Type-directed Program Transformation for Constant-Time Enforcement
Frédéric Besson, Thomas Jensen, Gautier Raimondi

To cite this version:

HAL Id: hal-04268830
https://inria.hal.science/hal-04268830
Submitted on 2 Nov 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
Type-directed Program Transformation for Constant-Time Enforcement

Frédéric Besson
Inria, Univ Rennes
Rennes, France

Thomas Jensen
Inria, Univ Rennes
Rennes, France

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:

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.

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, Baffier 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:

```
expr ≝ e :: x | c | e₁ + e₂ | e₁ ? e₂ → e₃ | t[e]
stmt ≝ skip | x = e | t[e₁] = e₂ | S₁ : S₂
      | if@p x then s₁ else s₂ next s₃
      | for x from e₁ to e₂ do s
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPDP 2023, October 22–23, 2023, Lisboa, Portugal.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0812-1/23/10...
https://doi.org/10.1145/3610612.3610618
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 sizes 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, if $e$ 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 conditional. We also write if $@p$ e then $s_1$ else $s_2$ for 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 $p \in \mathcal{L}$ 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

$$(P, \tau) \Downarrow^t \tau'$$

where $P$ is a program, $\tau$ is an environment mapping variables $x$ and arrays $a$ to their values, $t$ is a leakage trace and $\tau'$ 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 $t$.

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., $e_1 ? e_2 : e_3 \equiv e_1 \& e_2 + 1 \& e_3$ 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.

---

**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 if, the following non-interference property holds:

$$CT(P, L) \equiv \forall x \in L, \sigma(x) = \sigma'(x).$$

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 Constant-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 $\Delta \vdashct \Gamma(p) \Gamma'$. The typing environments $\Gamma, \Gamma', \Delta : \text{Var} \rightarrow \{\mathcal{H}, \mathcal{L}\}$ map a program variable $x$ to its type $t \in \{\mathcal{H}, \mathcal{L}\}$. $\Gamma$ and $\Gamma'$ assign types to scalar variables while $\Delta$ assigns types to array variables. As the type system is flow-sensitive for scalar variables, $\Gamma$ is the typing environment before running $P$ and $\Gamma'$ is the typing environment obtained after running $P$. The typing environments for arrays $\Delta$ is not flow-sensitive. The rationale is that, unlike a variable assignment, an array update would be modelled as a `weak update` and therefore

---

1. The array variable is not explicitly leaked because it can be reconstructed as control-flow decisions are also leaked.
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 \cite{volpano1996information,smith1997information}, 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.

\[
\begin{align*}
\Delta, \Gamma \vdash e : \tau & \implies \Delta, \Gamma \vdash e \text{ true} : \tau \\
\Delta, \Gamma \vdash e_{1} : \gamma_{1} & \implies \Delta, \Gamma \vdash e_2 : \gamma_{2} \\
\Delta, \Gamma \vdash e_{1} \circ e_{2} : \gamma_{1} \cap \gamma_{2} & \implies \Delta, \Gamma \vdash e \text{ true} : \gamma_{1} \cap \gamma_{2} \\
\Delta, \Gamma \vdash c_{\text{skip}} : \tau & \implies \Delta, \Gamma \vdash \text{skip} : \tau \\
\Delta, \Gamma \vdash e \text{ true} \Gamma \{ \text{skip} \} & \implies \Delta, \Gamma \vdash \text{false} \Gamma \{ \text{skip} \} \\
\Delta, \Gamma \vdash x = e \Gamma \{ x \leftrightarrow \tau \} & \implies \Delta, \Gamma \vdash c_{\text{skip}} \Gamma \{ x \leftrightarrow \tau \} \\
\Delta, \Gamma \vdash c \text{ true} \Gamma \{ \text{skip} \} & \implies \Delta, \Gamma \vdash c \text{ true} \{ \text{skip} \} \\
\Delta, \Gamma \vdash \{ \text{if} \theta \text{ then } c_{1} \text{ else } c_{2} \text{ do } s \} \Gamma & \implies \Delta, \Gamma \vdash \{ \text{if} \theta \text{ then } c_{1} \text{ else } c_{2} \text{ do } s \} \Gamma \\
\end{align*}
\]

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

\[
\Delta, \Gamma \vdash \Gamma[P] \Gamma' \\
\] We have that $CT(P, L) = \{ x \mid \Gamma(x) = L \lor \Delta(x) = L \}$. 

2.4 Program Transformations of FACT

FaCT \cite{volpano1997information} 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 \cite{smith1997information}. 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* \cite{giblin1998automatic} 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 $l_2$.

\[
\begin{align*}
\text{if } h \text{ then } x = l_1 & \quad \text{else } x = l_2
\end{align*}
\]

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 $l_1$ performs a memory access $e.g., t[l]$ and that the condition $h$ guards against out-of-bound accesses $i.e., h \leq i < \text{size}(t)$. 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 $P_0$.

\[
P_0: \text{if} \theta \text{ then } (x = l_1; y = t[x]) \text{ else } (x = l_2; y = t[x])
\]

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.

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

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.

for \( i \) from 0 to size(t)−1 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.

\[
\begin{align*}
x_1 &= l_1; y_1 = t[x_1]; x_2 = l_2; y_2 = t[x_2]; \\
x &= h?x_1 : e; y = h?y_1 : y_2.
\end{align*}
\]

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 \( P1' \).

\[
P1 : \text{if}^\oplus h \text{ then } (x = l_1; t[x] = 0) \text{ else } (x = l_2; t[x] = 1)
\]

\[
P1' : \begin{align*}
x_1 &= l_1; t[x_1] = h?0:t[x_1] \\
x_2 &= l_2; t[x_2] = h?1:t[x_2] \\
x &= h?x_1 : e
\end{align*}
\]

As \( x_1 \) and \( x_2 \) 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 \( t[x] \).

\[
P2 : \text{if}^\oplus h \text{ then } x = l_1 \text{ else } x = l_2; (t[l_1] = l_4; y = t[x])
\]

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 \( t[l_1] = l_4 \) 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:

\[
\text{if}^\oplus h \text{ then } x = l_1 \text{ else } x = l_2 \text{ next } t[l_1] = l_4; y = t[x];
\]

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 \( t[l_1] = l_4 \) are kept unchanged. For our example P2, we obtain the code in Figure 3. The resulting code is constant-time.

\[
x_1 = l_1; x_2 = l_2; t[l_1] = l_4; \\
y_1 = t[x_1]; y_2 = t[x_2]; \\
x = h?x_1 : e; y = h?y_1 ; y_2
\]

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 \( t[0 \leq x < size(t)]x : 0 \). Because the program is safe, we have the invariant that \( 0 \leq x < size(t) \), 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 \( \{H, L\} \) lattice with information flow types of form \( I(l) \) where \( l \subseteq L \) is a subset of program locations. The type \( I(l) \) 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”.

\[
\text{IFType} = \mathcal{P}(L) \cup \{H\}
\]

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

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

\[\Delta, \Gamma \vdash x : \Gamma(x), \emptyset\]
\[\Delta, \Gamma \vdash i : L, \emptyset\]
\[\Delta, \Gamma \vdash t[e] : \Delta(t) \cup \tau, \tau \cup I'\]
\[\Delta, \Gamma \vdash e_1 : \tau_1, I_1\]
\[\Delta, \Gamma \vdash e_2 : \tau_2, I_2\]
\[\Delta, \Gamma \vdash e_1 \odot e_2 : [I_{1} \cap \tau_{1} \cup I_{2} \cup \tau_{2}]\]
\[\Delta, \Gamma \vdash e_1 : \tau_i, I_i\]
\[\Delta, \Gamma \vdash e_2 : \tau_j, I_j\]

Statements:

\[\Delta, \kappa \vdash \Gamma((s_1)_{g_1})\Gamma_1\]
\[\tau = r_1 \cup r_2\]
\[g = g_1 \cup g_2\]
\[\Delta, \kappa \vdash \Gamma((s_2)_{g_2})\Gamma'\]
\[\Delta, \kappa \vdash \Gamma((x = e)_{g_2})\Gamma'\]

\[\Delta, \kappa \vdash \Gamma((\text{skip})_{g_2})\Gamma\]

\[\Delta, \kappa \vdash \Gamma((\text{if} \theta \circ \Gamma (\text{then}) s_1 \text{ else } s_2 \text{ next } s_3)_{g_2})\Gamma'\]

\[\bigcap_{\text{cond}(s)} \Gamma' = \Gamma'\]
\[\Delta, \kappa \vdash \Gamma((\text{for} i \text{ from } c_1 \text{ to } c_2 \text{ do } s_3)_{g_2})\Gamma'\]

where

\[\tau \leftarrow d \leftarrow v = \begin{cases} 
    d & \text{if } \tau = L \\
    v & \text{otherwise}
\end{cases}\]

\[(\Gamma \Gamma)(x) = \begin{cases} 
    \Gamma(x) & \text{if } \Gamma(x) = \Gamma'(l) \land I \land I' = \emptyset \\
    H & \text{otherwise}
\end{cases}\]

\[\text{cond}(s) = \{ p \mid \text{if} \theta \circ p \text{ c then } s_1 \text{ else } s_2 \text{ next } s_3 \in s\}\]

Figure 4: Flow Tracking Type System

execution. The typing judgments operate on annotated programs of form \((P')_{g}\) and are of the form

\[\Delta, \kappa \vdash \Gamma((P')_{g}) \Gamma'\]

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

The program annotations in \((P')_{g}\) consists of two sets, \(g\) and \(r\). The set \(g \subseteq L\) over-approximates the secret conditionals in \(P\). The set \(r \subseteq L\) 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 \(H\) is ruled out by the type system). In the rest, we write high((\(P')_{g}\)) for the set \(g\) i.e., the set of \(H\) conditionals within the program \(P\) and we write leak((\(P')_{g}\)) for the set \(r\) i.e., the set of \(H\) 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

\[\Delta, \Gamma \vdash e : \tau, I\]

where \(\tau\) is the security type of the result and \(I\) 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 \(I'(l)\) i.e., strictly below \(H\). If it is \(L\), 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 \text{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 \(\tau\) of the expression \(e\) and the type of the security context \(\kappa\). The rule for array update is flow insensitive. It checks that the type \(\tau_1\) of the index is not \(H\). It also checks that the type obtained from the type of the index \(\tau_1\) and the type of the written value \(\tau_2\) and an upgraded security context \(\kappa \cup \tau_2\) are below the type \(\Delta(t)\) of the array. Therefore, if the security context \(\kappa\) is \(L\), we have \(\tau_1 \cup \tau_2 \subseteq \Delta(t)\). Otherwise, if the array update is performed under a security context \(\kappa \neq L\), the typing constraints entail that \(\Delta(t) = H\). The \(\tau\) annotation is updated to reflect that an array access with an index of type \(\tau_1\) has been made.

The rule for conditions computes the security level \(\tau\) of the condition \(c\). If \(\tau\) is different from \(L\), \(c\) contains secret information and the security context \(\kappa'\) is updated with \(I((p))\), 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 \( q \) that is the set of labels of the secret conditions in the statements that it annotates.

The typing rule for \( \text{for} \) checks that \( \Gamma' \) is an invariant typing environment for the loop, by checking the body \( p \) can be type checked in the slightly more constraining typing environment \( \Gamma'[i \mapsto k] \). In this environment, the iteration variable \( i \) gets the type of the security context \( k \). The rule also ensures that \( \Gamma' \) does not contain any dependency to conditions within \( s \) by ensuring that \( \uparrow_{\text{cond}}(s) \Gamma' = \Gamma' \).

This equality means that there is no variables of type \( I(l) \) in \( \Gamma \), 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 \( (k = L) \) and a program \( P \) with empty annotations \( (\Psi^0) \), 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 \( i.e., \),

\[
\Delta, L \vdash \Gamma_1 \{P^0\} \Gamma_2
\]

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

\[
\{ x \mid \Gamma_1(x) \neq H \} \cup \{ t \mid \Lambda(t) \neq H \} \subseteq L.
\]

**Proof outline.** Given a type-derivation \( \Delta, L \vdash \Gamma_1 \{P^0\} \Gamma_2 \), we can exhibit a type derivation \( \Delta \vdash ct \{\Gamma_1\} \{P\} \{\Gamma_2\} \) where \( \downarrow \Gamma_1 \) is obtained by mapping all the indirect flow types \( i.e., I(l) \) for some \( l \), to \( L \). By Theorem 1, we conclude the proof. \( \square \)

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 \( H \) 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 \( H \) variable, say \( h \), and confine the array accesses which are indirectly leaking the secret \( h \).

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

**Example 3.** Consider the following snippet

\[
\begin{array}{l}
\text{if}\, @p^0 \, h_0 & \text{then} \begin{array}{l}
\text{if}\, @p^0 \, h_1 & \text{then} x = 0 \quad \text{else} \quad x = 1
\end{array} \end{array}
\]

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

\[
\begin{array}{l}
\text{if}\, @p^0 \, h_0 & \text{then} \begin{array}{l}
\text{if}\, @p^0 \, h_1 & \text{then} x = 0 \quad \text{else} \quad x = 1 \quad \text{next} \, s
\end{array} \end{array}
\]

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 \( H \) conditional is selected, we get

\[
\begin{array}{l}
\text{if}\, @p^0 \, h_0 \quad \text{then} \begin{array}{l}
\text{if}\, @p^0 \, h_1 \quad \text{then} x = 0 \quad \text{else} \quad x = 1 \quad \text{next} \, s
\end{array} \end{array}
\]

In that case, \( s \) is still typed under the \( L \) 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 \( H \) 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 \( H \) conditional. Initially, the statement \( s \) is skip. If the program is the \( H \) 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 \( L \) and there are two symmetric cases depending on whether the \( H \) 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 \( s \). However, if the continuation is skip, we can optimise and split \( s_2 \) into a pair of statements \( (s_1, s_2) \in sep_p(s_2) \) such that \( s_1 \) does not leak \( p \). Therefore, \( SI_p \) is recursively called over \( s_1 \) with a reduced continuation \( s_2 \) 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 \( H \) 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 \( \Delta, k \vdash p^@ \Gamma(s)\Gamma' \). It is
Theorem 3 (Security of Scope Increase). Let $p$ be the rightmost (high) condition of $c$ i.e., $R_O(c)$ and $s$ be a program without any high condition i.e., $\text{high}(s) = 0$. Suppose that $c, s$ is well-typed i.e.,

$$\Delta, \Gamma \vdash c : \tau, r_e \quad \kappa' = \kappa \cup \{r \mapsto \text{I}(p')\}$$

$$\Delta, \kappa' \cup t@P \Delta \vdash (s_1 \vec{t}; I((p'))) \Gamma_3$$

$$\Delta, \kappa, \Gamma \vdash \{s_2\}_{I(p')} \Gamma'$$

$$r = r_1 \cup r_2 \cup r_3 \cup r_e \quad g = g_1 \cup g_2 \cup g_3 \cup r \mapsto \emptyset \{p'\}$$

$$\Gamma'' = \begin{cases} \Gamma\{p\} \Gamma' & \text{if } r \neq \emptyset \land p = p' \\ \Gamma & \text{otherwise} \end{cases}$$

$$\Delta, \kappa \cup t@P \Gamma \vdash \{\text{if}\oplus p \ c \ then \ s_1 \ else \ s_2 \ next \ s\} \Gamma''$$

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 $L$. In that case, instead of $\Gamma'$, the final typing environment is $\Gamma'' = \{p\} \Gamma'$ 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 $H$ conditional.

Theorem 3 (Security of Scope Increase). Let $p$ be the rightmost (high) condition of $c$ i.e., $R_O(c)$ and $s$ be a program without any high condition i.e., $\text{high}(s) = 0$. Suppose that $c, s$ is well-typed i.e.,

$$\Delta, \Gamma \vdash c : \tau, r_e \quad \kappa' = \kappa \cup \{r \mapsto \text{I}(p')\}$$

$$\Delta, \kappa' \cup t@P \Delta \vdash (s_1 \vec{t}; I((p'))) \Gamma_3$$

$$\Delta, \kappa, \Gamma \vdash \{s_2\}_{I(p')} \Gamma'$$

$$r = r_1 \cup r_2 \cup r_3 \cup r_e \quad g = g_1 \cup g_2 \cup g_3 \cup r \mapsto \emptyset \{p'\}$$

$$\Gamma'' = \begin{cases} \Gamma\{p\} \Gamma' & \text{if } r \neq \emptyset \land p = p' \\ \Gamma & \text{otherwise} \end{cases}$$

$$\Delta, \kappa \cup t@P \Gamma \vdash \{\text{if}\oplus p \ c \ then \ s_1 \ else \ s_2 \ next \ s\} \Gamma''$$

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 $H$ conditional identified by the $S_I$ algorithm. Each array access is recursively transformed using the following rule

$$\text{ARR-SAN} \quad t\{i\} \leadsto t\{0 \leq i < \text{size}(t)\}$$

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 $0 \leq i < \text{size}(t)$ when 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 $P$ which either performed an array read or array update depending on a $H$ conditional.

$$P_3 : \text{if}\oplus h \ then \ (x = l_1; y = t\{x\}) \ else \ (x = l_2; t\{x\} = y)$$

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

$$\text{if}\oplus h \ then \ (x = l_1; y = t\{0 \leq x < \text{size}(t)\}) \ else \ (x = l_2; t\{0 \leq x < \text{size}(t)\} = y)$$

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

$$P[\text{if}\oplus p \ c \ then \ s'_1 \ else \ s'_2]$$

where $s'_1$ (resp. $s'_2$) is obtained by applying the rules $\text{ARR-SAN}$ to all the array accesses of $s_1$ (resp. $s_2$).

Theorem 4 (Security of Index Instrumentation). Let $p$ a program point, and $P$ a program typable for our strengthened type system i.e.,

$$\Delta, \kappa \vdash \{s\}_{I(P)} \Gamma'$$

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

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

$$\Delta, \kappa \vdash \{s\}_{I(P)} \Gamma'$$
5.4 Delayed if-conversion

The delayed if-conversion algorithm takes a H condition of the form \( \text{if}^AP \) \( t \) \( \text{then} \) \( s_t \) \( \text{else} \) \( s_e \) 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

\[
\text{pre}; s_t'; s_e'; \text{post}.
\]

The statement \( \text{pre} \) 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 \( \{ x_t = x; x_e = x \} \subseteq \text{pre} \) 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 \( Nxt^{P_1,P_2}_h \) generates code that copies variables, according to two renamings \( P_1 \) and \( P_2 \). In general, \( P_1 \) and \( P_2 \) map the program variables to fresh copies in the \( \text{then} \) and the \( \text{else} \) branch, respectively. The statement \( s_t' \) is obtained by applying recursively the renaming \( P_1 \) to \( s_t \). Similarly, \( s_e' \) is obtained by applying recursively the renaming \( P_2 \) to \( s_e \). The only twist is for array variables that are not copied. For those, we perform an conditional update predicated by the condition \( h \).

\[
Rn_h^n(t[e_1] = e_2) = \{ \rho(e_1) \} = \text{h}!\rho(e_2) : \{ \rho(e_1) \}
\]

The transformation of \( s \) is more complicated. Intuitively, we keep renaming \( s \) with both the renaming inherited from the \( \text{then} \) and \( \text{else} \) 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 \( v \)-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

\[
\{ \rho(e_1) \} = \rho(e_2)
\]

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 \( \text{then} \) and the \( \text{else} \) 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 \( \text{then} \) and \( \text{else} \) branch is copied to fresh variables \( x_t \) and \( x_e \), to be used in the \( \text{then} \) and the \( \text{else} \) branch, respectively. To this end, we define the function \( \text{copy} : \mathcal{P}(\text{Var}) \rightarrow \text{Var} \rightarrow \text{Var} \). Given a set \( V \), \( \text{copy}(V) \) returns a renaming \( \rho \) such that every variable \( x \in V \) is mapped to a variable \( x' \in \text{fresh} \).
that each variable of the loop body is given a fresh variable. The loop body \( s \) is renamed using the obtained renaming maps \( \rho'_1 \) and \( \rho'_2 \). In order to synthesise the renaming maps with the program variables, we insert \( \phi \) 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 \( \text{post}_L \) and \( \text{post}_H \). For each variable \( x \) that is given the same renaming after the next statement i.e., \( \rho_1(x) = \rho_2(x) = x' \), there is an assignment \( x = x' \) \( \in \text{post}_L \). If the renaming differs, i.e., \( \rho_1(x) = x_1 \) and \( \rho_2(x) = x_e \) for \( x_1 \neq x_e \), there is conditional assignment \( x = h?x_1:x_e \) \( \in \text{post}_H \). As a result, the transformation is local to the \( H \) condition of interest and the implicit flow is transferred in a direct flow outside the scope of the condition.

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

\[ \text{IConv}_P(P) \triangleq P'[\text{IConv}(\text{if} @P \text{ c then } s_1 \text{ else } s_2 \text{ next } s_3)] \]

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 \( C = \text{if} h \text{ then } s_1 \text{ else } s_2 \text{ next } s_3 \) be a conditional, and \( P = P'[C] \) a well-typed program containing \( C \) i.e.,

\[ \Delta, L \vdash @P \Gamma(P) \Gamma' \]

for some \( \Delta, \Gamma \) and \( \Gamma' \). The transformed program \( \text{IConv}_P(P) \) is still well-typed and there exists \( \Gamma'' \) such that

\[ \Delta, L \vdash \Gamma(\text{IConv}_P(P)) \Gamma'' \]

Moreover, high(\( \text{IConv}_P(P) \)) \( \subseteq \) high(\( P \) \{p\})
5.5 Constant-Time Transformation

The constant-time transformation consists in iterating the previous transformations \(IConv, IS_p, IS_p\), 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 \(P\).

\[
T(P) = \begin{cases} 
IConv(IS_p(IS_p(P))) & \text{if } RO_p(P) \text{ for some } p \\
\text{otherwise} & 
\end{cases}
\]

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

\[
CTT(P) = \begin{cases} 
P & \text{if } T(P) = P \\
CTT(T(P)) & \text{otherwise} 
\end{cases}
\]

To summarise, we consider a program \(P\) that is well-typed according to our information flow tracking type-system \(\Delta, \Gamma \vdash \Gamma(P)\). 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 \(\Gamma, \Gamma'\) and \(\Delta\), i.e.,

\[
\Delta, \Gamma \vdash \Gamma(P)\]

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

\[
\Delta, \Gamma \vdash \Gamma(CTT(P))\]

Proof outline. The proof is by induction over the number of \(H\) conditionals of \(P\). Suppose a well-typed program \(\Delta, \Gamma \vdash \Gamma(P)\).

- \(high(P) \neq \emptyset\). There is a \(H\) conditional \(p\) such that \(RO_p(P)\).
  After the scope increase pass, by Theorem 3, we obtain
  \[
  \Delta, \Gamma \vdash \Gamma(IS_p(IS_p(P, \text{skip})))\]
  After the index sanitise pass, by Theorem 4, we obtain
  \[
  \Delta, \Gamma \vdash \Gamma(IS_p(IS_p(P, \text{skip})))\]
  After delayed \(if\)-conversion, by Theorem 5, we get a well-typed program \(P' = IConv_p(IS_p(IS_p(P, \text{skip})))\) 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 \(I(l)\) 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: \(\text{if}^{\@ p} h \text{ then } x = l_1 \text{ else } x = l_2\)
- PotentialOOB: \(\text{if}^{\@ h} \text{ then } t[x] = 0 \text{ else skip}\)
- ReturnDeferall: \(\text{if}^{\@ h} \text{ then return } x \text{ else skip}\)

We also include the motivating examples of the current paper i.e., \(P_0\) of Example 2 and \(P_2\) from Section 3, together with hand-crafted programs (see Figure 10) as well as bubble sort. These programs are detailed in table 9. Programs \(P_3, P_4, P_5\) and \(P_7\) need the scope-increase algorithm, while \(P_6\) 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/taoalves/libjc
Type-directed Program Transformation for Constant-Time Enforcement

Type-directed Program Transformation for Constant-Time Enforcement

Table 1: Case study of our transformation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cswap</td>
<td>Swap function from Curve25519</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>Standard bubble sort algorithm</td>
</tr>
<tr>
<td>cswap</td>
<td>Conditional access</td>
</tr>
<tr>
<td>P3</td>
<td>Two imbricated conditionals</td>
</tr>
<tr>
<td>P6</td>
<td>For loop followed by a memory access</td>
</tr>
<tr>
<td>P7</td>
<td>For containing a memory access</td>
</tr>
</tbody>
</table>

Figure 8: Case study of our transformation.

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 ReturnDeferral 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.

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 conditional: 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_i 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.

**Assembly 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 cmov 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 inbricated 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 analyze 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 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.