Type-directed Program Transformation for Constant-Time Enforcement

Frédéric Besson, Thomas Jensen, Gautier Raimondi

Type-directed Program Transformation for Constant-Time Enforcement

Frédéric Besson
Inria, Univ Rennes
Rennes, France

Thomas Jensen
Inria, Univ Rennes
Rennes, France
University of Copenhagen
Copenhagen, Denmark

Gautier Raimondi
Inria, Univ Rennes
Rennes, France

ABSTRACT

Constant-time is a programming discipline which protects security sensitive code against a wide class of timing attacks. This discipline can be formalised as a non-interference property and enforced by an information flow type system which prevents branching and memory accesses over secret data. We propose a relaxed information flow type system which tracks indirect flows but only rejects programs leaking secrets through direct flows. The main result of this paper is that any program that is accepted using this relaxed type system can be transformed automatically into a semantically equivalent constant-time program. Our algorithms are implemented in the JASMIN compiler and validated against synthetic programs.

ACM Reference Format:
Frédéric Besson, Thomas Jensen, and Gautier Raimondi. 2023. Type-directed Program Transformation for Constant-Time Enforcement. In *International Symposium on Principles and Practice of Declarative Programming (PPDP 2023), October 22–23, 2023, Lisboa, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3610612.3610618

1 INTRODUCTION

Cryptographic code is notoriously hard to implement because it must be correct, fast and secure. Cryptographic code is costly in terms of computation time but nevertheless necessary for protecting communication, so efficiency is essential. Therefore, implementations use sophisticated algorithms and exploit particular features of the hardware's mathematical functionalities [27] in order to optimise execution time. In addition, the implementation also needs to be secure with respect to side-channel attacks. In this work, we are concerned with timing attacks where attackers attempt to extract confidential information, *e.g.*, cryptographic keys, by observing the execution time [20]. *Constant-time programming* [6] is the *de facto* standard to protect implementations against a wide range of timing attacks that exploit micro-architecture side-channels *e.g.*, cache attacks. The constant-time programming discipline imposes two constraints on the code: it is forbidden to make control flow decisions [24] or access memory addresses [22] that depend on secret data.

PPDP 2023, October 22–23, 2023, Lisboa, Portugal

ACM ISBN 979-8-4007-0812-1/23/10…$15.00
https://doi.org/10.1145/3610612.3610618

Several formal approaches have been proposed to ensure that the constant-time programming property holds at the binary level. Barthe *et al.* [7, 8] develop a formal model and a verifier for constant-time programming at the assembly level. Other approaches ensure the constant-time programming at source level [13] and transfer the property at the binary level using a verified compiler [10]. To ease the burden of developing constant-time code, Cauligi *et al.* [16] propose FaCT, a Domain Specific Language (DSL), to automatically generate constant-time code. FaCT is based on an information flow type system which ensures that a program accepted by the type system can be automatically transformed into a constant-time program. We follow the same methodology but propose a more permissive type system which allows more programs to be transformed into a constant-time equivalent. The main observation of the paper is that *distinguishing between direct and indirect information flows enables a number of program transformations for constant-time that make it possible to transform programs which previous methods would have rejected.* More precisely, our contributions can be phrased as follows: i) we define an information flow tracking type system which distinguishes indirect and direct flows and only forbids leakage due to direct flows; ii) we present type directed program transformations which transform a well-typed program into a constant-time program. iii) we also present experiments over small but challenging synthetic programs.

The rest of the paper is organised as follows. In Section 2, we present our core language and the constant-time type system. We also present the main features of the FaCT DSL [16]. In Section 3, we present informally our program transformations and how they lift certain limitations of FaCT. In Section 4, we define our information flow tracking type system and show how it relates to the usual information flow type system for constant-time. In Section 5, we describe our program transformations and prove their security. We report on experiments using the Jasmin compiler [5] in Section 6. Related work is presented in Section 7 and Section 8 concludes. The paper contains outlines of proofs of the main theorems. For detailed proofs, see Raimondi's forthcoming thesis [26].

2 BACKGROUND

2.1 Syntax

We consider an imperative language with arrays and a **for** loop. The syntax is given below:

$$\begin{array}{lcl} \mathit{expr} \ni e & ::= & x \mid c \mid e_1 \oplus e_2 \mid e_1 ? e_2 : e_3 \mid t[e] \\ \mathit{stmt} \ni s & ::= & \mathbf{skip} \mid x = e \mid t[e_1] = e_2 \mid s_1; s_2 \\ & & \mid \mathbf{if}^{@p}\ x\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{next}\ s_3 \\ & & \mid \mathbf{for}\ x\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s \end{array}$$

An expression e may be a constant c, a variable x, a binary operator applied to arguments $e_1 \oplus e_2$, a conditional expression $e_1 ? e_2 : e_3$ or an array access $t[e]$ where t is an array variable and e a computed index. The size of arrays is constant and statically known.

A statement s may be a **skip**, an assignment $x = e$, an array update $t[e] = e$, a conditional or a **for** loop. Compared to the usual **if** e **then** s_1 **else** s_2, our conditional takes an additional **next** statement s_3. Semantically, $\mathbf{if}^{@p}$ e **then** s_1 **else** s_2 **next** s_3, behaves as (**if** e **then** s_1 **else** s_2); s_3. The **next** s_3 syntax is introduced to ease the presentation of our program transformation (see Section 3) and make explicit that the statement s_3 is within the scope of the condition. We also write $\mathbf{if}^{@p}$ e **then** s_1 **else** s_2 for $\mathbf{if}^{@p}$ e **then** s_1 **else** s_2 **next skip**. For the sake of simplifying the presentation of our code transformation, we impose the following syntactic constraints: i) the condition e is restricted to be a variable, say x, which is neither modified within the conditional nor after the conditional; ii) the conditional statement is uniquely identified by an annotation $@p$ where $p \in \mathbb{L}$ is a program point or location; iii) the loop bounds are known constants and; iv) the loop index is not modified within the loop body. The restrictions to constant array size and loops bounds are common for cryptographic code and can be found *e.g.*, in the input language of the Jasmin compiler [5], specifically designed for cryptography.

2.2 A Semantic Definition of Constant-time

The language is given a big-step semantics which is standard except that the execution also generates a trace of leakage. A derivation is of the form

$$(P, \sigma) \downarrow^{t} \sigma'$$

where P is a program, σ is an environment mapping variables x and arrays a to their values, t is a leakage trace and σ' is the final environment. Following Barthe *et al.* [12], a leakage event is generated when evaluating a conditional or performing an access to an array. In the evaluation of expressions, the index of an array read is leaked in the trace t[1]. For simplicity, we assume that binary operations have an execution time independent of the arguments and, therefore, no leakage is produced when evaluating $\oplus$. The conditional expression is strict and evaluates all its arguments without leaking the value of the condition e_1. At runtime, such conditionals can be implemented without branching instructions using either bitwise operations *e.g.*, $e_1 ? e_2 : e_3 \equiv e_1 \& e_2 + !e_1 \& e_3$ or by using hardware support *e.g.*, conditional moves available on various architectures. The evaluation of statements is standard with the exception that, similarly to an array read, the index of an array write is leaked. Moreover, the control-flow decisions *e.g.*, which branch of a condition is taken, are also leaked in the trace.

A program P abides to the constant-time programming discipline if starting from environments which agree on the variables containing public values, the execution traces of leakage events are indistinguishable for an attacker. Given the leakage semantics, the constant-time property (see Definition 1) can be formalised as a non-interference property with respect to a low-equivalence relation over environments. In the following definition, the set L is to be thought of as the set of public ("low") variables.

[1]The array variable is not explicitly leaked because it can be reconstructed as control-flow decisions are also leaked.

$$\frac{}{(c, \sigma) \downarrow^{\epsilon} c} \qquad \frac{}{(x, \sigma) \downarrow^{\epsilon} \sigma(x)}$$

$$\frac{(e_i, \sigma) \downarrow^{t_i} v_i \quad i \in \{1,2\} \qquad v_3 = v_1 \oplus v_2}{(e_1 \oplus e_2, \sigma) \downarrow^{t_1 \cdot t_2} v_3} \qquad \frac{(e_i, \sigma) \downarrow^{t_i} v_i \quad i \in \{1,2,3\} \qquad v = \mathit{if}\ v_1\ \mathit{then}\ v_2\ \mathit{else}\ v_3}{(e_1 ? e_2 : e_3, \sigma) \downarrow^{t_1 \cdot t_2 \cdot t_3} v}$$

$$\frac{(e, \sigma) \downarrow^{t_e} i \quad 0 \le i < \mathit{size}(t) \quad \sigma(t)[i] = v}{(t[e], \sigma) \downarrow^{t_e \cdot i} v}$$

$$\frac{}{(\mathbf{skip}, \sigma) \downarrow^{\epsilon} \sigma} \qquad \frac{(e, \sigma) \downarrow^{t} v}{(x = e, \sigma) \downarrow^{t} \sigma[x \mapsto v]}$$

$$\frac{(s_1, \sigma_1) \downarrow^{t_1} \sigma_2 \quad (s_2, \sigma_2) \downarrow^{t_2} \sigma_3}{(s_1; s_2, \sigma_1) \downarrow^{t_1 \cdot t_2} \sigma_3}$$

$$\frac{(e_1, \sigma) \downarrow^{t_1} i \quad 0 \le i < \mathit{size}(t) \quad (e_2, \sigma) \downarrow^{t_2} v}{(t[e_1] = e_2, \sigma) \downarrow^{t_1 \cdot t_2 \cdot i} \sigma[t \mapsto (\sigma(t)[i \mapsto v])]}$$

$$\frac{\sigma(x) = b \quad (s_b, \sigma) \downarrow^{t} \sigma' \quad (s, \sigma') \downarrow^{t'} \sigma''}{(\mathbf{if}^{@p}\ x\ \mathbf{then}\ s_{true}\ \mathbf{else}\ s_{false}\ \mathbf{next}\ s, \sigma) \downarrow^{b \cdot t \cdot t'} \sigma''}$$

$$\frac{\forall_{i \in [c_1; c_2]} (x = i; s, \sigma_i) \downarrow^{t_i} \sigma_{i+1} \quad t = t_{c_1} \cdot \cdots \cdot t_{c_2}}{(\mathbf{for}\ x\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s, \sigma_{c_1}) \downarrow^{t} \sigma_{c_2+1}}$$

Figure 1: Leaky semantics

Definition 1 (Constant-time). *Let L be a set of variables and P be a program. The program P abides to the constant-time programming discipline for L, written $CT(P, L)$, if the following non-interference property holds:*

$$CT(P, L) \triangleq \bigwedge \begin{pmatrix} \sigma_1 \equiv_L \sigma_2 \\ (P, \sigma_1) \downarrow^{t_1} \sigma_1' \\ (P, \sigma_2) \downarrow^{t_2} \sigma_2' \end{pmatrix} \Rightarrow t_1 = t_2$$

where $\sigma \equiv_L \sigma' \triangleq \forall x \in L, \sigma(x) = \sigma'(x)$.

Note that programs respecting the constant-time programming discipline do not have the same timing behaviour for any input. The guarantee is that programs run with 2 distinct secrets execute the exact same sequence of instructions and perform the exact same memory accesses in the same order. In practice, this is an effective countermeasure protecting against micro-architectural timing leaks due to branch prediction and cache memory.

2.3 Contant-time Type System

The constant-time programming discipline can be enforced by a flow-sensitive information flow control type system in the style of Hunt and Sands [18]. A typing judgement is of the form $\Delta \vdash_{\mathrm{CT}} \Gamma\{p\}\Gamma'$. The typing environments $\Gamma, \Gamma', \Delta : \mathit{Var} \to \{\mathbf{H}, \mathbf{L}\}$ map a program variable x to its type $\tau \in \{\mathbf{H}, \mathbf{L}\}$. Γ and Γ' assign types to scalar variables while Δ assigns types to array variables. As the type system is flow-sensitive for scalar variables, Γ is the typing environment before running P and Γ' is the typing environment obtained after running P. The typing environments for arrays Δ is not flow-sensitive. The rationale is that, unlike a variable assignment, an array update would be modelled as a *weak update* and therefore it is very unlikely that flow-sensitivity would increase precision. The typing judgement for expressions is of the form $\Delta, \Gamma \vdash e : \tau$. Compared to the usual Volpano-Smith style type systems [18, 31], the type system is flow-sensitive and enforces the additional typing constraints that conditions and array indices must be of type **L**. Therefore, we obtain the typing rules of Figure 2. Theorem 1

$$\frac{}{\Delta, \Gamma \vdash^{ct} x : \Gamma(x)} \qquad \frac{}{\Delta, \Gamma \vdash^{ct} i : \mathbf{L}} \qquad \frac{\Delta, \Gamma \vdash^{ct} e : \mathbf{L}}{\Delta, \Gamma \vdash^{ct} t[e] : \Delta(t)}$$

$$\frac{\Delta, \Gamma \vdash^{ct} e_i : \tau_i \quad i \in \{1,2\}}{\Delta, \Gamma \vdash^{ct} e_1 \oplus e_2 : \bigsqcup_i \tau_i} \qquad \frac{\Delta, \Gamma \vdash^{ct} e_i : \tau_i \quad i \in \{1,2,3\}}{\Delta, \Gamma \vdash^{ct} e_1 ? e_2 : e_3 : \bigsqcup_i \tau_i}$$

$$\frac{}{\Delta \vdash^{ct} \Gamma\{\mathbf{skip}\}\Gamma} \qquad \frac{\Delta \vdash^{ct} \Gamma_1\{s_1\}\Gamma_2 \quad \Delta \vdash^{ct} \Gamma_2\{s_2\}\Gamma_3}{\Delta \vdash^{ct} \Gamma_1\{s_1; s_2\}\Gamma_3}$$

$$\frac{\Delta, \Gamma \vdash^{ct} e : \tau}{\Delta \vdash^{ct} \Gamma\{x = e\}\Gamma[x \mapsto \tau]} \qquad \frac{\Delta, \Gamma \vdash^{ct} e_1 : \mathbf{L} \quad \Delta, \Gamma \vdash^{ct} e_2 : \tau_2 \quad \tau_2 \sqsubseteq \Delta(t)}{\Delta \vdash^{ct} \Gamma\{t[e_1] = e_2\}\Gamma}$$

$$\frac{\Delta, \Gamma \vdash^{ct} c : \mathbf{L} \quad \Delta \vdash^{ct} \Gamma\{s_1\}\Gamma_1 \quad \Delta \vdash^{ct} \Gamma\{s_2\}\Gamma_2}{\Delta \vdash^{ct} \Gamma\{\mathbf{if}^{@p}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\}\Gamma_1 \sqcup \Gamma_2}$$

$$\frac{\Gamma \sqsubseteq \Gamma' \quad \Gamma_1 \sqsubseteq \Gamma' \quad \Delta \vdash^{ct} \Gamma'[i \mapsto \mathbf{L}]\{s\}\Gamma_1}{\Delta \vdash^{ct} \Gamma\{\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s\}\Gamma'}$$

Figure 2: Constant Time Type System

states that the type-system of Figure 2 ensures that the program is constant-time according to Definition 1.

Theorem 1 (Soundness of Constant Time Type System). *Consider a program P typable according to the type-system of Figure 2*

$$\Delta \vdash^{ct} \Gamma\{P\}\Gamma'$$

We have that $CT(P, L)$ for $L = \{x \mid \Gamma(x) = \mathbf{L} \vee \Delta(x) = \mathbf{L}\}$.

2.4 Program Transformations of FaCT

FaCT [16] is a DSL for writing constant-time code. FaCT defines an information flow type system and guarantees that any well-typed program can be transformed into a functionally equivalent but constant-time program. As we follow a similar methodology, we precisely describe the main features and algorithms of FaCT adapted to our language.

Type System. The type system of FaCT is based on a classic Volpano-Smith information flow control type system [31]. The main difference with respect to a type system for constant time is that the type system allows conditionals to branch on a secret value. The rationale is that the FaCT program transformations are able to eliminate such potential leaks.

Predicated Code. In order to remove secret control dependencies, FaCT performs a so-called *if-conversion* [2] and generates branchless, predicated code.

Example 1. *Consider the following code which, depending on a secret h, sets the variable x to either l_1 or l_2.*

$$\boldsymbol{if}\ h\ \boldsymbol{then}\ x = l_1\ \boldsymbol{else}\ x = l_2$$

After if-conversion*, we get the following branchless code which eliminates the leakage due to the conditional.*

$$x = h?l_1 : x; x = !h?l_2 : x$$

In terms of information flow, *if-conversion* has the effect of turning an indirect flow into a direct flow, which respects the constant-time discipline.

Public Safety. An issue with *if-conversion* is that it is not always a semantics-preserving transformation. The problem arises when the safety of memory accesses within the **then** (resp. the **else**) branch relies on whether the condition holds or not. In Example 1, suppose that the expressions l_i performs a memory access *e.g.*, $t[i]$ and that the condition h guards against out-of-bound accesses *i.e.*, $h \triangleq i < size(t)$. After *if-conversion* the target code performs the memory access unconditionally and may perform an illegal access. To solve this issue, the FaCT type system generates verification conditions to ensure that the memory accesses are still valid after transformation *i.e.*, that the expressions l_1 and l_2 in a predicated assignment $x = h?l_1 : l_2$ are safe to evaluate, independently of the value of h.

3 OVERVIEW

In this section, we show through examples how a more fine-grained and relaxed type system enables more sophisticated program transformations, thereby increasing the set of programs that can be automatically transformed into constant-time programs. In particular, the code snippets we present are all rejected by the FaCT type system and its implementation.

Limitation of Classic if-conversion. As explained above, *if-conversion* and predicated code turn an indirect flow into a direct flow by removing the leakage due to tests on **H** values. However, *if-conversion* is not sufficient to remove indirect leakage due to assignments inside the conditional.

Example 2. *Consider the following program $P0$.*

$$P0 : \boldsymbol{if}^{@p}\ h\ \boldsymbol{then}\ (x = l_1; y = t[x])\ \boldsymbol{else}\ (x = l_2; y = t[x])$$

The expressions l_1 and l_2 have type **L** *but since the assignment is performed in a* **H** *context, the variable x is given type* **H**. *Therefore, the ensuing array access, $t[x]$, is rejected by the* FaCT *type system.*

The *if-conversion* performed by FaCT would generate the following insecure code.

$$x = h?l_1 : x; y = h?t[x] : y; x = !h?l_2 : x; y = !h?t[x] : y$$

It is insecure because there is a direct flow from the secret h to the variable x. It follows that the array accesses are secret dependent thus violating the constant-time discipline. Therefore, FaCT rightly rejects this code.

Naive Constant-time Rewriting. To avoid leaking an array access $y = t[x]$, a standard but inefficient countermeasure consists in iterating over all the indices i of the array and select the relevant value using a conditional expression.

$$\textbf{for } i \textbf{ from } 0 \textbf{ to } \mathit{size}(t){-}1 \textbf{ do } y = (x == i)?t[i] : y$$

We fall back on this transformation for direct **H** flows.

Delayed if-conversion. For information leakage due to indirect flows, our transformation is more efficient than the naive constant-time rewriting. It is based on the observation that the code can be made constant-time by postponing the if-conversion at the cost of introducing extra variables. We perform a *delayed if-conversion* so that the direct **H** flow with h is generated after the array access. Concretely, our type system accepts $P0$ and generates the following secure code.

$$\begin{array}{l} x_t = l_1; y_t = t[x_t]; x_e = l_2; y_e = t[x_e]; \\ x = h?x_t : x_e; y = h?y_t : y_e. \end{array}$$

To enable this transformation, our type system (see Figure 4) tracks the information that the array index x *is secret due to an indirect flow*.

The previous approach works when we only assign to scalar variables but is not realistic for assignments to arrays, as this would require having a distinct copy of the array for each branch of the conditional which would be too costly. Instead, array writes are predicated. For the following program $P1$, we get the program $P1'$.

$$P1 : \textbf{if}^{@p}\ h \textbf{ then } (x = l_1; t[x] = 0) \textbf{ else } (x = l_2; t[x] = 1)$$

$$P1' : \begin{array}{l} x_t = l_1; t[x_t] = h?0{:}t[x_t] \\ x_e = l_2; t[x_e] = !h?1{:}t[x_e] \\ x = h?x_t : x_e \end{array}$$

As x_t and x_e both contain **L** values, there is no leakage. Yet, being predicated by h, the content of the array t depends on h. As a consequence, our typing rule for array updates is stricter than for scalar variables.

Out-of-scope Indirect Flows. In the previous case, the leaky memory access is within the scope of the condition h and therefore a delayed *if-conversion* is sufficient to remove the leaky access. This is not enough for the following program $P2$, where the leaky memory access $t[x]$ occurs after the condition h. To complicate matters, there is also a harmless array update of the **L** array t before the problematic memory access $t[x]$.

$$P2 : (\textbf{if}^{@p}\ h \textbf{ then } x = l_1 \textbf{ else } x = l_2); (t[l_3] = l_4; y = t[x])$$

If *if-conversion* is applied to the body of the conditional only, the variable x has a direct flow w.r.t h and the array access $t[x]$, that is outside the scope of the conditional, still leaks information about the secret h. A naive solution would be to perform code motion and duplicate the offending code in both branches of the conditional. However, in our example, this has the adverse effect of moving the harmless statement $t[l_3] = l_4$ into a **H** context and, from the type-system standpoint, introducing a novel information flow violation. Our solution is to move the offending code using the **next** statement of the conditional. We obtain:

$$(\textbf{if}^{@p}\ h \textbf{ then } x = l_1 \textbf{ else } x = l_2 \textbf{ next } t[l_3] = l_4; y = t[x]);$$

Intuitively, the statements in the **next** are within the syntactic scope of the conditional but are run outside the **H** context of the condition. The statement in the **next** statement requires a specific transformation performed by our enhanced *if-conversion*. For the statements in the scope of the **next**, the statements with an indirect dependency on the conditional are duplicated on-the-fly. A subtlety is that **L** statements, here $t[l_3] = l_4$ are kept unchanged. For our example $P2$, we obtain the code in Figure 3. The resulting code is constant-time.

$$\begin{array}{l} x_t = l_1;\ x_e = l_2; t[l_3] = l_4; \\ y_t = t[x_t];\ y_e = t[x_e]; \\ x = h?x_t{:}x_e;\ y = h?y_t{:}y_e; \end{array}$$

Figure 3: Result of transforming program $P2$.

Preserving Safety. We assume that the source program is safe which here means that it does not make any out-of-bounds array accesses. In a security context, this prerequisite is natural as an unsafe program cannot be deemed secure. With this hypothesis, we instrument the source program with dynamic bound checks, preventing *if-conversion* from generating unsafe target programs. Our instrumentation transforms the array access $t[x]$ into $t[0 \le x < \mathit{size}(t)?x : 0]$. Because the program is safe, we have the invariant that $0 \le x < \mathit{size}(t)$, so the conditional expression always evaluates to x. Hence, the semantics of the program remains unchanged and the program remains safe after *if-conversion*. In addition, the transformation does not introduce a security leak because the length of an array is a constant and therefore a **L** value. This instrumentation has a performance penalty but optimising compilers should be able to remove most of the redundant checks.

4 INDIRECT FLOW TRACKING TYPE SYSTEM

In this section, we present our flow-sensitive information flow type system which distinguishes between direct and indirect flows. The type system accepts leakage due to indirect flows, formalising the intuition that indirect flows are benign when it comes to constant-time transformations. The main result for this indirect flow tracking type system is that our constant-time transformation succeeds for any source program that is well-typed.

We extend the usual two-point $\{\mathbf{H}, \mathbf{L}\}$ lattice with information flow types of form $\mathbf{I}(l)$ where $l \subseteq \mathbb{L}$ is a subset of program locations. The type $\mathbf{I}(l)$ is given to variables that are secret because of an indirect secret flow arising from a conditional labelled with one of the labels in l. We keep the type **H** which now means "secret because of a direct or indirect flow".

$$\mathit{IFType} = \mathcal{P}(\mathbb{L}) \cup \{\mathbf{H}\}.$$

In this lattice, **H** is the greatest element and $\mathbf{I}(l_1) \sqsubseteq \mathbf{I}(l_2)$ if $l_1 \subseteq l_2$ because it is safe to over-approximate the set of conditionals that caused an indirect flow. The element $\mathbf{I}(\varnothing)$ intuitively means "does not depend on secrets" and types values with only public information. We will use **L** as an abbreviation for the type $\mathbf{I}(\emptyset)$.

The flow-sensitive type system keeps track of security levels *and* the program labels of secret conditionals encountered so far in the

Expressions :

$$\frac{}{\Delta, \Gamma \vdash x : \Gamma(x), \emptyset} \qquad \frac{}{\Delta, \Gamma \vdash i : \mathbf{L}, \emptyset} \qquad \frac{\Delta, \Gamma \vdash e : \tau, l \quad \tau = \mathbf{I}(l')}{\Delta, \Gamma \vdash t[e] : \Delta(t) \sqcup \tau, l \cup l'} \qquad \frac{\Delta, \Gamma \vdash e_i : \tau_i, l_i \quad i \in \{1,2\}}{\Delta, \Gamma \vdash e_1 \oplus e_2 : \bigsqcup_i \tau_i, \bigcup_i l_i} \qquad \frac{\Delta, \Gamma \vdash e_i : \tau_i, l_i \quad i \in \{1,2,3\}}{\Delta, \Gamma \vdash e_1?e_2{:}e_3 : \bigsqcup_i \tau_i, \bigcup_i l_i}$$

Statements :

$$\frac{}{\Delta, \kappa \vdash \Gamma\{\mathbf{skip}^{\emptyset}_{\emptyset}\}\Gamma} \qquad \frac{\begin{array}{c}\Delta, \kappa \vdash \Gamma\{(s_1)^{r_1}_{g_1}\}\Gamma_1 \quad \Delta, \kappa \vdash \Gamma_1\{(s_2)^{r_2}_{g_2}\}\Gamma' \\ r = r_1 \cup r_2 \quad g = g_1 \cup g_2\end{array}}{\Delta, \kappa \vdash \Gamma\{(s_1; s_2)^{r}_{g}\}\Gamma'} \qquad \frac{\Delta, \Gamma \vdash e : \tau, r}{\Delta, \kappa \vdash \Gamma\{(x = e)^{r}_{\emptyset}\}\Gamma[x \mapsto \tau \sqcup \kappa]}$$

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash e_1 : \tau_1, l_1 \quad \Delta, \Gamma \vdash e_2 : \tau_2, l_2 \quad \tau_1 = \mathbf{I}(l'_1) \\ \tau_1 \sqcup \tau_2 \sqcup (\kappa \ltimes_{\mathbf{L}} \mathbf{H}) \sqsubseteq \Delta(t) \\ r = l_1 \cup l'_1 \cup l_2\end{array}}{\Delta, \kappa \vdash \Gamma\{(t[e_1] = e_2)^{r}_{\emptyset}\}\Gamma} \qquad \frac{\begin{array}{c}\Delta, \Gamma \vdash c : \tau, r_c \quad \kappa' = \kappa \sqcup (\tau \ltimes_{\mathbf{L}} \mathbf{I}(\{p\})) \\ \Delta, \kappa' \vdash \Gamma\{(s_1)^{r_1}_{g_1}\}\Gamma_1 \quad \Delta, \kappa' \vdash \Gamma\{(s_2)^{r_2}_{g_2}\}\Gamma_2 \\ \Delta, \kappa \vdash \Gamma_1 \sqcup \Gamma_2\{(s_3)^{r_3}_{g_3}\}\Gamma' \\ r = r_1 \cup r_2 \cup r_3 \cup r_c \\ g = g_1 \cup g_2 \cup g_3 \cup \tau \ltimes_{\emptyset} \{p\}\end{array}}{\Delta, \kappa \vdash \Gamma\{(\mathbf{if}^{@p}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{next}\ s_3)^{r}_{g}\}\Gamma'}$$

$$\frac{\begin{array}{c}\Gamma \sqsubseteq \Gamma' \quad \Gamma_1 \sqsubseteq \Gamma' \quad \Uparrow_{cond(s)} \Gamma' = \Gamma' \\ \Delta, \kappa \vdash \Gamma'[i \mapsto \kappa]\{(s)^{r}_{g}\}\Gamma_1\end{array}}{\Delta, \kappa \vdash \Gamma\{(\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s)^{r}_{g}\}\Gamma'}$$

$$\text{where} \quad \begin{array}{l} \tau \ltimes_d v = \begin{cases} d & \textbf{if } \tau = \mathbf{L} \\ v & \textbf{otherwise} \end{cases} \qquad (\Uparrow_l \Gamma)(x) = \begin{cases} \Gamma(x) & \textbf{if } \Gamma(x) = \mathbf{I}(l') \wedge l \cap l' = \emptyset \\ \mathbf{H} & \textbf{otherwise} \end{cases} \\[2ex] cond(s) = \{p \mid \mathbf{if}^{@p}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{next}\ s_3 \in s\} \end{array}$$

Figure 4: Flow Tracking Type System

execution. The typing judgments operates on *annotated* programs of form $(P)^r_g$ and are of the form

$$\Delta, \kappa \vdash \Gamma\{(P)^r_g\}\Gamma'$$

Here, Δ is a typing environment for array variables that are restricted to being *simple* (*i.e.*, all the array variables have a type $\tau \in \{\mathbf{L}, \mathbf{H}\}$) and *global* (*i.e.*, arrays do not change type during analysis of P). κ is the security context in which P is analysed. Γ and Γ' are the typing environments for scalar variables before and after running the program P.

The program annotations in P^r_g consists of two sets, g and r. The set $g \subseteq \mathbb{L}$ over-approximates the secret conditionals in P. The set $r \subseteq \mathbb{L}$ is an upper bound of the security levels of the indices that have been used to access an array (notice that an access with index of type $\mathbf{H}$ is ruled out by the type system). In the rest, we write $high((P)^r_g)$ for the set g *i.e.*, the set of $\mathbf{H}$ conditionals within the program P and we write $leak((P)^r_g)$ for the set r *i.e.*, the set of $\mathbf{H}$ conditionals responsible for array accesses with indirect flows.

The typing rules of Figure 4 are syntax-directed and can be turn into a type-inference algorithm able to compute both the typing environments and the program annotations. For expressions, the typing judgment is of the form

$$\Delta, \Gamma \vdash e : \tau, l$$

where τ is the security type of the result and l is the upper bound of security levels of array indices used to compute the value of e. For array accesses, an additional hypothesis enforces that the type of the index is $\mathbf{I}(l')$ *i.e.*, strictly below $\mathbf{H}$. If it is $\mathbf{L}$, there is no leakage and the expression is well typed. Otherwise, there is leakage of secrets due to indirect flows but the expression is still well-typed because the leakage will be erased by our program transformation.

For statements, the rule for **skip** and sequence are standard for a flow-sensitive type system. For an assignment $x = e$, the type for x is updated to be the least upper bound of the type τ of the expression e and the type of the security context κ. The rule for array update is flow insensitive. It checks that the type τ_1 of the index is not $\mathbf{H}$. It also checks that the type obtained from the type of the index τ_1 and the type of the written value τ_2 and an upgraded security context $\kappa \ltimes_{\mathbf{L}} \mathbf{H}$ are below the type $\Delta(t)$ of the array. Therefore, if the security context κ is $\mathbf{L}$, we have $\tau_1 \sqcup \tau_2 \sqsubseteq \Delta(t)$. Otherwise, if the array update is performed under a security context $\kappa \neq \mathbf{L}$, the typing constraints entail that $\Delta(t) = \mathbf{H}$. The r annotation is updated to reflect that an array access with an index of type τ_1 has been made.

The rule for conditions computes the security level τ of the condition c. If τ is different from $\mathbf{L}$, c contains secret information and the security context κ' is updated with $\mathbf{I}(\{p\})$, recording that execution in the branches takes place under a secret condition located at program point p. This information is also added to the annotation g that is the set of labels of the secret conditions in the statements that it annotates.

The typing rule for **for** checks that Γ' is an invariant typing environment for the loop, by checking the body s can be type checked in the slightly more constraining typing environment $\Gamma'[i \mapsto \kappa]$. In this environment, the iteration variable i gets the type of the security context κ. The rule also enforces that Γ' does not contain any dependency to conditions within s by ensuring that $\uparrow_{cond(s)} \Gamma' = \Gamma'$. This equality means that there is no variables of type $\mathbf{I}(l)$ in Γ, with l containing at least one program point p of a condition within s. This means that we keep track of indirect flows within a loop only, and do not propagate them outside of its containing loop.

Observe that if we only consider a type derivation in the empty security context ($\kappa = \mathbf{L}$) and a program P with empty annotations ($P^{\emptyset}_{\emptyset}$), our flow tracking type system enforces the constant-time property of Definition 1. Theorem 2 states this essential property that will serve as a guiding principle for our program transformations.

Theorem 2 (Constant-Time Enforcement). *If a program P is well-typed in the flow tracking type system from Figure 4 with empty annotations i.e.,*

$$\Delta, \mathbf{L} \vdash \Gamma_1\{P^{\emptyset}_{\emptyset}\}\Gamma_2$$

then P is constant-time. *More precisely, the predicate $CT(P, L)$ holds for any set of variables L satisfying*

$$\{x \mid \Gamma_1(x) \neq \mathbf{H}\} \cup \{t \mid \Delta(t) \neq \mathbf{H}\} \subseteq L.$$

Proof outline. Given a type-derivation $\Delta, \mathbf{L} \vdash \Gamma_1\{(P)^{\emptyset}_{\emptyset}\}\Gamma_2$, we can exhibit a type derivation $\Delta \vdash^{ct} (\downarrow \Gamma_1)\{P\}(\downarrow \Gamma_2)$ where $\downarrow \Gamma_i$ is obtained by mapping all the indirect flow types *i.e.*, $\mathbf{I}(l)$ for some l, to $\mathbf{L}$. By Theorem 1, we conclude the proof. □

5 PROGRAM TRANSFORMATION

The guiding principle of our constant-time program transformations is to take a program with only indirect flow leakage (*i.e.*, typable according to the type system of Figure 4), and transform it into a constant-time program (*i.e.*, typable according to the standard constant-time type system of Figure 2). The core transformation erases the leakage induced by a single conditional. It is iterated until no leakage remains.

In the following, we describe the three program transformations i) scope increase, ii) index sanitising and iii) delayed if-conversion. and prove the security of the transformations.

5.1 Pre-processing of Direct Information Leaks

Our type system rejects programs leaking $\mathbf{H}$ values through memory accesses. We detect those typing failures and pre-process the program using the naive transformation of Section 3 which fixes the information leak at the cost of iterating over all the array indices. Note that this transformation is limited to direct information flow leaks. For indirect flows, we propose novel transformations avoiding the iteration.

5.2 Scope Increase Algorithm

The purpose of the *scope increase* algorithm is to identify a conditional branching over a $\mathbf{H}$ variable, say h, and confine inside the **next** statement of the conditional all the memory accesses which are indirectly leaking the secret h.

Condition Selection. As observed in Section 3, duplicating code in both branches of a $\mathbf{H}$ conditional may introduce spurious information leaks. To avoid this issue, we select the *outermost, rightmost* $\mathbf{H}$ conditional *i.e.*, a conditional which is not within the syntactic scope of another $\mathbf{H}$ conditional and is the last in the textual order. Example 3 illustrates the spurious leaks that would be introduced by a misselection.

Example 3. *Consider the following snippet*

$$\left(\begin{array}{l}\mathbf{if}^{@p_0}\ h_0 \\ \mathbf{then}\ \boldsymbol{if}^{@p_1}\ h_1\ \boldsymbol{then}\ x = 0\ \boldsymbol{else}\ x = 1 \\ \mathbf{else\ skip}\end{array}\right); s$$

where the statement s indirectly leaks both h_1 and h_2. If we select the innermost *conditional h_1, code motion would yield*

$$\begin{array}{l}\mathbf{if}^{@p_0}\ h_0\ \mathbf{then}\ (\boldsymbol{if}^{@p_1}\ h_1\ \boldsymbol{then}\ x = 0\ \boldsymbol{else}\ x = 1\ \boldsymbol{next}\ s) \\ \mathbf{else}\ s\end{array}$$

where the statement s is duplicated and crosses the boundaries of h_1. This may introduces additional potential leakage as s will be typed under the more restrictive security context h_1. On the contrary, if the outermost $\mathbf{H}$ *conditional is selected, we get*

$$\begin{array}{l}\mathbf{if}^{@p_0}\ h_0\ \mathbf{then}\ (\boldsymbol{if}^{@p_1}\ h_1\ \boldsymbol{then}\ x = 0\ \boldsymbol{else}\ x = 1)\ \mathbf{else\ skip} \\ \mathbf{next}\ s\end{array}$$

In that case, s is still typed under the $\mathbf{L}$ *security context.*

Introduction of **next**. Figure 5 presents the *scope increase* algorithm $SI_p : stmt \times stmt \rightarrow stmt$. $SI_p(s_1, s)$ takes a statement s_1 so that p is the rightmost outermost $\mathbf{H}$ conditional and a statement s which is the continuation of the program s_1 *i.e.*, the statement to be executed after s_1. The SI_p algorithm recursively performs code motion until inserting the motioned code in the **next** statement of the $\mathbf{H}$ conditional. Initially, the statement s is **skip**. If the program is the $\mathbf{H}$ conditional with annotation p, SI_p inserts the continuation s as the **next** statement. If this is another conditional with annotation $p' \neq p$, the condition c is necessarily $\mathbf{L}$ and there are two symmetric cases depending on whether the $\mathbf{H}$ conditional is located in the **then** branch (*i.e.*, $p \in s_1$) or in the **else** branch (*i.e.*, $p \in s_2$). W.l.o.g. consider $p \in s_2$. In that case, the continuation s is appended to the statement s_1 of the **then** branch and we call recursively SI_p over the statement s_2 of the **else** branch. If the statement is a sequence of the form $s_1; s_2$, there are two cases depending on whether $p \in s_1$ or $p \in s_2$. If $p \in s_2$, the statement s_1 is kept unchanged and SI_p is recursively called over s_2. If $p \in s_1$, the continuation of s is augmented by s_2 and SI_p is recursively called over s_1. However, if the continuation is **skip**, we can optimise and split s_2 into a pair of statements $(s_l, s_r) \in sep_p(s_2)$ such that s_r does not leak p. Therefore, SI_p is recursively called over s_1 with a reduced continuation s_l which contains all the statements of s_2 which may leak p. For a **for**, the type system ensures that p cannot escape the loop body. As a result, SI_p is recursively called over the loop body s_1 with the continuation **skip**.

Intuitively, after running $SI_p(s_1, s)$, all the indirect flows due to the $\mathbf{H}$ conditional labelled by p within s_1 are localised within the **then**, **else** or **next** statement of the conditional. To formalise this intuition, we devise a strengthened type system which, given a label p, prevents indirect flows from escaping the conditional labelled by p. The typing judgement is of the form $\Delta, \kappa \vdash^{@p} \Gamma\{s\}\Gamma'$. It is

$$\frac{(s_l, s_s) \in sep_p(s) \quad (\mathbf{skip}, t_s) \in sep_p(t)}{(s_l, s_s; t_s) \in sep_p(s;t)} \qquad \frac{(t_l, t_s) \in sep_p(t)}{(s; t_l; t_s) \in sep_p(s;t)} \qquad \frac{p \in leak(s)}{(s, \mathbf{skip}) \in sep_p(s)} \qquad \frac{p \notin leak(s)}{(\mathbf{skip}, s) \in sep_p(s)}$$

$$\begin{array}{lcll}
SI_p(\mathbf{if}^{@p}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, s) & = & \mathbf{if}^{@p}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{next}\ s & \\
SI_p(\mathbf{if}^{@p'}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, s) & = & \mathbf{if}^{@p'}\ c\ \mathbf{then}\ (s_1; s)\ \mathbf{else}\ SI_p(s_2, s) & p \in s_2 \wedge p \neq p' \\
SI_p(\mathbf{if}^{@p}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2', s) & = & \mathbf{if}^{@p'}\ c\ \mathbf{then}\ SI_p(s_1, s)\ \mathbf{else}\ (s_2; s) & p \in s_1 \wedge p \neq p' \\
SI_p(s_1; s_2, s) & = & s_1; SI_p(s_2, s) & p \in s_2 \\
SI_p(s_1; s_2, s) & = & SI_p(s_1, s_2; s) & p \notin s_2 \wedge s \neq \mathbf{skip} \\
SI_p(s_1; s_2, \mathbf{skip}) & = & let\ s_l, s_r \in sep_p(s_2)\ in\ SI_p(s_1, s_l); s_r & p \notin s_2 \\
SI_p(\mathbf{for}\ x\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s_1, s) & = & \mathbf{for}\ x\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ SI_p(s_1, \mathbf{skip}); s &
\end{array}$$

Figure 5: Scope Increase Algorithm

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash c : \tau, r_c \quad \kappa' = \kappa \sqcup (\tau \ltimes_{\mathbf{L}} \mathbf{I}(\{p'\})) \\ \Delta, \kappa' \vdash^{@p} \Gamma\{(s_1)^{r_1}_{g_1}\}\Gamma_3 \quad \Delta, \kappa' \vdash^{@p} \Gamma\{(s_2)^{r_2}_{g_2}\}\Gamma_3 \\ \Delta, \kappa \vdash^{@p} \Gamma\{(s_3)^{r_3}_{g_3}\}\Gamma' \\ r = r_1 \cup r_2 \cup r_3 \cup r_c \quad g = g_1 \cup g_2 \cup g_3 \cup \tau \ltimes_{\emptyset} \{p'\} \\ \Gamma'' = \begin{cases} \uparrow_{\{p\}} \Gamma'\ if\ \tau \neq \mathbf{L} \wedge p = p' \\ \Gamma'\ otherwise \end{cases}\end{array}}{\Delta, \kappa \vdash^{@p} \Gamma\{\mathbf{if}^{@p'}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{next}\ s_3\}\Gamma''}$$

Figure 6: Localised Implicit Flows Typing Rule

obtained from the type system of Figure 4 by keeping all the typing rules except the typing rule for the conditional that is replaced by the typing rule of Figure 6. The typing rule of Figure 6 is very similar to original typing rule. Actually, the typing judgments only differ when the label p' of the condition is p and when the typing of the condition c is not $\mathbf{L}$. In that case, instead of Γ', the final typing environment is $\Gamma'' = \uparrow_{\{p\}} \Gamma'$ which classifies the indirect flows due the current conditional annotated by p.

The security of the *scope increase* algorithm is given by Theorem 3. Essentially, it states that after code motion, the program can be typed using the strengthened type system which classifies the rightmost outermost $\mathbf{H}$ conditional.

Theorem 3 (Security of *Scope Increase*). *Let p be the rightmost (high) condition of c i.e., $RO_p(c)$ and s be a program without any high condition i.e., $high(s) = \emptyset$. Suppose that $c; s$ is well-typed i.e.,*

$$\Delta, \mathbf{L} \vdash \Gamma\{c; s\}\Gamma'$$

and $\forall x, p \notin \Gamma(x)$.

We have that $SI_p(c, s)$ is well-typed with respect to the strengthened type system. More precisely,

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{SI_p(c, s)\} \uparrow_{\{p\}} \Gamma'$$

Proof outline. The proof is by induction over the typing derivation of c. The two main arguments are: i) if $p \notin P$, we can reconstruct a derivation using $\vdash^{@p}$ using the same typing environments; ii) if $high(P) = \emptyset$ and $p \notin leak(P)$, then given a typing judgement $\Delta, \mathbf{L} \vdash \Gamma\{P\}\Gamma'$, we can classify p and get $\Delta, \mathbf{L} \vdash \uparrow_{\{p\}} \Gamma\{P\} \uparrow_{\{p\}} \Gamma'$. □

5.3 Index Sanitising

In order to prevent array out-of-bounds accesses that may happen after *if-conversion* (see Section 3), we instrument the program with dynamic array bounds checks. For our language, this is easily done because array bounds are statically known by design. In a more general setting, the instrumentation is still possible but is more invasive and requires to explicitly pass around the array size using auxiliary variables [28].

The transformation is only applied to the **then** and **else** statements of the $\mathbf{H}$ conditional identified by the SI_p algorithm. Each array access is recursively transformed using the following rule

$$\text{ARR-SAN} \qquad t[i] \rightsquigarrow t[(0 \leq i < size(t))?i{:}0]$$

The rule applies for both array read performed within an expression and array update. In both cases, the index i is replaced by the conditional expression $(0 \leq i < size(t))?i{:}0$ which returns i if the index is in-bounds and returns 0 otherwise. As array sizes are strictly positive, $t[0]$ is always a valid access. As a result, in both cases, we get a valid array access.

Example 4. *Consider the program $P3$ which either performed an array read or array update depending on a $\mathbf{H}$ conditional.*

$$P3 : \boldsymbol{if}^{@p}\ h\ \boldsymbol{then}\ (x = l_1; y = t[x])\ \boldsymbol{else}\ (x = l_2; t[x] = y)$$

The array accesses are executed in a security context $\kappa = \mathbf{I}(\{p\}) \neq \emptyset$. As a result, they need to be instrumented and we get the following code.

$$\begin{array}{l}\mathbf{if}^{@p}\ h\ \mathbf{then}\ (x = l_1; y = t[0 \leq x < size(t)?x{:}0]) \\ \mathbf{else}\ (x = l_2; t[0 \leq x < size(t)?x{:}0] = y)\end{array}$$

Definition 2 (Index Sanitisation). *Consider a program containing a conditional labelled by $@p$ i.e., $P[\boldsymbol{if}^{@p}\ c\ \boldsymbol{then}\ s_1\ \boldsymbol{else}\ s_2]$. The program $IS_p(P[\boldsymbol{if}^{@p}\ c\ \boldsymbol{then}\ s_1\ \boldsymbol{else}\ s_2])$ is of the form*

$$P[\boldsymbol{if}^{@p}\ c\ \boldsymbol{then}\ s_1'\ \boldsymbol{else}\ s_2']$$

where s_1' (resp. s_2') is obtained by applying the rules ARR-SAN to all the array accesses of s_1 (resp. s_2).

Theorem 4 states that the instrumentation of array accesses does not change the typing of the program.

Theorem 4 (Security of index instrumentation). *Let p a program point, and P a program typable for our strengthened type system i.e.,*

$$\Delta, \kappa \vdash^{@p} \Gamma\{P\}\Gamma'$$

for some $\kappa, \Gamma, \Delta, \Gamma'$.

Then, the instrumented program $IS_p(P)$ is also well-typed, and we have

$$\Delta, \kappa \vdash^{@p} \Gamma\{IS_p(P)\}\Gamma'$$

Proof outline. The proof is by induction over P. The main insight of the proof is that the instructions of P that are modified in $IS_p(P)$ are obtained by the rewrite rule arr-san which preserve the typing judgments. To see this, consider an expression e used to index an array t such that $\Gamma \vdash e : \tau$. After instrumentation, we obtain $e' \triangleq 0 \leq e < size(t)?e : 0$. Because $\Gamma \vdash e : \mathbf{L}$ and $\Gamma \vdash size(t) : \mathbf{L}$, we obtain the same typing *i.e.*, $\Gamma \vdash e' : \tau$. □

5.4 Delayed *if-conversion*

The *delayed if-conversion* algorithm takes a **H** condition of the form $\mathbf{if}^{@p}\ h\ \mathbf{then}\ s_t\ \mathbf{else}\ s_e\ \mathbf{next}\ s$ produced by SI_p. By construction, all the h-dependent memory accesses are within either s_t, s_e or s. The result of the transformation is of the form

$$pre; s_t'; s_e'; s'; post.$$

The statement pre makes fresh copies of the scalar variables that are modified in either branch of the conditional. As a result, if x is used in one of the branches, we have $\{x_t = x; x_e = x\} \subseteq pre$ for x_t and x_e fresh variables. An example of such a transformation result was given in Example 2.

The transformation is defined in Figure 7. The function $Nxt_h^{\rho_t,\rho_e}$ generates code that copies variables, according to two renamings ρ_t and ρ_e. In general, ρ_t and ρ_e map the program variables to fresh copies in the **then** and the **else** branch, respectively. The statement s_t' is obtained by applying recursively the renaming ρ_t to s_t. Similarly, s_e' is obtained by applying recursively the renaming ρ_e to s_e. The only twist is for array variables that are not copied. For those, we perform a conditional update predicated by the condition h.

$$Rn_\rho^h(t[e_1] = e_2) = t[\rho(e_1)] = h?\rho(e_2) : t[\rho(e_1)]$$

The transformation of s is more complicated. Intuitively, we keep renaming s with both the renaming inherited from the **then** and **else** branch. However, the renaming is limited to instructions that are secret-dependent and the renaming is dynamically updated to avoid variable clashes. It is crucial *not* to rename expressions that are not altered by the **H** condition, in order to avoid introducing spurious secret-dependent information flows. For an array update, $t[e_1] = e_2$, if the expressions are renamed the same way in both branches, the runtime values are independent from the secret h, and we can avoid predicating the update by the condition h. Hence, we simply generate a renamed array update

$$t[\rho(e_1)] = \rho(e_2)$$

Eventually, at the end of s', it is necessary to merge the copies of variables from both branches using a conditional expression. The whole transformation is given Figure 7 and is explained in more details in the following sections.

5.4.1 Initialisation. After renaming, the **then** and the **else** branch are executed sequentially. To make sure that variables assigned in one branch are not read in another branch, a variable x modified in either the **then** and **else** branch is copied into fresh variables x_t and x_e, to be used in the **then** and the **else** branch, respectively. To this end, we define the function $copy : \mathcal{P}(Var) \rightarrow Var \hookrightarrow Var$. Given a set V, $copy(V)$ returns a renaming ρ such that every variable $x \in V$ is mapped to a variable $x' \in fresh$.

The expression $pre_t \in Seq(\{x_t = x \mid x \in V \wedge \rho_t(x) = x_t\})$ is a statement such that for each variable $x \in V$, there is an assignment $x_t = x$. As all the variables are fresh, the order of the assignment is not relevant. For the **else** branch, the statement pre_e is built in a similar manner.

5.4.2 Branch Renaming. After the initialisation, the function Rn recursively applies the renaming ρ_t (resp. ρ_e) to the statement of the **then** (resp. **else**) branch. The renaming is standard except for the update of array variables. As array variables are not copied, we perform a conditional assignment using the condition variable h and we have

$$Rn_\rho^h(t[e_1] = e_2) = t[\rho(e_1)] = h?\rho(e_2) : t[\rho(e_1)]$$

If we are in the branch for which h holds, the array is updated with the renamed values. Otherwise, if the condition h does not hold, the array is not modified because it is updated using $t[\rho(e_1)]$ which is the previous value. Note that we have the syntactic restriction that the condition variable h is modified by neither the **then** nor the **else** branch.

5.4.3 Simultaneous Renaming of **next***.* The transformation of the statement s in the **next** statement is given by the function Nxt_h of Figure 7. It is more complicated because it requires a simultaneous renaming using both the renaming ρ_t for the **then** branch and the renaming ρ_e for the **else** branch. Consider the case of an assignment $x = v$. If both renaming yield the same expression ($\rho_t(v) = \rho_e(v)$), we have the guarantee that x is independent from the condition h. Therefore, we generate a single statement $x' = \rho(v)$ and update ρ_t and ρ_e so that the current value of x is bound to x'. If the expressions are different ($\rho_t(v) \neq \rho_e(v)$), the value of x may depend of the condition h and therefore we generate two assignments: $x_t = \rho_t(v)$ models the assignment as if it occurs in the **then** branch and $x_e = \rho_e(v)$ models the assignment as if it occurs in the **else** branch. The renaming ρ_t is updated so that x is bound to x_t and the renaming ρ_e is updated so that x is bound to x_e. For the case of an array update $t[e_1] = e_2$, the reasoning is similar. If both expressions are renamed the same way ($\rho_t(e_1) = \rho_e(e_1)$ and $\rho_t(e_2) = \rho_e(e_2)$), we simply generate a renamed array update because the values are independent from the condition h. Otherwise, we generate two array updates predicated by the condition h or its negation $!h$ depending on whether we model the array update in the **then** branch or in the **else** branch. For the sequence, both statements are renamed and the renamings are threaded along.

For the conditional, by construction, we have the guarantee that there is no **next** statement. This is because the only **next** in the program has just been introduced by the SI_p transformation and we are currently processing the generated **next** statement. Both branches are recursively renamed using the same initial renaming maps ρ_t and ρ_e. At the end of the conditional, to reconcile the renaming ρ_t^1 and ρ_t^2 (resp. ρ_e^1 and ρ_e^2) we join the renaming maps $\dot{\rho}_t = \rho_t^1 \bowtie \rho_t^2$ (resp. $\dot{\rho}_e = \rho_e^1 \bowtie \rho_e^2$) which, returns a fresh variable if the renaming maps disagree. To synchronise the program variables with the renaming maps $\dot{\rho}_t$ and $\dot{\rho}_e$, we append to each of the branches a sequences of assignments using the ϕ function. Given ρ and ρ', $\phi(\rho, \rho')$ contains an assignment $\rho(x) = \rho'(x)$ for each variable x such that $\rho(x) \neq \rho'(x)$. For the **for** loop, before renaming the loop body, we update the initial renaming maps ρ_t and ρ_e so

$$\frac{\begin{array}{c} V = mod(s_t) \cup mod(s_e) \quad \rho_t = copy(V) \quad \rho_e = copy(V) \\ pre_t \in Seq(\{x_t = x \mid x \in V \land \rho_t(x) = x_t\}) \qquad pre_e \in Seq(\{x_e = x \mid x \in V \land \rho_e(x) = x_e\}) \\ Nxt_h^{\rho_t,\rho_e}(s) = (\dot{\rho_t}, \dot{\rho_e}, s') \\ post_{\mathbf{L}} \in Seq(\{x = \dot{\rho_t}(x) \mid \dot{\rho_t}(x) = \dot{\rho_e}(x)\}) \qquad post_{\mathbf{H}} \in Seq(\{x = h?\dot{\rho_t}(x) : \dot{\rho_e}(x) \mid \dot{\rho_t}(x) \neq \dot{\rho_e}(x)\}) \end{array}}{Iconv(\mathbf{if}^{@p}\ h\ \mathbf{then}\ s_t\ \mathbf{else}\ s_e\ \mathbf{next}\ s) = pre_t; pre_e; Rn^h_{\rho_t}(s_t); Rn^{!h}_{\rho_e}(s_e); s'; post_{\mathbf{L}}; post_{\mathbf{H}}}$$

$$\begin{array}{lcl} Rn^h_\rho(x = e) & = & \rho(x) = \rho(e) \\ Rn^h_\rho(t[e_1] = e_2) & = & t[\rho(e_1)] = h?\rho(e_2) : t[\rho(e_1)] \\ Rn^h_\rho(s_1; s_2) & = & Rn^h_\rho(s_1); Rn^h_\rho(s_2) \\ Rn^h_\rho(\mathbf{if}^{@p'}\ h'\ \mathbf{then}\ s_t\ \mathbf{else}\ s_e) & = & \mathbf{if}^{@p'}\ \rho(h')\ \mathbf{then}\ Rn^h_\rho(s_t)\ \mathbf{else}\ Rn^h_\rho(s_e) \\ Rn^h_\rho(\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s) & = & \mathbf{for}\ \rho(i)\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ Rn^h_\rho(s) \end{array}$$

$$\begin{array}{lcl} Nxt_h^{\rho_t,\rho_e}(x = v) & = & \left\{ \begin{array}{ll} (\rho_t[x \mapsto x'], \rho_e[x \mapsto x'], x' = \rho_e(v)) & \text{if } \rho_t(v) = \rho_e(v) \\ (\rho_t[x \mapsto x_t], \rho_e[x \mapsto x_e], x_t = \rho_t(v); x_e = \rho_e(v)) & \text{otherwise} \end{array} \right. \\ & & \text{where } x_t \in fresh, x_e \in fresh, x' \in fresh \\ Nxt_h^{\rho_t,\rho_e}(t[e_1] = e_2) & = & \left\{ \begin{array}{ll} (\rho_t, \rho_e, t[\rho_t(e_1)] = \rho_t(e_2)) & \text{if } \bigwedge \begin{array}{l} \rho_t(e_1) = \rho_e(e_1) \\ \rho_t(e_2) = \rho_e(e_2) \end{array} \\ \left(\rho_t, \rho_e, \left(\begin{array}{l} t[\rho_t(e_1)] = h?\rho_t(e_2) : t[\rho_t(e_1)] \\ t[\rho_e(e_1)] = !h?\rho_e(e_2) : t[\rho_e(e_1)] \end{array} \right)\right) & \text{otherwise} \end{array} \right. \\ Nxt_h^{\rho_t,\rho_e}(s_1; s_2) & = & let\ (\rho_t', \rho_e', s_1') = Nxt_h^{\rho_t,\rho_e}(s_1)\ in\ let\ (\rho_t'', \rho_e'', s_2') = Nxt_h^{\rho_t',\rho_e'}(s_2)\ in \\ & & (\rho_t'', \rho_e'', s_1'; s_2') \\ Nxt_h^{\rho_t,\rho_e}(\mathbf{if}^{@p'}\ h'\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) & = & let\ (\rho_t^1, \rho_e^1, s_1') = Nxt_h^{\rho_t,\rho_e}(s_1)\ and\ (\rho_t^2, \rho_e^2, s_2') = Nxt_h^{\rho_t,\rho_e}(s_2)\ in \\ & & let\ \dot{\rho_t} = \rho_t^1 \bowtie \rho_t^2\ and\ \dot{\rho_e} = \rho_e^1 \bowtie \rho_e^2\ in \\ & & (\dot{\rho_t}, \dot{\rho_e}, \mathbf{if}^{@p'}\ \rho_t(h')\ \mathbf{then}\ (s_1'; \phi(\dot{\rho_t}, \rho_t^1); \phi(\dot{\rho_e}, \rho_e^1))\ \mathbf{else}\ (s_2'; \phi(\dot{\rho_t}, \rho_t^2); \phi(\dot{\rho_e}, \rho_e^2))) \\ Nxt_h^{\rho_t,\rho_e}(\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s) & = & let\ \rho_t' = \rho_t[x \mapsto x' | x' \in fresh \land x \in mod(s)]\ in \\ & & let\ \rho_e' = \rho_e[x \mapsto x' | x' \in fresh \land x \in mod(s)]\ in \\ & & let\ (\rho_t'', \rho_e'', s') = Nxt_h^{\rho_t',\rho_e'}(s)\ in \\ & & (\rho_t', \rho_e', \phi(\rho_t', \rho_t); \phi(\rho_e', \rho_e); \mathbf{for}\ \rho_t(i)\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ (s'; \phi(\rho_t', \rho_t''); \phi(\rho_e', \rho_e''))) \end{array}$$

$$\begin{array}{r} mod(S) = \{x \mid x \text{ is a variable modified by the statement S}\} \\ Seq(S) = \{s_1; \ldots; s_n \mid \cup_i s_i = S \land n = |S|\} \\ \phi(\rho, \rho') \in Seq\{\rho(x) = \rho'(x) \mid x \in \rho(x) \land \rho(x) \neq \rho'(x)\} \\ \forall x. \rho_1 \bowtie \rho_2(x) = \left\{ \begin{array}{ll} \rho_1(x) & \mathbf{if}\ \rho_1(x) = \rho_2(x) \\ x' \text{ where } x' \in fresh & \mathbf{otherwise} \end{array} \right. \end{array}$$

Figure 7: Delayed if-conversion

that each variable of the loop body is given a fresh variable. The loop body s is renamed using the obtained renaming maps ρ_t' and ρ_e'. In order to synchronise the renaming maps with the program variables, we insert ϕ functions before and after the renaming of the loop body s'. This is needed to ensure that the variable names are coherent for the next loop iteration.

5.4.4 Finalisation. The last step of the transformation is performed by the statements $post_{\mathbf{L}}$ and $post_{\mathbf{H}}$. For each variable x that is given the same renaming after the **next** statement *i.e.*, $\dot{\rho_t}(x) = \dot{\rho_e}(x) = x'$, there is an assignment $x = x' \in post_{\mathbf{L}}$. If the renaming differs, *i.e.*, $\dot{\rho_t}(x) = x_t$ and $\dot{\rho_e}(x) = x_e$ for $x_t \neq x_e$, there is conditional assignment $x = h?x_t : x_e \in post_{\mathbf{H}}$. As a result, the transformation is local to the **H** condition of interest and the implicit flow is transformed in a direct flow outside the scope of the condition.

Definition 3 (Delayed if-conversion). *Consider a program of the form $P = P'[\boldsymbol{if}^{@p}\ h\ \boldsymbol{then}\ s_1\ \boldsymbol{else}\ s_2\ \boldsymbol{next}\ s_3]$ containing a conditional labelled by @p. The program obtained after delayed if-conversion is defined by*

$$IConv_p(P) \triangleq P'[IConv(\boldsymbol{if}^{@p}\ c\ \boldsymbol{then}\ s_1\ \boldsymbol{else}\ s_2\ \boldsymbol{next}\ s_3)]$$

Theorem 5 states that the transformation of a **H** conditional labelled by p is still typable in the original flow tracking type system.

Theorem 5 (Security of delayed if conversion). *Let $C = \boldsymbol{if}^{@p}\ h\ \boldsymbol{then}\ s_1\ \boldsymbol{else}\ s_2\ \boldsymbol{next}\ s_3$ be a conditional, and $P = P'[C]$ a well-typed program containing C i.e.,*

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{P\}\Gamma'$$

for some Δ, Γ and Γ'. The transformed program $IConv_p(P)$ is still well-typed and there exists Γ'' such that

$$\Delta, \mathbf{L} \vdash \Gamma\{IConv_p(P)\}\Gamma''$$

Moreover, $high(IConv_p(P)) \subseteq high(P) \setminus \{p\}$

Proof outline. The proof is by induction over the typing derivation. For the program P' containing the condition C, the typing derivation can be rebuilt easily because P' is not modified. For the condition C, the proof relies on the fact, for modified variables, the renaming acts on copies of the variables and the array access are predicated. This ensures that we can construct a typing derivation for $IConv(C)$. □

5.5 Constant-Time Transformation

The constant-time transformation consists in iterating the previous transformations *i.e.*, SI_p, IS_p and $IConv_p$, until the program does not contain a single **H** condition.

Definition 4 (Constant-Time Transformation). *Let T be the function removing (if it exists) the rightmost outermost* **H** *conditional of a program P.*

$$T(P) = \begin{cases} IConv_p(IS_p(SI_p(P))) & \textbf{\textit{if}}\ RO_p(P)\ \textit{for some}\ p \\ P & \textbf{\textit{otherwise}} \end{cases}$$

For a program P, the Constant-Time Transformation $CTT(P)$ iterates the function T until there is no **H** *conditional left.*

$$CTT(P) = \begin{cases} P & \textbf{\textit{if}}\ T(P) = P \\ CTT(T(P)) & \textbf{\textit{otherwise}} \end{cases}$$

To summarise, we consider a program P that is well-typed according to our information flow tracking type-system *i.e.*, $\Delta, \mathbf{L} \vdash \Gamma\{P_g^r\}\Gamma'$. At each iteration of the CTT transformation, we select the *rightmost*, *outermost* **H** conditional of program P, say p. By construction, we have that $p \in g$. The *scope increase* transformation (see Section 5.2) identifies the indirect flows that leak outside the scope of the condition and perform code motion to install a **next** statement. The array accesses within the statements of the conditions are then protected using the index sanitising transformation (see Section 5.3) and the **H** condition of program point p is removed using *delayed if-conversion* (see Section 5.4). As each iteration of the transformation removes a **H** condition, the transformation terminates. Theorem 6 states that the resulting program is constant-time according to the constant-time type system.

Theorem 6 (Constant-Time enforcement). *Let P be a typable program for a* **L** *security context and* simple *typing environments Γ, Γ' and Δ, i.e.,*

$$\Delta, \mathbf{L} \vdash \Gamma\{P\}\Gamma'$$

Then we have that $CTT(P)$ is typable with an empty guard i.e.,

$$\Delta, \mathbf{L} \vdash \Gamma\{CTT(P)_\emptyset^\emptyset\}\Gamma''.$$

Proof outline. The proof is by induction over the number of **H** conditionals of P *i.e.*, $|high(P)|$. Suppose a well-typed program $\Delta, \mathbf{L} \vdash \Gamma\{P\}\Gamma'$.

- $high(P) \neq \emptyset$. There is a **H** conditional p such that $RO_p(P)$. After the scope increase pass, by Theorem 3, we obtain
$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{SI_p(P, \mathbf{skip})\} \uparrow_{\{p\}} \Gamma'$$
After the index sanitise pass, by Theorem 4, we obtain
$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{IS_p(SI_p(P, \mathbf{skip}))\} \uparrow_{\{p\}} \Gamma'$$
After delayed if-conversion, by Theorem 5, we get a well-typed program $P' = IConv_p(IS_p(SI_p(P, \mathbf{skip})))$ such that $high(P') \subset high(P)$ because at least the **H** conditional p is removed. The proof follows by induction hypothesis.
- $high(P) = \emptyset$. There are no **H** conditionals and therefore no indirect flows. As the program is well-typed, there are no **H** array accesses. It follows that all the array accesses are **L** accesses and this concludes the proof.

□

6 IMPLEMENTATION AND EXPERIMENTS

We have implemented and tested our constant-time enforcement transformation as a pass in the Jasmin compiler [3, 5].

6.1 Implementation

The code of the transformations amounts to around 4 KLOC in the Gallina language of the Coq proof assistant. An advantage of using Jasmin is that the constant-time property that we enforce at source level is preserved at assembly level. We can actually check this *a posteriori* by using Jasmin's Constant-Time type checker. Moreover, as Jasmin is built upon a simple imperative language, there is little gap between our formal model and the real implementation. Because we do not have a formal support for function calls, we transform each function at a time, before the aggressive inlining pass of Jasmin. To allow for this, we over-approximate the output typing of a function from $\mathbf{I}(l)$ to **H**.

6.2 Experiments

We evaluate our constant-time enforcement transformation on simple but challenging programs that illustrate the expressiveness of our constant-time enforcement transformations. They include the following programs which are taken from the FaCT test suite

- BranchRemoval : $\mathbf{if}^{@p}\ h\ \mathbf{then}\ x = l_1\ \mathbf{else}\ x = l_2$
- PotentialOOB : $\mathbf{if}^{@p}\ h\ \mathbf{then}\ t[x] = 0\ \mathbf{else\ skip}$
- ReturnDeferall : $\mathbf{if}^{@p}\ h\ \mathbf{then}\ \mathit{return}\ x\ \mathbf{else\ skip}$

We also include the motivating examples of the current paper *i.e.*, $P0$ of Example 2 and $P2$ from Section 3, together with hand-crafted programs (see Figure 10) as well as bubble sort. These programs are detailed in table 9. Programs P3, P4, P5 and P7 need the scope-increase algorithm, while P6 require the naive transformation to get rid of the problematic memory access. We also include the `cswap` function which is used by the existing implementation of Curve25519 in Jasmin from *libjc*[2] [5]. More precisely, we have rewritten the existing constant-time `cswap` to the more natural non constant-time version.

6.2.1 Metrics.

In Figure 9, we evaluate each one of these programs, and their transformations, using the following metrics :

a) Constant-Time: We check whether the program is successfully transformed and if the transformed version is indeed Constant-Time according to the type checker of Jasmin.
b) Code Size Overhead: We provide the size of the initial code and its size after the transformation in terms of number of statements.

[2] https://github.com/tfaoliveira/libjc

Program	Constant-Time		Source code size		Variables - Source		Variables - Compiled		Compilation Time		Assembly size (CompCert)		Assembly Size (gcc -O3)	
	FaCT	Ours	Input	Output	Input	Output	C[Input]	C[Output]	Classic	Transformed	Classic	Transformed	Classic	Transformed
BranchRemoval	✓	✓	3	8	1	3	6	6	0.003	0.016	19	19	24	24
PotentialOOB	~	✓	3	5	1	2	6	6	0.004	0.011	22	25	26	34
ReturnDeferral	✓	×	–	–	–	–	–	–	–	–	–	–	–	–
`cswap`	✓	✓	27	45	10	21	6	7	0.007	0.212	61	136	75	82
BubbleSort	✓	✓	8	12	4	6	7	8	0.268	2.126	42	52	53	56
$P0$	×	✓	9	19	2	7	6	7	0.004	0.028	27	43	31	31
$P2$	×	✓	8	17	3	7	6	7	0.004	0.036	24	27	27	28
$P3$	×	✓	7	16	2	7	6	7	0.006	0.023	21	24	25	26
$P4$	×	✓	11	46	3	18	6	12	0.004	0.093	28	94	31	67
$P5$	×	✓	10	24	3	10	6	9	0.005	0.025	27	24	26	29
$P6$	×	✓	8	14	3	7	6	7	0.005	0.056	27	37	25	45
$P7$	×	✓	8	17	3	8	6	8	0.007	0.034	28	25	26	24

Figure 8: Case-study of our transformation

Program	Description
cswap	Swap function from Curve25519
BubbleSort	Standard bubble sort algorithm
P3	Conditionnal access
P4	Two imbricated conditionnals
P5	Two sequentials ifs
P6	For loop followed by a memory access
P7	For loop containing a memory access

Figure 9: Description of test cases

c) Number of Variables: We also provide the number of variables used by the program, at source level, and after transformation but before optimisations. We also provide the number of variables after optimisations just before assembly generation.
d) Compilation Time: We provide the time taken by Jasmin to complete the compilation of the program, whether the transformation is enabled or not, in seconds.
e) Assembly Size: The size of the compiled code, before and after transformations. Due to restrictions on the Jasmin compiler, our introduction of complex expressions sometimes prevents the compilation to terminate. To evaluate our compiled code, we export the resulting high-level program to C, and compile it using CompCert, thus preserving the Constant-Time property. As to compensate CompCert's lack of optimisation, we also compile using GCC -O3.

The results of our evaluation are summarized in Figure 8.

6.2.2 *Evaluation of Results.*

Constant-Time Property. We are able to transform all the benchmarks except ReturnDeferall which is rejected because Jasmin only accept a *return* as the last instruction. For PotentialOOB, our generated program is different from FaCT which inserts an *assume* statement to ensure safety. Instead, we instrument the array access and get safety for free. Yet, our transformation is only semantically correct if the initial program has no array of bound access. Programs P0, P2, P3, P4, P5, P6 and P7 are rejected by FaCT but accepted by our enhanced transformation at the cost of some code duplication.

$$\begin{array}{l} cswap : \mathbf{if}^{@p}\ swap\ \mathbf{then} \\ \quad\quad \mathbf{for}\ i\ \mathbf{from}\ 0\ \mathbf{to}\ 4\ \mathbf{do} \\ \quad\quad\quad tmp = z2p[i];\ z2p[i] = z3p[i];\ z3p[i] = tmp; \\ \quad\quad\quad tmp = x2[i];\ x2[i] = x3p[i];\ x3p[i] = tmp; \\ \quad \mathbf{else\ skip} \end{array}$$

$$P3 : (\mathbf{if}^{@p}\ h\ \mathbf{then}\ x = l_1\ \mathbf{else}\ x = l_2); y = t[x];$$

$$P4 : \left(\begin{array}{l} \mathbf{if}^{@p}\ h\ \mathbf{then} \\ \left(\begin{array}{l} \mathbf{if}^{@p'}\ h'\ \mathbf{then} \\ \ x = l_1 \\ \mathbf{else}\ x = l_2 \end{array}\right); y = t[x] \\ \mathbf{else}\ y = l_3 \end{array}\right); t[y] = l_4;$$

$$P5 : \left(\begin{array}{l} \mathbf{if}^{@p}\ h\ \mathbf{then} \\ \ r = 0 \\ \mathbf{else}\ r = 1 \end{array}\right); \left(\begin{array}{l} \mathbf{if}^{@p'}\ r\ \mathbf{then} \\ \ x = 1 \\ \mathbf{else}\ x = 2 \end{array}\right); y = t[x];$$

$$P6 : \left(\mathbf{for}\ i\ \mathbf{from}\ c1\ \mathbf{to}\ c2\ \mathbf{do}\ \begin{array}{l} \mathbf{if}^{@p}\ h\ \mathbf{then} \\ \ x = l_1 \\ \mathbf{else}\ x = l_2 \end{array}\right); y = t[x];$$

$$P7 : \left(\mathbf{for}\ i\ \mathbf{from}\ c1\ \mathbf{to}\ c2\ \mathbf{do}\ \begin{array}{l} \mathbf{if}^{@p}\ h\ \mathbf{then} \\ \ x = l_1 \\ \mathbf{else}\ x = l_2 \end{array}\ ; y = t[x]\right);$$

Figure 10: Example programs

Code Size Overhead. For most of the benchmark, the resulting source code is around twice the size of the original. This observation is true for programs containing at most *1* imbricated conditionnal : the code duplication pass is only applied once.

For other programs, such as $P4$, the overhead is proportional to 2^n, with n the *depth* of the program. In the case of $P4$, the $t[y] = l_4$ instruction is duplicated by the first `if-conv` pass, *inserted* into the **next** for the p conditional, and later duplicated again. This repeats at every level of nesting in the program.

Number of Variables - Source. By the same reasoning as above, the number of variables in the transformed code, before compilation, is around 2^n times the size of the original program, with n the depth of the program.

Number of Variables - Compiled. The JASMIN compiler applies a number of aggressive optimisations. To evaluate the impact of optimisations on the transformed code, we also compare the number of written variables with and without constant-time enforcement.

For most of the programs, the variables overhead is reduced to 1 or 2 and is almost insignificant. However, for program such as $P4$, where there are more than one level, the overhead is around n times the initial source code, with n the depth of the program.

Compilation Time. For most of the programs, we have at most one order of magnitude added by the compilation of the transformed program. However, when a secret conditional is within a loop, our variable overhead is demultiplied by the loop unrolling of Jasmin, resulting in greater compilation time, although it stays reasonable.

Asssembly Size. For most of our benchmark, the resulting assembly code using CompCert does not differ by much in size. Notable exceptions are `cswap` and P4. The one common factor between these two code snippets is the introduction of multiples **cmove** instructions. The construction of the expressions used within those instructions provoke a significant overhead in code size. When compiling using all optimisations offered by gcc, we don't notice such a high overhead anymore, except for P4. Overall, both our transformation, whether the compilation method used, struggles with imbricated conditionals, but offers satisfying results on other programs.

Summary of Evaluation. Our type-directed transformation allows for more programs to be transformed into a Constant-Time semantically-equivalent version than FaCT. This transformation implies a performance and size overhead at most doubles in performance and size. Subsequent compiler optimising passes remove most of the increase in code size and number of variables used.

7 RELATED WORK

The constant-time property can be verified at different level ranging from assembly [8, 9], intermediate code [4, 27] to source level [13] using a taint analysis. One difficulty is to precisely model aliases and avoid false alarms. The logic implemented by our type system is simpler but is sufficiently precise to analyse JASMIN programs [3] which are equipped with a functional semantics. In particular, it is the role of the JASMIN compiler to ensure that arrays are not aliases and that memory updates can be performed in-place.

Agat [1] pioneered type-directed program transformations to eliminate information leaks. The transformation ensures that both branches of a **H** conditional perform the same amount of computations by inserting dummy instructions. Using unification, Köpf and Mantel [23] propose an enhanced type-system reducing the amount of dummy operations. The constant-time programming discipline is a stronger guarantee. It protects against micro-architectural timing leaks for which inserting dummy computations is not an effective countermeasure. As a result, our transformation is more aggressive and completely removes **H** conditionals. SC ELIMINATOR [32] uses a taint analysis to detect and repair information flow leaks. They eliminate **H** conditionals using a *if-conversion*, similar to FaCT [16], where each statement is predicated by the conditional. Their approach has been improved by Sores and Pereira [28] to avoid generating programs with out-of-bound accesses. JASMIN arrays have a known size, which simplifies our instrumentation of array accesses. Our delayed *if-conversion* is more sophisticated and allows a more efficient transformation of array accesses that are secret due to indirect flows.

Domain Specific Languages (DSL) have been designed to program and verify constant-time cryptographic algorithms. VALE [14, 17] is a high-level assembly language for programming and verifying cryptographic algorithms using either DAFNY [21] or F* [29]. To prove the constant-time property, they implement in F* a proved-correct taint analysis. HACL* [33] is a verified cryptographic library programmed and proved in F* and compiled with the KREMLIN compiler [25] which preserves the constant-time property. JASMIN [3, 5] is another language for programming cryptographic algorithms. The constant-time proof is obtained by embedding the JASMIN language in the EASYCRYPT proof-assistant [11] and preserved by the JASMIN compiler. In our work, benefit from the infrastructure of the existing JASMIN compiler. We implement a front-end compiler pass which ensures that a well-typed program is transformed into a constant-time program.

FACT [15, 16] (described in Sec. 2.4) use type-based transformations to ensure constant-time. We employ a more permissive type system and additional, non-local program transformations such as *scope increase*. This means that there are programs that we can transform to constant-time which FACT would reject. On the other hand, FACT handles language features that we do not deal with, in particular the deferral of early returns from functions. Another difference is our handling of so-called *public safety*. FACT augments the type system with partial verification conditions, whereas we assume the safety of the input program and instrument array accesses, at the cost of some potential overhead.

8 CONCLUSION

We have proposed type-directed transformations which turn a potentially insecure program into a more secure constant-time program. A key insight of the approach is that programs with indirect secret flows can always be transformed into semantically equivalent constant-time programs at the cost of duplicating problematic array accesses with the scope of the conditional and performing delayed *if-conversion*. The transformations subsume those offered by the state-of-the-art tool FACT. In addition, our type system enables transformations that are not available FACT, at the cost of increasing the size of the generated code. Furthermore, the transformations are fully automatic, and do not require prior annotations of the program. Experiments with the Jasmin compiler shows that the increase in code size can be mitigated by subsequent compiler optimisations.

Our type system does not protect against speculative leaks *e.g.* Spectre [19]. A promising approach is to perform speculative post-analysis à la BLADE [30] over the constant-time program. As future work, we will extend our type system to cope with function calls. We will also investigate how to get a more permissive type system. A challenge would be to automatically transform **while** loops with **H** conditions.

REFERENCES

[1] Johan Agat. 2000. Transforming Out Timing Leaks. In *POPL*, Mark N. Wegman and Thomas W. Reps (Eds.). ACM, 40–53. https://doi.org/10.1145/325694.325702

[2] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe D. Warren. 1983. Conversion of Control Dependence to Data Dependence. In *POPL*, John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum (Eds.). ACM Press, 177–189. https://doi.org/10.1145/567067.567085

[3] 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 *CCS*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. https://doi.org/10.1145/3133956.3134078

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 53–70. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[5] 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 *S&P*. IEEE, 965–982. https://doi.org/10.1109/SP40000.2020.00028

[6] Jean-Philippe Aumasson et al. 2020. *Cryptocoding*. https://github.com/veorq/cryptocoding

[7] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. 2019. System-Level Non-interference of Constant-Time Cryptography. Part I: Model. *J. Autom. Reason.* 63, 1 (2019), 1–51. https://doi.org/10.1007/s10817-017-9441-5

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna, and David Pichardie. 2020. System-Level Non-interference of Constant-Time Cryptography. Part II: Verified Static Analysis and Stealth Memory. *J. Autom. Reason.* 64, 8 (2020), 1685–1729. https://doi.org/10.1007/s10817-020-09548-x

[9] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *CCS*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1267–1279. https://doi.org/10.1145/2660267.2660283

[10] 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. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. https://doi.org/10.1145/3371075

[11] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *FOSAD Tutorial Lectures (LNCS)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.), Vol. 8604. Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

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

[13] Sandrine Blazy, David Pichardie, and Alix Trieu. 2019. Verifying constant-time implementations by abstract interpretation. *J. Comput. Secur.* 27, 1 (2019), 137–163. https://doi.org/10.3233/JCS-181136

[14] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 917–934. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

[15] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *IEEE Cybersecurity Development, SecDev 2017*. IEEE Computer Society, 69–76. https://doi.org/10.1109/SecDev.2017.24

[16] 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 *PLDI*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 174–189. https://doi.org/10.1145/3314221.3314605

[17] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A verified, efficient embedding of a verifiable assembly language. *Proc. ACM Program. Lang.* 3, POPL (2019), 63:1–63:30. https://doi.org/10.1145/3290376

[18] Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. In *POPL*. ACM, 79–90. https://doi.org/10.1145/1111037.1111045

[19] 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. 2020. Spectre attacks: exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101. https://doi.org/10.1145/3399742

[20] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *16th Annual International Cryptology Conference (LNCS)*, Neal Koblitz (Ed.), Vol. 1109. Springer, 104–113. https://doi.org/10.1007/3-540-68697-5_9

[21] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[22] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *CSF*. IEEE Computer Society, 51–65. https://doi.org/10.1109/CSF.2013.11

[23] Heiko Mantel and Artem Starostin. 2015. Transforming Out Timing Leaks, More or Less. In *ESORICS (LNCS)*, Vol. 9326. Springer, 447–467.

[24] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC (LNCS)*, Dongho Won and Seungjoo Kim (Eds.), Vol. 3935. Springer, 156–168. https://doi.org/10.1007/11734727_14

[25] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *Proc. ACM Program. Lang.* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

[26] Gautier Raimondi. 2023. *Secure compilation against side-channel attacks.* Ph.D. Dissertation. Université de Rennes.

[27] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse representation of implicit flows with applications to side-channel detection. In *CC*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 110–120. https://doi.org/10.1145/2892208.2892230

[28] Luigi Soares and Fernando Magno Quintão Pereira. 2021. Memory-Safe Elimination of Side Channels. In *IEEE/ACM CGO*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 200–210. https://doi.org/10.1109/CGO51591.2021.9370305

[29] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *POPL*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

[30] 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

[31] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304

[32] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *ISSTA*, Frank Tip and Eric Bodden (Eds.). ACM, 15–26. https://doi.org/10.1145/3213846.3213851

[33] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *CCS*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1789–1806. https://doi.org/10.1145/3133956.3134043