Control-flow Flattening Preserves the Constant-Time Policy*

Matteo Busi[1], Pierpaolo Degano[1], and Letterio Galletta[2]

[1] Università di Pisa, Pisa, Italy — {matteo.busi, degano}@di.unipi.it
[2] IMT School for Advanced Studies, Lucca, Italy — letterio.galletta@imtlucca.it

Abstract

Obfuscating compilers protect a software by obscuring its meaning and impeding the reconstruction of its original source code. The typical concern when defining such compilers is their robustness against reverse engineering and the performance of the produced code. Little work has been done in studying whether the security properties of a program are preserved under obfuscation. In this paper we start addressing this problem: we consider control-flow flattening, a popular obfuscation technique used in industrial compilers, and a specific security policy, namely constant-time. We prove that this obfuscation preserves the policy, i.e., that every program satisfying the policy still does after the transformation.

1 Introduction

Secure compilation is an emerging research field that puts together techniques from security, programming languages, formal verification, and hardware architectures to devise compilation chains that protect various aspects of software and eliminate security vulnerabilities [15, 10]. As any other compiler, a secure one not only translates a *source program*, written in a *high-level language*, into an efficient *object* code (low-level), but also provides mitigations that make exploiting security vulnerabilities more difficult and that limit the damage of an attack. Moreover, a secure compilation chain deploys mechanisms to enforce secure interoperability between code written in safe and unsafe languages, and makes it hard extracting confidential data from information gained by examining program runs (e.g., soft information as specific outputs given certain inputs, or physical one as power consumption or execution time).

An important requirement for making a compiler secure is that it must grant that the security properties at the source level are fully preserved into the object level or, equivalently, that all the attacks that can be carried out at the object level can also be carried out at the source level. In this way, it is enough showing that the program is secure at the source level, where reasoning is far more comfortable than at low level!

In this paper we focus on obfuscating compilers designed to protect a software by obscuring its meaning and impeding the reconstruction of its original source code. Usually, the main concern when defining such compilers is their robustness against reverse engineering and the performance of the produced code. Very few papers in the literature address the problem of proving their correctness, e.g., [8], and, to the best our knowledge, there is no paper about the preservation of security policies. Here, we offer a first contribution in this direction: we consider a popular program obfuscation (namely *control-flow flattening* [18]) and a specific

*The first two authors have been partially supported by project PRA_2018_66 *DECLware: Declarative methodologies for designing and deploying applications* of the Università di Pisa; the third author by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems).

security policy (namely *constant-time*), and we prove that every program satisfying the policy still does after the transformation, i.e., the obfuscation preservers the policy.

For the sake of presentation, our source language is rather essential, as well as our illustrative examples. The proof that control-flow flattening is indeed secure follows the approach of [3] (briefly presented in Section 2), and only needs paper-and-pencil on our neat, foundational setting. Intuitively, we prove that if two executions of a program on different secret values are indistinguishable (i.e., they take the same time), then also the executions of its obfuscated version are indistinguishable (Section 3).

Actually, we claim that extending our results to a richer language will only require to handle more details with no relevant changes in the structure of the proof itself; similarly, other security properties can be accommodated with no particular effort in this framework, besides those already studied in [3], and also other program transformations can be proved to preserve security in the same manner.

Below, we present the security policy and the transformation of interest.

Constant-time policy An intruder can extract confidential data by observing the physical behavior of a system, through the so-called side-channel attacks. The idea is that the attacker can recover some pieces of confidential information or can get indications on which parts are worth her cracking efforts, by measuring some physical quantity about the execution, e.g., power consumption and time. Many of these attacks, called *timing-based* attacks, exploit the execution time of programs [17]. For example, if the program branches on a secret, the attacker may restrict the set of values it may assume, whenever the two branches have different execution times and the attacker can measure and compare them. A toy example follows (in a sugared syntax), where a user digits her pin then checked against the stored one character by character: here the policy is violated since checking a correct pin takes longer than a wrong one.

```
pin := read_secret();
current_char := 1;
while (current_char  ≤  stored_pin_length and pin(current_char) = stored_pin(current_char))
   current_char := current_char+1;

if (current_char = stored_pin_length+1) then print("OK!");
   else print("KO!");
```

Many mitigations of timing-based attacks have been proposed, both hardware and software. The *program counter* [19] and the *constant-time* [5] policies are software-based countermeasures, giving rise to the *constant-time programming* discipline. It makes programs *constant-time* w.r.t. secrets, i.e., the running times of programs is independent of secrets. The requirement to achieve is that neither the control-flow of programs nor the sequence of memory accesses depend on secrets, e.g., the value of `pin` in our example. Usually, this is formalized as a form of an information flow policy [13] w.r.t. an instrumented semantics that records information leakage. Intuitively, this policy requires that two executions started in equivalent states (from an attacker's point of view) yield equivalent leakage, making them *indistinguishable* to an attacker.

The following is a constant-time version of the above program that checks if a pin is correct:

```
pin := read_secret();
current_char := 1;
pin_ok := true
while (current_char  ≤  stored_pin_length)
   current_char := current_char+1
   if (pin(current_char) = stored_pin(current_char)) then pin_ok := pin_ok
       else pin_ok := false

```

```
if (pin_ok = true) then print("OK!");
   else print("KO!");
```

Control-flow flattening A different securing technique is code obfuscation, a program transformation that aims at hiding the intention and the logic of programs by obscuring (portions of) source or object code. It is used to protect a software making it more difficult to reverse engineer the (source/binary) code of the program, to which the attacker can access. In the literature different obfuscations have been proposed. They range from only performing simple syntactic transformations, e.g., renaming variables and functions, to more sophisticated ones that alter both the data, e.g., constant encoding and array splitting [12], and the control flow of the program, e.g., using opaque predicates [12] and inserting dead code.

Control-flow flattening is an advanced obfuscation technique, implemented in state-of-the-art and industrial compilers, e.g., [16]. Intuitively, this transformation re-organizes the Control Flog Graph (CFG) of a program by taking its basic blocks and putting them as cases of a selective structure that dispatches to the right case. In practice, CFG flattening breaks each sequences of statements, nesting of loops and if-statements into single statements, and then hides them in the cases of a large `switch` statement, in turn wrapped inside a `while` loop. In this way, statements originally at different nesting level are now put next each other. Finally, to ensure that the control flow of the program during the execution is the same as before, a new variable `pc` is introduced that acts as a program counter, and is also used to terminate the `while` loop. The `switch` statement dispatches the execution to one of its cases depending on the value of `pc`. When the execution of a case of the `switch` statement is about to complete `pc` is updated with the value of the next statement to executed.

The obfuscated version of our constant-time example follows.

```
pc := 1;
while(1 ≤ pc)
    switch(pc):
        case 1: pin := read_secret(); pc:= 2;
        case 2: current_char := 1; pc:= 3;
        case 3: pin_ok := true; pc:= 4;
        case 4: if (current_char  ≤  stored_pin_length) then pc:= 5; else pc := 9;
        case 5: current_char := current_char+1; pc:= 6;
        case 6: if (pin(current_char) = stored_pin(current_char)) then pc:= 7; else pc := 8;
        case 7: pin_ok := pin_ok; pc:= 4;
        case 8: pin_ok := false; pc:= 4;
        case 9: skip; pc:= 10;
        case 10: if (pin_ok = true) then pc:= 11; else pc := 12;
        case 11: print("OK!"); pc:= 0;
        case 12: print("KO!"); pc:= 0;
```

Now the point is whether the new obfuscated program is still constant-time, which is the case. In general we would like to have guarantees that the attacks prevented by the constant-time based countermeasure are not possible in the obfuscated versions.

2 Background: CT-simulations

Typically, for proving the correctness of a compiler one introduces a simulation relation between the computations at the source and at the target level: if such a relation exists, we have the guarantee that the source program and the target program have the same observable behavior, i.e., the same set of traces.

A general method for proving that constant-time is also preserved by compilation generalizes this approach and is based on the notion of CT-simulation [3]. It considers three relations: a simulation relation between source and target, and two equivalences, one between source and the other between target computations. The idea is to prove that, given two computations at source level that are equivalent, they are simulated by two equivalent computations at the target level. Actually, CT-simulations guarantee the preservation of a particular form of non-interference, called *observational non-interference*. In the rest of this section, we briefly survey observational non-interference and how CT-simulations preserve it.

The idea is to model the behavior of programs using a labeled transition system of the form $\mathsf{A} \xrightarrow{t} \mathsf{B}$ where A and B are program configurations and t represents the leakage associated with the execution step between A and B. The semantics is assumed deterministic. Hereafter, let the configurations of the source programs be ranged over by $\mathsf{A}, \mathsf{B}, \ldots$ and those of the target programs be ranged over by $\alpha, \beta, \ldots$. We will use the dot notation to refer to commands and state inside configurations, e.g., $\mathsf{A}.\mathit{cmd}$ refers to the command part of the configuration A.[1]

The leakage represents what the attacker learns by the program execution. Formally, the leakage is a list of atomic leakages where not cancellable. Observational non-interference is defined for complete executions (we denote S_f the set of final configurations) and w.r.t. an equivalence relation ϕ on configurations (e.g., states are equivalent on public variables):

Definition 2.1 (Observational non-interference [3]). *A program p is observationally non-interferent w.r.t. a relation ϕ, written $p \models ONI(\phi)$, iff for all initial configurations $A, A' \in S_i$ and configurations B, B' and leakages t, t' and $n \in \mathbb{N}$,*

$$A \xrightarrow{t}{}^{n} B \wedge A' \xrightarrow{t'}{}^{n} B' \wedge \phi(A, A') \implies t = t' \wedge (B \in S_f \text{ iff } B' \in S_f).$$

Hereafter, we denote a compiler/transformation with $[\![\cdot]\!]$ and with $[\![\mathsf{p}]\!]$ the result of compiling a program p. Intuitively, a compiler $[\![\cdot]\!]$ preserves observational non-interference when for every program p that enjoys the property, $[\![\mathsf{p}]\!]$ does as well. Formally,

Definition 2.2 (Secure compiler). *A transformation $[\![\cdot]\!]$ preserves observational non-interference* iff*, for all programs p*

$$\mathsf{p} \models ONI(\phi) \Rightarrow [\![\mathsf{p}]\!] \models ONI(\phi).$$

To show that a compiler $[\![\cdot]\!]$ is secure, we follow [3], and build a *general CT-simulation* in two steps. First we define a simulation, called *general simulation*, that relates computations between source and target languages. The idea is to consider related a source and a target configuration whenever, after they perform a certain number of steps, they end up in two still related configurations. Formally,

Definition 2.3 (General simulation [3]). *Let num-steps$(\cdot, \cdot)$ be a function mapping source and target configurations to $\mathbb{N}$. Also, let $|\cdot|$ be a function from source configurations to $\mathbb{N}$. The relation $\approx_{\mathsf{p}}$ is a* general simulation *w.r.t. num-steps$(\cdot, \cdot)$ whenever:*

1. $(\forall \mathsf{B}, \alpha.\ \mathsf{A} \to \mathsf{B} \wedge \mathsf{A} \approx_{\mathsf{p}} \alpha \implies (\exists \beta.\ \alpha \longrightarrow^{\mathit{num\text{-}steps}(\mathsf{A}, \alpha)} \beta \implies \mathsf{B} \approx_{\mathsf{p}} \beta)$,
2. $(\forall \mathsf{B}, \alpha.\ \mathsf{A} \to \mathsf{B} \wedge \mathsf{A} \approx_{\mathsf{p}} \alpha \wedge \mathit{num\text{-}steps}(\mathsf{A}, \alpha) = 0 \implies |\mathsf{B}| < |\mathsf{A}|$,
3. *For any source configuration $\mathsf{B} \in S_f$ and target configuration α there exists a target configuration $\beta \in S_f$ such that $\alpha \longrightarrow^{\mathit{num\text{-}steps}(\mathsf{A}, \alpha)} \beta \implies \mathsf{A} \approx_{\mathsf{p}} \beta$.*

[1]Following the convention of secure compilation, we write in a blue, sans-serif font the elements of the source language, in a **red, bold** one those of the target and in black those that are in common.

Given two configurations A and α in the simulation relation, the function num-steps(A, α) predicts how many steps α has to perform for reaching a target configuration β related with the corresponding source configuration B. When num-steps$(\mathsf{a}, \alpha) = 0$, a possibly infinite sequence of source steps is simulated by an empty one at the target level. To avoid these situations the measure function $|\cdot|$ is introduced and the condition 2 of the above definition ensures that the measure of source configuration strictly decreases whenever the corresponding target one stutters.

The second step consists of introducing two equivalence relations between configurations: $\overset{\mathsf{c}}{\equiv}_s$ relates configurations at the source and $\overset{\mathsf{c}}{\equiv}_t$ at the target. These two relations and the simulation relation form a *general CT-simulation.* Formally,

Definition 2.4 (General CT-simulation [3])**.** *A pair* $(\overset{\mathsf{c}}{\equiv}_s, \overset{\mathsf{c}}{\equiv}_t)$ *is a* general CT-simulation *w.r.t.* $\approx_{\mathsf{p}}$ *, num-steps*$(\cdot, \cdot)$ *and* $|\cdot|$ *whenever:*

1. $(\overset{\mathsf{c}}{\equiv}_s, \overset{\mathsf{c}}{\equiv}_t)$ *is a* manysteps CT-diagram*, i.e., if*
   - $\mathsf{A} \overset{\mathsf{c}}{\equiv}_s \mathsf{A}'$ *and* $\alpha \overset{\mathsf{c}}{\equiv}_t \alpha'$;
   - $\mathsf{A} \xrightarrow{t} \mathsf{B}$ *and* $\mathsf{A}' \xrightarrow{t} \mathsf{B}'$;
   - $\alpha \xrightarrow{\tau}{}^{\mathit{num\text{-}steps}(\mathsf{A},\alpha)} \beta$ *and* $\alpha' \xrightarrow{\tau'}{}^{\mathit{num\text{-}steps}(\mathsf{A}',\alpha')} \beta'$;
   - $\mathsf{A} \approx_{\mathsf{p}} \alpha$, $\mathsf{A}' \approx_{\mathsf{p}} \alpha'$, $\mathsf{B} \approx_{\mathsf{p}} \beta$ *and* $\mathsf{B}' \approx_{\mathsf{p}} \beta'$

   then
   - $\tau = \tau'$ *and num-steps*$(\mathsf{A}, \alpha) =$ *num-steps*(A', α');
   - $\mathsf{B} \overset{\mathsf{c}}{\equiv}_s \mathsf{B}'$ *and* $\beta \overset{\mathsf{c}}{\equiv}_t \beta'$;
2. *if* A, A' *are initial configurations, with targets* α, α'*, and* $\phi(\mathsf{A}, \mathsf{A}')$*, then* $\mathsf{A} \overset{\mathsf{c}}{\equiv}_s \mathsf{A}'$ *and* $\alpha \overset{\mathsf{c}}{\equiv}_t \alpha'$;
3. *If* $\mathsf{A} \overset{\mathsf{c}}{\equiv}_s \mathsf{A}'$*, then* $\mathsf{A} \in S_f \iff \mathsf{A}' \in S_f$;
4. $(\overset{\mathsf{c}}{\equiv}_s, \overset{\mathsf{c}}{\equiv}_t)$ *is a* final CT-diagram *[3], i.e., if*
   - $\mathsf{A} \overset{\mathsf{c}}{\equiv}_s \mathsf{A}'$ *and* $\alpha \overset{\mathsf{c}}{\equiv}_t \alpha'$;
   - A *and* A' *are final;*
   - $\alpha \xrightarrow{\tau}{}^{\mathit{num\text{-}steps}(\mathsf{A},\alpha)} \beta$ *and* $\alpha' \xrightarrow{\tau'}{}^{\mathit{num\text{-}steps}(\mathsf{A}',\alpha')} \beta'$;
   - $\mathsf{A} \approx_{\mathsf{p}} \alpha$, $\mathsf{A}' \approx_{\mathsf{p}} \alpha'$, $\mathsf{B} \approx_{\mathsf{p}} \beta$ *and* $\mathsf{B}' \approx_{\mathsf{p}} \beta'$

   then
   - $\tau = \tau'$ *and num-steps*$(\mathsf{A}, \alpha) =$ *num-steps*(A', α');
   - $\beta \overset{\mathsf{c}}{\equiv}_t \beta'$ *and they are both final.*

The idea is that the relations $\overset{\mathsf{c}}{\equiv}_s$ and $\overset{\mathsf{c}}{\equiv}_t$ are stable under reduction, i.e., preservation of the observational non-interference is guaranteed. The following theorem, referred to in [3] as Theorem 6, gives a sufficient condition to establish constant-time preservation.

Theorem 2.1 (Security)**.** *If* p *is constant-time w.r.t.* ϕ *and there is a general CT-simulation w.r.t. a general simulation, then* $[\![\mathsf{p}]\!]$ *is constant-time w.r.t.* ϕ*.*

3 Proof of preservation

In this section, we present the proof that control-flow flattening preserves constant-time policy. We first introduce a small imperative language, its semantics in the form of a LTS and our leakage model. Then, we formalize our obfuscation as a function from syntax to syntax, and finally we prove the preservation of the security policy.

3.1 The language and its (instrumented) semantics

We consider a small imperative language with arithmetic and boolean expressions. Let *Var* be a set program identifiers, the syntax is

$$\begin{aligned} AExpr \ni e &::= v \mid \mathtt{x} \mid e_1 \mathtt{op}\ e_2 \qquad v \in \mathbb{Z}, \quad \mathtt{op}\ \in \{\mathtt{+}\,,\mathtt{-}\,,\mathtt{*}\,,\mathtt{/}\,,\mathtt{\%}\,\}, \quad \mathtt{x} \in Var \\ BExpr \ni b &::= \mathtt{true} \mid \mathtt{false} \mid b_1 \mathtt{or}\ b_2 \mid \mathtt{not}\ b \mid e_1 \leq e_2 \mid e_1 = e_2 \\ Cmd \ni c &::= \mathtt{skip} \mid \mathtt{x} := e \mid c_1 \mathtt{;}\ c_2 \mid \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ b\ \mathtt{do}\ c \end{aligned}$$

We assume that each command in the syntax carries a permanent color either *white* or not, typically F. Also, we stipulate that each `while` statement and all its components get a unique non-white color, and that there is a function *color* yielding the color of a statement.

Now, we define the semantics and instantiate the framework of [3] to the *non-cancelling constant-time policy.* For that, we define a *leakage model* to describe the information that an attacker can observe during the execution. Recall from the previous section that the leakage is a list of atomic leaks. We denote with $\cdot$ the list concatenation and with $[a]$ a list with a single element a. Arithmetic and boolean expressions leak the sequence of operations required to be evaluated; we assume that there is an observable $\underline{\text{op}}$, associated with the arithmetic operation being executed, but not with the logical ones (slightly simplifying [3]). Also we denote with $\bullet$ absence of leaking. Our leakage model is defined by the following function $leak(\cdot,\cdot)$ that given an expression (either arithmetic or boolean) and a state returns the corresponding leakage:

$$\begin{aligned} leak(v,\sigma) = leak(\mathtt{x},\sigma) = leak(\mathtt{true},\sigma) = leak(\mathtt{false},\sigma) &= [\bullet] \\ leak(\mathtt{not}\ b,\sigma) &= leak(b,\sigma) \\ leak(e_1 \mathtt{op}\ e_2,\sigma) &= leak(e_1,\sigma) \cdot leak(e_2,\sigma) \cdot \underline{\text{op}} \\ leak(e_1 \leq e_2,\sigma) = leak(e_1 = e_2,\sigma) = leak(b_1 \mathtt{or}\ b_2,\sigma) &= leak(e_1,\sigma) \cdot leak(e_2,\sigma) \end{aligned}$$

Accesses to constants and identifiers leak nothing; boolean and relational expressions leak the concatenation of the leaks of their sub-expressions; the arithmetic expressions append the observable of the applied operator to the leaks of their sub-expressions.

We omit the semantics of arithmetic and boolean expression $[\cdot]_\sigma$ because fully standard [20]; we only assume that each syntactic arithmetic operator `op` has a corresponding semantic operator *op*.

The semantics of commands is given in term of a transition relation $\xrightarrow{t}$ between configurations where t is the leakage of that transition step. As usual a configuration is a pair c,σ consisting of a command and a state $\sigma \in Store$ assigning values to program identifiers. Given a program p the set of initial configurations is $S_i = \{p,\sigma \mid \sigma \in Store\}$, and that of final configurations is $S_f = \{\mathtt{skip},\sigma \mid \sigma \in Store\}$.

Figure 1 reports the instrumented semantics of the language. Moreover, the semantics is assumed to keep colors, in particular in the rule for an F-colored `while`, all the components of the `if` in the target are also F-colored, avoiding color clashes (see the .pdf for colors).

$$\frac{}{\mathtt{x} := e, \sigma \xrightarrow{leak(e,\sigma)\ \cdot\ [\mathtt{x}]} \mathtt{skip}, \sigma\{\mathtt{x} \mapsto [a]_\sigma\}} \qquad \frac{c_1, \sigma \xrightarrow{t} c_1', \sigma'}{c_1\,;\ c_2, \sigma \xrightarrow{t} c_1'\,;\ c_2, \sigma'} \qquad \frac{}{\mathtt{skip};\ c_2, \sigma \xrightarrow{t} c_2, \sigma'}$$

$$\frac{[b]_\sigma = \mathit{true}}{\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma \xrightarrow{leak(b,\sigma)\ \cdot\ [\mathit{true}]} c_1, \sigma} \qquad \frac{[b]_\sigma = \mathit{false}}{\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma \xrightarrow{leak(b,\sigma)\ \cdot\ [\mathit{false}]} c_2, \sigma}$$

$$\frac{}{\mathtt{while}\ b\ \mathtt{do}\ c, \sigma \xrightarrow{[\bullet]} \mathtt{if}\ b\ \mathtt{then}\ (c;\ \mathtt{while}\ b\ \mathtt{do}\ c)\ \mathtt{else}\ \mathtt{skip}, \sigma}$$

Figure 1: Instrumented operational semantics for commands.

3.2 Control-flow flattening formalization

Recall that the initial program being obfuscated is $\mathtt{p}$. For the sake of presentation, we will adopt the sugared syntax we used in Section 1 and represent a sequence of nested conditionals in the obfuscated program as the command $\mathtt{switch}\ \mathbf{e} : \mathbf{cs}$, where $\mathbf{cs} = [(\mathbf{v_1}, \mathbf{c_1});\ \ldots;\ (\mathbf{v_n} : \mathbf{c_n})]$, with semantics

$$\frac{([\mathtt{e}]_\sigma, \mathtt{c}) \notin \mathtt{cs}}{\mathtt{switch}\ \mathbf{e} : \mathtt{cs}, \sigma \xrightarrow{leak(e,\sigma)} \mathtt{skip}, \sigma} \qquad \frac{([\mathtt{e}]_\sigma, \mathtt{c}) \in \mathtt{cs}}{\mathtt{switch}\ \mathbf{e} : \mathtt{cs}, \sigma \xrightarrow{leak(e,\sigma)} \mathbf{c}, \sigma}$$

Now, let $\mathtt{pc}$ be a fresh identifier, called *program counter*. Then, following [8],the obfuscated version $[\![\mathtt{c}]\!]$ of the command $\mathtt{c}$ is

$$\begin{array}{l} \mathtt{pc} := 1; \\ \mathtt{while}\ 1 \leq \mathtt{pc}\ \mathtt{do} \\ \qquad \mathtt{switch}\ \mathtt{pc} : \mathit{labeled}(\mathtt{pc}, \mathtt{c}, \mathit{1}, \mathit{0}) \end{array}$$

where

$$\begin{array}{l}
\mathit{labeled}(\mathtt{pc}, \mathtt{skip}, n, m) = [(n, \mathtt{skip};\ \mathtt{pc} := m)] \\
\mathit{labeled}(\mathtt{pc}, \mathtt{x} := \mathtt{e}, n, m) = [(n, \mathtt{x} := \mathtt{e};\ \mathtt{pc} := m)] \\
\mathit{labeled}(\mathtt{pc}, \mathtt{c}_1\,;\ \mathtt{c}_2, n, m) = \mathit{labeled}(\mathtt{pc}, \mathtt{c}_1, n, n + \mathit{size}(\mathtt{c}_1)) \cdot \mathit{labeled}(\mathtt{pc}, \mathtt{c}_2, n + \mathit{size}(\mathtt{c}_1), m) \\
\mathit{labeled}(\mathtt{pc}, \mathtt{if\ b\ then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2, n, m) = \\
\qquad [(n, \mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathtt{pc} := n + \mathit{1}\ \mathtt{else}\ \mathtt{pc} := n + \mathit{1} + \mathit{size}(\mathtt{c}_1))] \cdot \\
\qquad \mathit{labeled}(\mathtt{pc}, \mathtt{c}_1, n + 1, m) \cdot \mathit{labeled}(\mathtt{pc}, \mathtt{c}_2, n + 1 + \mathit{size}(\mathtt{c}_1), m) \\
\mathit{labeled}(\mathtt{pc}, \mathtt{while\ b\ do\ c}, n, m) = \\
\qquad [(n, \mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathtt{pc} := n + \mathit{1}\ \mathtt{else}\ \mathtt{pc} := n + \mathit{1} + \mathit{size}(\mathtt{c}))] \cdot \\
\qquad \mathit{labeled}(\mathtt{pc}, \mathtt{c}, n + 1, n) \cdot [(n + 1 + \mathit{size}(\mathtt{c}), \mathtt{skip};\ \mathtt{pc} := m)]
\end{array}$$

with $\mathit{size}(\cdot)$ defined as follows

$$\begin{aligned}
\mathit{size}(\mathtt{c}) &= 1 \text{ if } c \in \{\mathtt{skip}, \cdot := \cdot\} \\
\mathit{size}(\mathtt{c}_1\,;\ \mathtt{c}_2) &= \mathit{size}(\mathtt{c}_1) + \mathit{size}(\mathtt{c}_2) \\
\mathit{size}(\mathtt{if\ b\ then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2) &= 1 + \mathit{size}(\mathtt{c}_1) + \mathit{size}(\mathtt{c}_2) \\
\mathit{size}(\mathtt{while\ b\ do\ c}) &= 2 + \mathit{size}(\mathtt{c})
\end{aligned}$$

The obfuscated version of a program $\mathtt{p}$ is a loop with condition $\mathbf{1} \leq \mathtt{pc}$ and with body a $\mathtt{switch}$ statement. The $\mathtt{switch}$ condition is on the values of $\mathtt{pc}$ and its cases correspond to the flattened statements, obtained from the function $\mathit{labeled}(\texttt{pc}, \texttt{c}, n, m)$. It returns a list containing the cases of the `switch` and it is inductively defined on the syntax of commands: the first parameter `pc` is the identifier to use for program counter; the second is the command `c` to be flattened; the parameter n represents the value of the guard of the case generated for the first statement of `c`; the last parameter m represents the value to be assigned to `pc` by the last `switch` case generated. For example, the flattening of a sequence $\texttt{c}_1; \texttt{c}_2$ generates the cases corresponding to $\texttt{c}_1$ and $\texttt{c}_2$, and then concatenates them. Note that the values of the program counter for the cases of $\texttt{c}_2$ start from the value assigned to `pc` by the last case generated for $\texttt{c}_1$, i.e., $n + \mathit{size}(\texttt{c}_1)$, where the function $\mathit{size}(\cdot)$ returns the "length" of $\texttt{c}_1$. For a program `p`, we use 1 as initial value of n and 0 as last value to be assigned so as to exit from the `while` loop.

3.3 Correctness and security

Since obfuscation does not change the language (apart from sugaring nested `if` commands); the operational semantics is deterministic; and there are no *unsafe* programs (i.e., a program gets stuck iff execution has completed), the correctness of obfuscation directly follows from the existence of a general simulation between the source and the target languages [3]. For that, inspired by [8], we define the relation $\approx_{\texttt{p}}$ between source and target configurations shown in Figure 2. Intuitively, the relation $\approx_{\texttt{p}}$ matches source and target configurations with the same behaviour, depending on whether they are final (third rule), their execution originated from a loop (Rule (Colored)) or not (Rule (White)). Note that we differentiate white and colored cases as to avoid circular reasoning in the derivations of $\approx_{\texttt{p}}$. More specifically, our relation matches a configuration A in the source with a corresponding α in the target. Actually, $\alpha.\mathit{cmd}$ is the `while` loop of the obfuscated program (fourth premise in Rule (White) and third in Rule (Colored)), whereas $\alpha.\sigma$ is equal to $\mathsf{A}.\sigma$ except for the value of `pc`. Its value is mapped to the case of the `switch` corresponding to the next command in A (first premise in Rule (White) and fifth in Rule (Colored)).

To understand how our simulation works, recall the example from Section 1. By Rule (White) we relate the configuration reached at line (3) at the source level with that of the obfuscated program starting at line (2) and with a state equal to that of the source level with the additional binding $\texttt{pc} \mapsto \mathbf{3}$. Similarly, we relate the configuration reached at line (6) at the source level and its obfuscated counterpart (again at line (2) at the obfuscated level), using Rule (Colored) and noting that the source configuration derives from the execution of a loop.

The following theorem ensures that the relation $\approx_{\texttt{p}}$ is a general simulation.

Theorem 3.1. *For all programs* `p`*, the relation* $\approx_{\texttt{p}}$ *is a general simulation.*

The correctness of the obfuscation is now a corollary of Theorem 3.1.

Corollary 3.1 (Correctness). *For all commands* `c` *and store* σ

$$\texttt{c}, \sigma \to^* \texttt{skip}, \sigma' \quad \textit{iff} \quad [\![\texttt{c}]\!], \sigma \to^* \texttt{skip}, \sigma'$$

The next step is showing that the control-flow flattening obfuscation preserves the constant-time programming policy. For that we define $\overset{\texttt{c}}{\equiv}$ below and we show that $(\overset{\texttt{c}}{\equiv}, \overset{\texttt{c}}{\equiv})$ is a general CT-simulation, as required by Theorem 2.1.

Definition 3.1. *Let* $\mathcal{A}$ *and* $\mathcal{A}'$ *be two (source or obfuscated) configurations, then* $\mathcal{A} \overset{\texttt{c}}{\equiv} \mathcal{A}'$ *iff* $\mathcal{A}.\mathit{cmd} = \mathcal{A}'.\mathit{cmd}$.

We prove the following:

(White)

$$\frac{\mathtt{ls} = \mathit{labeled}(\mathtt{pc}, \mathtt{p}, 1, 0) \qquad \begin{array}{c} \mathit{color}(\mathtt{c}) = \mathit{white} \qquad \sigma' = \sigma \cup \{\mathtt{pc} \mapsto n\} \\ \mathtt{c}' = \mathtt{while}\ (\mathbf{1} \leq \mathtt{pc})\ \mathtt{do}\ (\mathtt{switch}\ \mathtt{pc} : \mathtt{ls}) \end{array} \qquad -, \mathtt{pc} \vdash c \bowtie \mathtt{ls}[n], m}{\mathtt{c}, \sigma \approx_{\mathtt{p}} \mathtt{c}', \sigma'}$$

(Colored)

$$\frac{\begin{array}{c} \sigma' = \sigma \cup \{\mathtt{pc} \mapsto n\} \quad \mathtt{ls} = \mathit{labeled}(\mathtt{pc}, \mathtt{p}, 1, 0) \\ \mathtt{c}' = \mathtt{while}\ (\mathbf{1} \leq \mathtt{pc})\ \mathtt{do}\ (\mathtt{switch}\ \mathtt{pc} : \mathtt{ls}) \qquad \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c}'' \in \mathtt{p} \\ \mathit{color}(\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c}'') = \mathit{color}(\mathtt{c}) \neq \mathit{white} \qquad -, \mathtt{pc} \vdash \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c}'' \bowtie \mathtt{ls}[n_0], m' \qquad n_0, \mathtt{pc} \vdash \mathtt{c} \diamond \mathtt{ls}[n], m \end{array}}{\mathtt{c}, \sigma \approx_{\mathtt{p}} \mathtt{c}', \sigma'}$$

$$\frac{\sigma' = \sigma \cup \{\mathtt{pc} \mapsto n\}}{\mathtt{skip}, \sigma \approx_{\mathtt{p}} \mathtt{skip}, \sigma'}$$

$$\frac{}{n_0, \mathtt{pc} \vdash \mathtt{skip} \sim \mathtt{ls}[n], m} \qquad \frac{\mathtt{ls}[n] = (n, \mathtt{x} := \mathtt{e};\ \mathtt{pc} := m)}{n_0, \mathtt{pc} \vdash \mathtt{x} := \mathtt{e} \sim \mathtt{ls}[n], m}$$

$$\frac{n_0, \mathtt{pc} \vdash \mathtt{c}_1 \sim \mathtt{ls}[n], m' \qquad n_0, \mathtt{pc} \vdash \mathtt{c}_2 \sim \mathtt{ls}[m'], m}{n_0, \mathtt{pc} \vdash \mathtt{c}_1;\ \mathtt{c}_2 \sim \mathtt{ls}[n], m}$$

$$\frac{\begin{array}{c} \mathtt{ls}[n] = (n, \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{pc} := n + \mathit{1}\ \mathtt{else}\ \mathtt{pc} := n + \mathit{1} + \mathit{size}(\mathtt{c}_1)) \\ n_0, \mathtt{pc} \vdash \mathtt{c}_1 \sim \mathtt{ls}[n+1], m \qquad n_0, \mathtt{pc} \vdash \mathtt{c}_2 \sim \mathtt{ls}[n + 1 + \mathit{size}(\mathtt{c}_1)], m \end{array}}{n_0, \mathtt{pc} \vdash \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2 \sim \mathtt{ls}[n], m} \qquad \frac{\mathtt{ls}[n] = (n, \mathtt{skip};\ \mathtt{pc} := n_0 c)}{n_0, \mathtt{pc} \vdash \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c} \diamond \mathtt{ls}[n], n_0}$$

$$\frac{}{n_0, \mathtt{pc} \vdash \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c} \diamond \mathtt{ls}[n_0], n_0} \qquad \frac{n, \mathtt{pc} \vdash \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ (\mathtt{c};\ \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c})\ \mathtt{else}\ \mathtt{skip} \diamond \mathtt{ls}[n], m}{-, \mathtt{pc} \vdash \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c} \bowtie \mathtt{ls}[n], m}$$

where $\sim \in \{\diamond, \bowtie\}$, and the first parameter (n_0) is immaterial in $\bowtie$.

Figure 2: Definition of $\approx_{\mathtt{p}}$ relation on configurations and its auxiliary relations.

Theorem 3.2. *The pair* $(\overset{\mathsf{c}}{\equiv}, \overset{\mathsf{c}}{\equiv})$ *is a* general CT-simulation *w.r.t.* $\approx_{\mathtt{p}}$, *num-steps*$(\cdot, \cdot)$ *and* $|\cdot|$.

The main result of our paper directly follows from the theorem above, because the transformation in Section 3.2 satisfies Definition 2.2:

Corollary 3.2 (Constant-time preservation)**.**
The control-flow fattening obfuscation preserves the constant-time policy.

The proofs of the theorems above are available online [9].

4 Conclusions

In this paper we applied a methodology from the literature [3] to the advanced obfuscation technique of control-flow flattening and proved that it preserves the constant-time policy. For that, we have first defined what programs leak. Then, we have defined the relation $\approx_{\mathtt{p}}$ between source and target configurations – that roughly relates configurations with the same behavior – and proved that it adheres to the definition of general simulation. Finally, we proved that the obfuscation preserves constant time by showing that the pair $(\overset{\mathsf{c}}{\equiv}, \overset{\mathsf{c}}{\equiv})$ is a general CT-simulation, as required by the framework we instantiated. As a consequence, the obfuscation based on control-flow flattening is proved to preserve the constant-time policy.

Future work will address proving the security of other obfuscations techniques, and considering other security properties, e.g., general safeties or hyper-safeties. Here we just considered a passive attacker that can only observe the leakage, and an interesting problem would be to explore if our result and the current proof technique scale to a setting with active attackers that also interferes with the execution of programs. Indeed, recently new secure compilation principles have been proposed to take active attackers into account [1].

Related Work Program obfuscations are widespread code transformations [18, 12, 16, 11, 4, 25] designed to protect software in settings where the adversary has physical access to the program and can compromise it by inspection or tampering. A great deal of work has been done on obfuscations that are resistant against reverse engineering making the life of attackers harder. However, we do not discuss these papers because they do not consider formal properties of the proposed transformations. We refer the interested reader to [14] for a recent survey.

Since to the best our knowledge, ours is the first work addressing the problem of security preservation, here we focus only on those proposals that formally studied the correctness of obfuscations. In [23, 24] a formal framework based on abstract interpretation is proposed to study the effectiveness of obfuscating techniques. This framework not only characterizes when a transformation is correct but also measures its resilience, i.e., the difficulty of undoing the obfuscation. More recently, other work went in the direction of fully verified, obfuscating compilation chains [6, 8, 7]. Among these [8] is the most similar to ours, but it only focusses on the correctness of the transformation, and studies it in the setting of the CompCert C compiler. Differently, here we adopted a more foundational approach by considering a core imperative language and proved that the considered transformation preserves security.

As for secure compilation, we can essentially distinguish two different approaches. The first one only considers passive attackers (as we do) that do not interact with the program but that try to extract confidential data by observing its behaviour. Besides [3], recently there has been an increasing interest in preserving the verification of the constant time policy, e.g., a version of the CompCert C compiler [2] has been released that guarantees that preservation of the policy in each compilation step. The second approach in secure compilation considers active attackers that are modeled as contexts in which a program is plugged in. Traditionally, this approach reduces proving the security preservation to proving that the compiler is fully-abstract [21]. However, recently new proof principles emerged, see [1, 22] for an overview.

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *32nd IEEE Computer Security Foundations Symposium*, pages 256–271, 2019.

[2] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *PACMPL*, 4(POPL):7:1–7:30, 2020.

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

[4] Mihai Bazon. Uglifyjs - javascript parser, compressor, minifier written in js. http://lisperator.net/uglifyjs/. Online; last access Dec 2019.

[5] Daniel J. Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005. Online; last access Nov 2019.

[6] Sandrine Blazy and Roberto Giacobazzi. Towards a formally verified obfuscating compiler. In *SSP 2012 - 2nd ACM SIGPLAN Software Security and Protection Workshop*, 2012.

[7] Sandrine Blazy and Rémi Hutin. Formal verification of a program obfuscation based on mixed boolean-arithmetic expressions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 196–208, 2019.

[8] Sandrine Blazy and Alix Trieu. Formal verification of control-flow graph flattening. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 176–187, 2016.

[9] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. Control-flow flattening preserves the constant-time policy (extended version). https://arxiv.org/abs/2003.05836.

[10] Matteo Busi and Letterio Galletta. A brief tour of formally secure compilation. In Pierpaolo Degano and Roberto Zunino, editors, *Proceedings of the Third Italian Conference on Cyber Security, ITASEC19*, volume 2315 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[11] Christian Collberg. The tigress c diversifier/obfuscator. http://tigress.cs.arizona.edu/. Online; last access Dec 2019.

[12] Christian S. Collberg and Jasvir Nagra. *Surreptitious Software - Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2010.

[13] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[14] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, 104:72–93, 2018.

[15] Catalin Hritcu, David Chisnall, Deepak Garg, and Mathias Payer. Secure compilation. https://blog.sigplan.org/2019/07/01/secure-compilation/, 2019. Online; last access Dec 2019.

[16] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9. IEEE, 2015.

[17] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[18] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[19] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference*, volume 3935 of *LNCS*, pages 156–168, 2005.

[20] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007.

[21] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019.

[22] Marco Patrignani and Deepak Garg. Robustly safe compilation or, efficient, provably secure compilation. *CoRR*, abs/1804.00489, 2018.

[23] Mila Dalla Preda and Roberto Giacobazzi. Control code obfuscation by abstract interpretation. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*, pages 301–310, 2005.

[24] Mila Dalla Preda and Roberto Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

[25] WebAssembly team. Binaryen - compiler infrastructure and toolchain library for webassembly. https://github.com/WebAssembly/binaryen. Online; last access Dec 2019.