that indistinguishable states are safe, we have to ensure that the side condition to the compiler correctness theorem is discharged. In a nutshell, this yields the following definition.

*Definition 6.2 (Indistinguishability of target states).* Given an equivalence relation $\cdot \sim \cdot$ between source states, its *lifting to target states* $\cdot \tilde{\sim} \cdot$ is defined as follows. We say that two target states $(m_1, r_1)$ and $(m_2, r_2)$ are indistinguishable, and note $(m_1, r_1) \tilde{\sim} (m_2, r_2)$, when all the following conditions hold:

- corresponding initial source states are indistinguishable, noted: $(m_1, \vec{v}_1) \sim (m_2, \vec{v}_2)$ (where $\vec{v}_1$, resp. $\vec{v}_2$, denotes the program arguments extracted from register bank $r_1$, resp. $r_2$);
- stack pointers agree: $r_1[\mathrm{sp}] = r_2[\mathrm{sp}]$;
- there is enough free stack space to allocate the local variables in both memories $m_1$ and $m_2$.

We can finally prove that our modified compiler *always* preserves the CCT property.

Theorem 6.3 (CCT-Preservation). *Given a source program $p$ that is CCT w.r.t. $\sim$, if the Jasmin compiler succeeds and produces a target program $\bar{p}$, then the target program is CCT w.r.t. $\tilde{\sim}$.*

### 6.2 Cost analysis

The run-time cost of a program — computational complexity, worst-case execution time (WCET), peak memory usage, etc. — crucially depends on low-level details hence on decisions made at compile-time: control-flow transformations and code layout, register spilling and memory layout of local variables, instruction selection and scheduling... Therefore a precise static cost analysis must be carried out near the end of the compilation pipeline (ideally on assembly code or even at binary level). However, the estimation of the run-time cost relies on loop bounds or other flow information (description of infeasible paths, for instance), either inferred by static analysis or provided by the programmer as annotations. In both cases, these flow facts are provided at the source level: programmers are more inclined towards annotating source code than target code, and static analyses are much more precise and efficient when they can rely on high-level abstractions from the source language.

This section describes how leakage transformers can reconcile these conflicting requirements: they are a sound way to transport source-level cost information down to the assembly level. More precisely, we first introduce a cost model as an abstraction of leakage and show how to deduce "cost transformers" from leakage transformers. Finally, we show how these cost transformers' soundness enables us to use the results of a source-level static analysis at the assembly level.

*6.2.1 Cost models.* In order to formally reason about the cost of program execution at either source or target level, we model it as the number of times each instruction is executed. In other words, a cost is a finite map from program points to natural numbers. For unstructured intermediate languages (linear, assembly), a program point is simply a position in the program text (i.e., a natural number); for structured languages, we define a language of paths to describe positions in the abstract syntax tree. Note that the usual order on natural numbers can be lifted pointwise to costs so that the set of costs forms a partial order.

The cost is defined by means of a function that *evaluates* a leakage trace into a cost map. Here again, the structure of the leakage is beneficial: it enables to perform this evaluation without looking at the program.

*Definition 6.4 (Cost).* Each intermediate language is equipped with an tocost $(\cdot)$ function that, given a leakage, computes a cost, i.e., a count for each program point. Given a execution $p : s \Downarrow_{\ell} s'$, its cost is tocost $(\ell)$.

*6.2.2 Cost transformers.* Program transformations found in compilers introduce, remove, or reorder instructions according to the program being compiled: they do not make up instructions out of the blue. Even though predicting how many times each instruction emitted by a compilation pass will be executed at run-time is usually not possible, the execution counts for target instructions can be related to execution counts for the corresponding source executions. More precisely, for most transformations found in the Jasmin compiler, the target costs can be precisely described by relating each target program point to one basic block of the source program. This link is to be interpreted as follows: "the instruction at this program point is executed in the target execution as many times as that basic block is executed in the source execution". For some compilation passes, unfortunately, we have to relax this property and interpret the predicted target count as an upper bound. As discussed in Section 7.3, this is not an issue in practice[6].

*Definition 6.5 (Cost transformer).* A cost transformer maps target program points to source basic blocks. It, therefore, enables the translation of a source-level cost into a target-level cost. A leakage transformer can be seen as a cost transformer by an interpretation function $\llbracket\cdot\rrbracket_{\kappa}^{\cdot}$; such an interpretation is sound when for all (source) leakage $\ell$ and matching leakage transformer $\tau$, the following holds (where $\sqsubseteq$ is a partial order on costs):

$$\text{tocost}\left(\llbracket\tau\rrbracket^{\ell}\right) \sqsubseteq \llbracket\tau\rrbracket_{\kappa}^{\text{tocost}(\ell)}. \qquad (1)$$

The interpretation of a leakage transformer as a cost transformer is defined from the leakage transformer only: it neither depends on the program nor on the compilation pass.

Cost transformers are monotone; therefore, they can be soundly composed. Indeed, given two leakage transformers $\tau_1$ and $\tau_2$ corresponding to two successive compilation passes, the following inequality holds:

$$\text{tocost}\left(\llbracket\tau_2\rrbracket^{\llbracket\tau_1\rrbracket^{\ell}}\right) \sqsubseteq \llbracket\tau_2\rrbracket_{\kappa}^{\llbracket\tau_1\rrbracket_{\kappa}^{\text{tocost}(\ell)}}.$$

It means that a sequence of two leakage transformer can be interpreted as a sound cost transformer by composing their interpretation[7].

The actual definition of the interpretation functions (for each of the three languages of leakage transformers described in this work) is tedious but unsurprising; the details, as well as the soundness proofs, can be found in the supplementary material.

[6]The study of more general languages for cost transformers that could improve the precision or cover more program transformations is left as further work.

[7]The result of this *semantic* composition may not be the most precise cost transformer, but we leave the study of *syntactic* composition of leakage transformers as future work.

```
fn poly1305_ref3(reg u64 out in len k) {
  reg u64 j;
  // ...
  while (len >=u 16) { // A
    // ...
    len -= 16;
  }
  if len >u 0 {
    // ...
    j = 0;
    while (j <u len) { // B
      // ...
      j += 1;
    }
    // ...
  }
  // ...
}
```

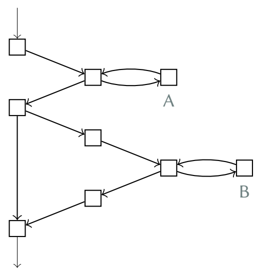

**Figure 12: Source code (left, excerpt) and target control-flow graph (right) of a MAC function (Poly1305)**

As already mentioned, the cost transformers for all but one pass are exact: the soundness relation (equation (1) above) holds even when the partial order on costs $\sqsubseteq$ is equality. For loop unrolling, however, it only holds for the slightly less precise pointwise ordering of counters (with natural numbers ordered as usual).

*6.2.3 Source-level cost analysis of target programs.* In order to illustrate the use of cost transformers, we have implemented a static analysis of the cost of Jasmin source programs. It infers linear relations between *counters*, auxiliary variables that are incremented at the beginning of basic blocks: their final values describe how many times each basic block has been executed. This method has already been used in the context of worst case execution time (WCET) analysis [19]. The value analysis that is part of the safety checker of Jasmin programs can be directly used to infer these relations. Notice that the analysis result, in particular, includes loop bounds.

The soundness of the cost transformer makes it possible to directly interpret these linear relations about source counters as linear relations between upper bounds on target counters. We illustrate this fact by a brief example.

The source code shown on the left of Figure 12 is excerpted from an implementation of the Poly1305 message authentication code (MAC). Given a message of arbitrary length and a 32-byte one-time key, it computes a 16-byte tag that can be used to authenticate the message. The message is processed in 16-byte chunks in a first loop; in case the message length is not evenly divisible by sixteen, the last bytes are finally processed in a second loop. For the sake of clarity, only this logic is reproduced in the code shown on the figure; in the actual implementation, the three control-flow constructions are spread in three distinct functions. The right of Figure 12 shows the (complete) control-flow graph (CFG) of this function at the end of the compilation. The leakage transformer produced by the compilation of this program interpreted as a cost transformer, proves (among others) that the basic-block labelled A (resp. B) in the CFG is executed as most as many times as the source basic-block equally labelled.

The source-level cost analysis infers the following properties, where $A$ (resp. $B$) is the number of times that the first (resp. second) loop body is executed, and $L$ is the message length (initial value of the variable len): $0 \leqslant A$, $0 \leqslant B \leqslant 15$, and $L = 16 \cdot A + B$.

The soundness of the cost transformers implies that these properties also apply at the assembly level (where $A$ and $B$ are upper bounds rather than exact counts). Using these properties, one can compute *at assembly level* a more basic notion of cost, for instance, the number of executed instructions.

## 7 EVALUATION

In this section, we evaluate our methodology in terms of proof effort, compile-time overhead, run-time overhead and precision of the source-level reasoning. We study the following questions.

- How much does the Jasmin compiler (programs & proofs) need to be modified and expanded to support our methodology?
- How much compile-time overhead is incurred by the generation of explicit leak-transformers?
- Is the code generated by our modified compiler different than before modifying the compiler?
- How precise is a cost analysis (of the target program) performed at the source level?

### 7.1 Proof effort

The Jasmin compiler (branch master) features sixteen compilation passes that are implemented or validated in Coq; they manipulate five different intermediate languages. Three languages have the same syntax; three languages have the same semantics for expressions. There are roughly 30 thousands lines of Coq.

In order to reason about non-functional properties like cryptographic constant-time and cost, all semantics have been instrumented with leakages as described in Section 5. Accordingly, all passes are modified to precisely describe how they transform the leakage. There is a single pass that preserves leakages (elimination of dead functions) for which there are no modifications to the implementation of the program transformation. Note that for passes that are implemented as an external oracle and validated in Coq, the validator infers the correct leakage transformer: the program analyses and transformations that are implemented in OCaml have not been modified at all.

The correctness statements for each pass have been strengthened, and their proofs updated accordingly. This is tedious but relatively straightforward. The main theorems presented in Section 6 are stated once for the compiler as a whole: they are simple corollaries of the correctness theorem. Their proofs are, therefore a few lines long. Globally, the changes made to the Coq files modify 5 thousands lines and add 6 thousands new lines (i.e., a 20 % increase).

The definitions and proofs related to cost and cost transformers are built on top of the leakages and leakages transformers only: they are independent of the number and complexity of the compilation passes. In particular, if the compiler is extended with new passes, no changes are required to this part. Only extensions of the language of leakage transformers would need to be reflected on the cost transformers.

**Table 1: Compilation times (s) of selected implementations of cryptographic primitives with (LT) and without (Ref.) computation of leak-transformers**

| Name | Ref. (s) | LT (s) |
|---|---|---|
| xxhash64 | 0.06 | 0.06 |
| poly1305 (ref) | 0.06 | 0.06 |
| gimli (AVX2) | 0.09 | 0.11 |
| chacha20 (ref) | 0.16 | 0.18 |
| poly1305 (AVX2) | 0.29 | 0.32 |
| gimli (ref) | 0.8 | 0.9 |
| bash (AVX2) | 2.4 | 2.6 |
| blake2b | 2.8 | 3.0 |
| chacha20 (AVX2) | 3.9 | 4.0 |
| bash (ref) | 6.5 | 7.4 |
| curve25519 | 7.6 | 8.3 |

### 7.2 Compiler behavior

When compiling a program, in order to produce accurate leak-transformers, the compiler computes more data; moreover, the modifications described above may imply that the generated code is different. In order to measure the compile-time overhead of the computation of leak-transformers, we compile a set of Jasmin programs with two versions of the Jasmin compiler (with and without our modifications), measure the total compilation time and compare the generated assembly code.

We have run this experiment on a machine running Ubuntu Linux on an Intel® Xeon® processor (E5-2687W v3 @ 3.10GHz) using a sample of Jasmin implementations of cryptographic primitives from various sources (examples available with the compiler, case studies in published works, private communication with Jasmin programmers). The compilation times are reported in Table 1. The compile-time overhead is about 10 %. The run-time overhead is zero (not shown in the table): the generated assembly is *identical* with the two versions of the compiler.

Note that to prove preservation of CCT, the leakage transformer needs only to exist (hence need not be computed at compile-time). If we are concerned by the running-time of the compiler, it is possible to define a modified version that does not compute the leakage transformer and straightforward to formally prove that they produce the same assembly program. Such modified compiler also preserves CCT.

Nonetheless, as described in other parts of this paper, the leakage-transformer as a product of the compilation can be a useful artifact: looking at the leakage-transformer for a particular program can tell precise information about the compilation of this program (for instance, how its run-time cost is transformed). In this case, indeed, the extra computation incurs an additional cost (but provides extra information).

### 7.3 Cost analysis

With accurate leakage-transformers at hand, a range of source-level reasoning becomes possible. In this section, we combine (as described in Section 6.2) a source-level cost analysis with the leakage transformers in order to statically compute upper-bounds of

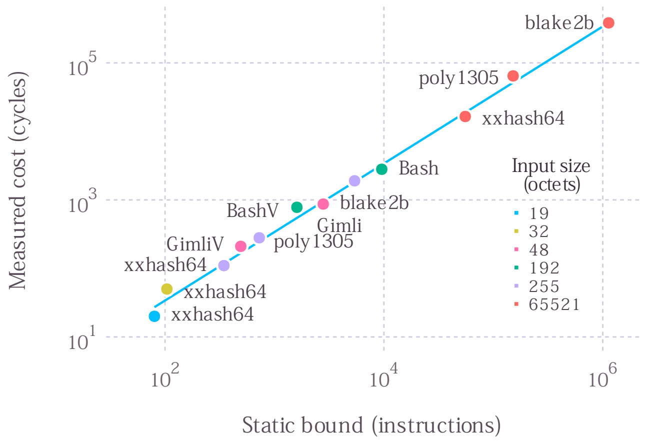

**Figure 13: Run-time cost analysis (blue line is 2.9 instr/cycle)**

run-time cost of target programs. We then compare the results with actual run-time measurements. The purpose of this experiment is to assess the precision of the leakage transformers and not to design a cycle-accurate cost analysis for x86 assembly programs. In particular, our cost model is fairly simple: we count the (total) number of executed instructions. Nonetheless, we are confident that using precise hardware models, it is possible to use the leakage transformers in a similar way to build source analyses that yield precise results about target programs.

*7.3.1 Methodology.* We have selected a sample of representative Jasmin programs (permutations, hash functions, etc.). For each program, the source-level cost analysis computes a set of linear constraints between execution counters (at the granularity of basic-blocks) and initial values of the (main) function arguments. The leakage transformers produced at compile-time yield *cost transformers* that map each target instruction to a source basic block. From this cost-transformer, we compute a symbolic upper bound of the total run-time cost: an affine combination of source execution counters.

We then fix some run-time parameters (typically, the size of the inputs) and solve the resulting integer linear program: we search for the maximal cost satisfying the constraints. This gives a static numerical estimate of the cost for the given input size.

We also run each compiled program on inputs of the corresponding sizes and measure[8] the number of executed instructions and elapsed CPU cycles. Elapsed time is estimated by a Rust program that calls the Jasmin functions; it is built on top of Criterion.rs, a "statistics-driven micro-benchmarking tool".

*7.3.2 Results.* Experimental results are shown in Figure 13. Each point of the graph corresponds to one program and one choice of input size. The programs Gimli and Bash are permutations: their inputs have fixed sizes. Both come in two versions: a reference and one optimized for platforms with AVX2 vectorized instructions (the capital V at the end of the names mark the vectorized versions).

[8]by reading Linux performance counters on a laptop running Linux 5.4 on a Intel® Core™ i7-8665U CPU @ 1.90GHz.

```
param int N = 10;
fn inc(reg u64 x) ⟶ reg u64 {
    inline int i;
    reg u64 r;
    r = x;
    for i = 0 to N {
        if x == i {
            r += 1; // A
        }
    }
    return r;
}
```

Exact cost: $21 + A$ instructions; computed bound: $21 + 10 \cdot A$.

**Figure 14: Precision loss in cost transformation**

The xxhash64 and poly1305 programs are a (non-cryptographic) hash algorithm and a MAC function (respectively). In both cases the control-flow structure is slightly complex as there are two different paths for short and long messages, and there are many loops to handle the input message in chunks of decreasing sizes. Finally, the program blake2b is a cryptographic hash algorithm that can produce digests of any size between 1 and 64 bytes. It is also made of several loops to first consume the message and then produce the digests of the appropriate size. In all cases, the measured number of executed instructions is *exactly* predicted by the static analysis (not shown on the graph). The measurement shows that the processor executes between 2 and 4 instructions per cycle. The plain line on the graph, obtained through linear regression of the measurements, has a slope of 2.9 instructions per cycle (with a correlation coefficient of 0.999).

*7.3.3 Remark on precision loss.* As mentioned in Section 6.2, the cost-transformer for loop unrolling may lose some precision, as illustrated in the (artificial) example depicted in Figure 14. When the loop is unrolled, its body is replicated, and each copy is executed as many times as the original loop. However, at most, a single copy of the nested basic block (labeled A) is executed, but the compiler cannot predict which one, hence the loss of precision, assuming that each copy *may* be executed.

Such pathological cases do not occur in practice as conditions that are nested in unrolled loops and involve the loop counters are usually resolved at compile-time.

## 8 RELATED WORK

*Compilation and Cryptographic Constant-Time.* To our best knowledge, preservation of CCT is first considered in [7], and proved formally (in Coq) for a toy compiler inspired from Jasmin. The proof is based on CT-simulations, an adaptation for CCT of the classic simulation technique. Informally, a CT-simulation establishes that equality of leakage in two source executions entails equality of leakage in the corresponding target executions. Although very general, CT-simulations require reasoning about four executions and are more difficult to establish than the classic simulations. By introducing structured leakage and leakage transformers, our work is the first to forego completely the use of CT-simulations.

More recently, [6] consider a direct method based on proving that leakage of target programs is identical (up to erasure) to leakage of the corresponding source programs. They use their method for proving preservation of CCT for a patched version of CompCert. We briefly discuss some of the main differences with our work: i) [6] does not handle leakage in expressions: the first verified pass of [6] is C#minorgen. In particular, SimplLocals is not proved CCT-preserving. In contrast, our techniques are able to reason about leakage in expressions; ii) the method of [6] applies to transformations that preserve or erase leakage. This excludes Linearize, for which a CT-simulation is used. In contrast, leakage transformers do not impose restrictions on how transformations modify leakage, and our proof of Linearize is straightforward; iii) instantiating the method of [6] requires syntactic restrictions on programs (e.g. the RTLgen pass is only proved correct for programs without switch statements). We believe that our method would be able to lift these restrictions; iv) our notion of instrumented correctness is new and simplifies the proofs: in [6], each pass must be independently proved to be both correct and CCT-preserving. Instead, each pass is just independently proved to be correct in our work, and CCT-preserving proof is done only once.

The FACT compiler [10] transforms an information flow secure program into cryptographic constant-time (CCT) programs that are protected against cache-based timing side-channels. Motivated by Spectre and other recent micro-architectural attacks, recent works explore compiler-based mitigations under speculative execution. Guarnieri and Patrignani [16] shows (in)security of several common compiler-based mitigation techniques, including fence insertion and speculative load hardening, against these attacks. Their analysis is based on speculative variants of CCT. Vassena et al. [22] designs and implements a provably sound automated compiler-based method for mitigating the BCB (bound check bypass) variant of Spectre attacks. The correctness of their approach is not machine-checked.

*Secure compilation.* Abate et al. [1] provides a systematic classification and comparison of the different notions of secure compilation. This work is primarily foundational; it does not target any specific compiler and does not address the problem of deploying secure compilers. To address the latter problem, Namjoshi and Tabajara [17] develop a translation validation framework for hyperproperties, and illustrate its application to several common optimizations.

The interaction between information-flow and compilation has been studied extensively. One line of work considers information flow types preserving compilers, see e.g. [8, 11]. In short, these works define information-flow type systems for source and target programs and show that typable source programs are transformed into typable target programs. Sison and Murray [21] follow a different approach: they define an information flow type system for source programs and develop secure refinement methods to prove that typable source programs are compiled into programs that satisfy (timing-sensitive) non-interference. Their proof is mechanized using the Isabelle proof assistant.

*Cost.* There is a vast body of work on automatically analyzing program efficiency. In particular, the fields of WCET (Worst-Case Execution Time) and cost analyses aim to provide estimates (upper and lower bounds) of program execution. These estimates use a broad range of cost models. One of the simplest models is the instruction counting model considered in this paper. However, many works in the WCET community also consider very precise cost models that account for underlying micro-architectural features. Analyses are either carried out on source programs (prevailing for cost analysis) and low-level programs (prevailing for WCET), but only a few works connect the costs of source and target programs. One of the first works in this direction is [13], which develops a time bounds-certifying compiler from a safe dialect of C to assembly. However, their work focuses on upper rather than exact bounds for assembly programs and follows the principles of certifying compilation. In contrast, our work is more focused on transferring the results of source-level cost analysis. In this sense, our work is more closely related to the CerCo compiler [4], which connects a cost analysis for source programs with the cost of target programs. Their work goes beyond the goals of our present study, as their compiler provides, via annotations in the source program, realistic estimates of the time and space cost of basic blocks of the target programs. A similar approach is taken by Carbonneaux et al. [9] to provide upper bounds on stack usage of assembly programs generated by the CompCert verified compiler. In a functional setting, Paraskevopoulou and Appel [18] prove preservation of stack space by closure conversion, whereas Gómez-Londoño et al. [15] prove a similar result for the CakeML compiler. To the exception of [13], all these works support mechanized correctness proofs using proof assistants.

## 9 CONCLUSION

We have introduced new tools: structured leakage and leakage transformers for reasoning about the compilation of non-functional, quantitative, properties such as cryptographic constant-time and cost. Using our tools, we have provided the first mechanized proof that the Jasmin compiler preserves CCT, and a provably correct method for computing the cost of assembly programs. In the future, we would like to consider other observational non-interference properties and richer execution models that accommodate speculative and out-of-order execution.

## ACKNOWLEDGMENTS

We thank our shepherd Chris Hawblitzel and the anonymous reviewers for their useful suggestions. This work is partially supported by ONR Grant N00014-19-1-2292 and Project-ANR-18-CE25-0014.

## REFERENCES

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2018. Exploring Robust Property Preservation for Secure Compilation. In *Computer Security Foundations 2019.* http://arxiv.org/abs/1807.04603

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy (S&P)*. 965–982. https://doi.org/10.1109/SP40000.2020.00028

[4] Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2013. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers (Lecture Notes*

*in Computer Science, Vol. 8552)*, Ugo Dal Lago and Ricardo Peña (Eds.). Springer, 1–18. https://doi.org/10.1007/978-3-319-12466-7_1

[5] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[6] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal Verification of a Constant-Time Preserving C Compiler. *Proceedings of the ACM on Programming Languages* (*POPL*) (2020).

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

[8] Gilles Barthe, Tamara Rezk, and David A. Naumann. 2006. Deriving an Information Flow Checker and Certifying Compiler for Java. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 230–242. https://doi.org/10.1109/SP.2006.13

[9] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 270–281. https://doi.org/10.1145/2594291.2594301

[10] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 174–189. https://doi.org/10.1145/3314221.3314605

[11] Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving compilation of end-to-end verification of security enforcement. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 412–423. https://doi.org/10.1145/1806596.1806643

[12] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. IEEE Computer Society, 51–65. https://doi.org/10.1109/CSF.2008.7

[13] Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, Mark N. Wegman and Thomas W. Reps (Eds.). ACM, 184–198. https://doi.org/10.1145/325694.325716

[14] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *Security and Privacy Workshops (SPW), 2015 IEEE*. 73–87. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7163211

[15] Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. 2020. Do you have space for dessert? a verified space cost semantics for CakeML programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 204:1–204:29. https://doi.org/10.1145/3428272

[16] Marco Guarnieri and Marco Patrignani. 2019. Exorcising Spectres with Secure Compilers. *CoRR* abs/1910.08607 (2019). arXiv:1910.08607 http://arxiv.org/abs/1910.08607

[17] Kedar S. Namjoshi and Lucas M. Tabajara. 2020. Witnessing Secure Compilation. In *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 11990)*, Dirk Beyer and Damien Zufferey (Eds.). Springer, 1–22. https://doi.org/10.1007/978-3-030-39322-9_1

[18] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP (2019), 83:1–83:29. https://doi.org/10.1145/3341687

[19] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, Fabienne Carrier, Mihail Asavoae, and Rémy Boutonnet. 2019. Improving WCET Evaluation using Linear Relation Analysis. *Leibniz Trans. Embed. Syst.* 6, 1 (2019), 02:1–02:28. https://doi.org/10.4230/LITES-v006-i001-a002

[20] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. 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 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 1–15. https://doi.org/10.1109/EuroSP.2018.00009

[21] Robert Sison and Toby Murray. 2019. Verifying that a compiler preserves concurrent value-dependent information-flow security. In *International Conference on Interactive Theorem Proving (Lecture Notes in Computer Science)*. Springer-Verlag.

[22] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434330

```
1  Fixpoint alloc_e (m: map) (e: pexpr) : cexec (pexpr × leak_e_tr) :=
2   match e with
3    | Pconst _ | Pbool _ | Parr_init _ | Pglobal _ ⇒ ret e id
4    | Papp1 o e ⇒
5      Let: (e, r) := alloc_e m e in
6      ret (Papp1 o e) r
7
8    | Pload ws x e ⇒ …
9      Let: (e, r) := alloc_e m e in
10     ret (Pload ws x e) [ r, id ]
11
12   | Pvar x ⇒
13     if … (* x has size ws and is allocated at offset ofs *) then
14       ret (Pload ws stk ofs) [ id; C (sp + cst ofs) ]
15     else … (* x is a register *)
16       ret e id
17
18   | Pget ws x e ⇒
19     Let: (e, r) := alloc_e m e in
20     if … (* x is allocated at offset ofs *) then
21       let: (ofs', t) := mk_ofs ws e ofs in
22       ret (Pload ws stk ofs') [ r ∘ t, I(x ↦ sp + cst x × cst ws + cst ofs) ]
23     else … ret (Pget ws x e) [ r, id ]
24
25   | … ⇒ …
26  end.
```

**Figure 15: Pseudo-code of the stack-allocation of expressions**

## A DETAILS ON STACK-ALLOCATION

This appendix completes Section 5.5. It presents in graphical form the leakage transformer produced by the stack-allocation pass applied to the example program, on Figure 16. It also briefly describes the (Coq) implementation of the stack-allocation pass.

The stack-allocation pass is split into two phases: an analysis that computes the layout of the stack frame; and a transformation that introduces the memory operations. As far as leakage transformers are concerned, only the second phase is relevant. The transformation phase is structured in three main parts, that transform expressions, left values, and instructions, respectively. The transformations of expressions and of left-values are very similar: they replace reads from (resp. writes to) variables and arrays with loads from (resp. stores to) the memory. The final transformation of instructions recursively applies these sub-transformations and composes their results in a straightforward way. For the sake of brevity, this exposition is thus restricted to the transformation of expressions, implemented by the function alloc_e whose pseudo-code is partially shown on Figure 15.

The compilation of an expression $e$ depends on a "map" $m$ that is the result of the analysis phase. It returns an expression and the corresponding leakage transformer, or fails; hence the monadic return type *cexec* and the ret notation for building successful results. The alloc_e function is recursively defined, following the structure of the expression to compile. Constant expressions (line 3) are returned as is; the leakage transformer is therefore the identity id. Unary operators (Papp1) applied to a sub-expression produce the same leakage as said sub-expression; therefore the leakage transformer $r$ corresponding to the compilation of the sub-expression (line 5) directly applies to the whole expression (line 6). The leakage for a memory load is a pair: the first component stands for the evaluation of the address and the second component reveals the address that is accessed. The compilation of a load expression recursively transforms the address expression but preserves its value

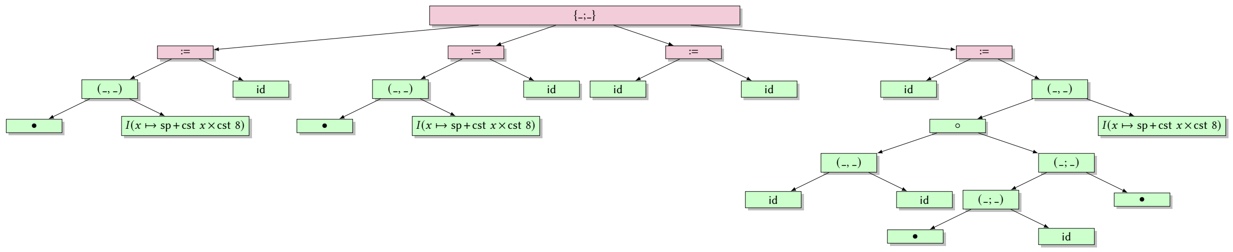
{_;_}
:=
(_, _)
id
•
$I(x \mapsto \text{sp} + \text{cst } x \times \text{cst } 8)$
∘
(_; _)

**Figure 16: Leakage transformer**

(line 10). There are two cases in the compilation of a read from a variable $x$ (line 12): if this variable is bound in the map $m$ to some offset *ofs*, then it must be transformed into a memory load and a leakage corresponding to this load is introduced; otherwise, the expression and its leakage are preserved. Finally, the compilation of a read of size $ws$ from array $x$ at index $e$ first recursively transforms the sub-expression $e$; then if the array is laid out at offset *ofs*, the relevant element is at offset $e \times ws + ofs$. The expression representing this offset is built by the smart constructor mk_ofs (not shown on the figure) which simplifies constant expressions and returns the corresponding leakage transformer. The complete leakage transformer (line 22) is the parallel composition of, on one hand, the composition of both transformations of the offset, and on the other hand, the transformation of the array leakage into a memory leakage.