Secure Compilation of Constant-Resource Programs

Gilles Barthe, Sandrine Blazy, Rémi Hutin, David Pichardie

Secure Compilation of Constant-Resource Programs

Gilles Barthe
MPI Security and Privacy, Germany
IMDEA Software Institute, Spain
gbarthe@mpi-sp.org

Sandrine Blazy
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
sandrine.blazy@irisa.fr

Rémi Hutin
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
remi.hutin@ens-rennes.fr

David Pichardie
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
david.pichardie@ens-rennes.fr

***Abstract*—Observational non-interference (ONI) is a generic information-flow policy for side-channel leakage. Informally, a program is ONI-secure if observing program leakage during execution does not reveal any information about secrets. Formally, ONI is parametrized by a leakage function ℓ, and different instances of ONI can be recovered through different instantiations of ℓ. One popular instance of ONI is the *cryptographic constant-time* (CCT) policy, which is widely used in cryptographic libraries to protect against timing and cache attacks. Informally, a program is CCT-secure if it does not branch on secrets and does not perform secret-dependent memory accesses. Another instance of ONI is the *constant-resource* (CR) policy, a relaxation of the CCT policy which is used in Amazon's s2n implementation of TLS and in several other security applications. Informally, a program is CR-secure if its cost (modelled by a *tick* operator over an arbitrary semi-group) does not depend on secrets.**

In this paper, we consider the problem of preserving ONI by compilation. Prior work on the preservation of the CCT policy develops proof techniques for showing that main compiler optimisations preserve the CCT policy. However, these proof techniques critically rely on the fact that the semi-group used for modelling leakage satisfies the property:

$$\ell_1 + \ell_1' = \ell_2 + \ell_2' \Longrightarrow \ell_1 = \ell_2 \wedge \ell_1' = \ell_2'$$

Unfortunately, this non-cancelling property fails for the CR policy, because its underlying semi-group is $(\mathbb{N}, +)$ and it is currently not known how to extend existing techniques to policies that do not satisfy non-cancellation.

We propose a methodology for proving the preservation of the CR policy during a program transformation. We present an implementation of some elementary compiler passes, and apply the methodology to prove the preservation of these passes. Our results have been mechanically verified using the Coq proof assistant.

***Index Terms*—Secure Compilation, Verified Compilation**

I. Introduction

Over the last two decades, correctness of (moderately) optimising compilers has turned from a distant goal into a tangible reality. Even better, many of these compiler correctness proofs are mechanically verified using proof assistants. Prominent examples of formally verified compilers include CompCert [18], CakeML [17], Vellvm [27] and Jasmin [4]. Formally verified compilers are appealing because they eliminate the possibility of compiler bugs, which can be devastating.

Unfortunately, provably correct compilers may still fail to preserve security properties. This is because many security properties reason about pairs of program executions, whereas compiler correctness reasons about single program execution. For instance, the baseline policy for confidentiality, called non-interference, informally requires that two executions started from low-equivalent states (diverge or) terminate in low-equivalent states, where two states are low-equivalent if they only differ in their secrets. This property ensures that an adversary learns no information from observing the input-output behaviour of programs. It is not difficult to see that under some mild assumptions compiler correctness entails preservation of non-interference. However this good news does not extend to other policies. For instance, it is folklore [22] that compilers do not generally preserve the cryptographic constant-time policy (CCT), a popular policy for protecting against cache and timing side-channel attacks [10]. More generally, it is well-known [15] that many compiler optimisations fall into a correctness/security gap, i.e., break the security of programs while preserving their semantics.

Secure compilation is an emerging area that aims to address the correctness/security gap, by developing compilers that preserve security properties, and formal techniques for preservation proofs. A significant challenge in secure compilation is to formalise the notion of secure compiler [2], [1]. Another challenge, and the focus of this paper, is to prove preservation for a specific compiler and a specific security policy. Prior work along this line [9], [7] has focused on CCT policy. Their main tool for proving preservation of CCT is the notion of 2-simulation, an adaptation of the notion of simulation used for compiler correctness proofs. Whereas simulations consider one source execution and one target execution, 2-simulations consider two source executions and two target executions. The notion of 2-simulation works well for proving preservation of the CCT policy, thanks to a key property: leakage cancellation. Recall that a program is CCT if it does not branch on secrets and does not make secret-dependent memory accesses. The formal definition of CCT security is captured using the notion of CCT leakage: informally, the CCT leakage of a program execution is the list of boolean guards (boolean values) and

```
if(secret) { x+=2; } else { y+=3; }
```

(a) A program with balanced branches

$\boxed{\texttt{if(secret) } \{\, p_1 \,\} \texttt{ else } \{\, p_2 \,\}}\ ;\ p_3$

(b) A program with an atomic annotation

Fig. 1: Examples of CR-secure programs

memory accesses (memory locations) in an execution. Using CCT leakage, it is then easy to define CCT security: a program is CCT if executions from any two low-equivalent states yield the same CCT leakage. Informally, the CCT leakage is non-cancelling because two executions that have differing CCT leakages at a program point cannot have equal CCT leakage at a further program point. More formally, CCT leakage is non-cancelling because its underlying semi-group (lists with concatenation) satisfies the following property:

$$l_1 + l_1' = l_2 + l_2' \Longrightarrow l_1 = l_2 \wedge l_1' = l_2'$$

(under the assumptions that the lists l_1 and l_2 have the same length, and so have the lists l_1' and l_2'). This property does not hold for notions of leakage based on cost, such as the constant-resource (CR) policy defined in [20].

The CR policy defines a notion of resource consumed during the execution of program. A resource can be a counter measuring the number of arithmetic operations, memory accesses or function calls. A more precise resource model can take branch prediction and cache into account to model execution time on a given architecture. The CR policy states that an attacker capable of measuring the resources consumed during the execution of a program cannot deduce any information on the secrets of the program.

The code snippet presented in Figure 1a consists of a branching on a secret value, and two branches performing the same kind of operations (i.e., incrementing a variable by a constant value). In a resource model counting the number of arithmetic operations, this snippet is considered CR-secure, as the resource consumption is constant and does not depend on the secret value of the snippet. Unlike the CCT policy, the CR policy tolerates a branching depending on a secret value, as long as the branches are balanced in terms of costs. Because leakages are not constrained in these branches, the CR policy does not satisfy the non-cancellation property.

In this paper, we study the preservation by compilation of the CR policy and present a proof methodology to prove that a transformation preserves the CR policy. Compiler optimisations may easily break the CR policy. Indeed, as a CR-secure program may contain balanced branches, any optimisation that reduces the resource consumption in one of the branches would directly break the balance.

To solve this issue, a first solution is to use information-flow typing to guide the transformation. Type systems can detect high branches and forbid or restrict optimisation inside them. For example, in [3] a standard type system [26] is used to *repair* a typable program that may contain unbalanced high branches. This an elegant approach but in our work we want to avoid the use of an information-flow type-system inside the compilation chain. Modern compilers perform their optimisations at a low-level program representation and running a taint analysis at this abstraction level is likely to conservatively declare all the contents of the memory as secret-dependent.

Our methodology introduces a new *atomic* special construct that guides the compiler, without asking it to perform a taint analysis before optimisation. The *atomic* constructs can be inserted at source level using a source type system or any program logic that detects high branches. We introduce a stronger policy, called CR$^\#$, which combines elements from the CCT [5] and CR [20] policies (i.e., leakages and costs). CR$^\#$ leakages track costs as well as the CCT boolean leakages of some branchings. The *atomic* construct is used to decide whether or not the CR$^\#$ policy tracks a branching statement.

Like other ONI policies, the CR$^\#$ policy enforces that two executions started in equivalent states yield the same leakage, but only branches inside *atomic* constructs are allowed to depend on secrets. These program regions have to be optimised without breaking cost balance between any execution path inside them. Figure 1b shows an example of an atomic annotation, depicted with a box notation around the if statement. Intuitively, the annotation is used to restrain the compiler on the first part of the program: p_1 and p_2 will only be optimised if they preserve the balance between the branches; p_3 will be optimised without restriction.

Our inspiration for atomic annotations comes from concurrent programs where *memory barriers* (or *fences*) are used as a synchronisation method, and as a compiler annotation to enforce an ordering constraint on memory operations. They save the compiler from performing a difficult alias analysis to detect data races. Intuitively, an atomic annotation protects an area of the program, instructing the compiler to restrain its optimisations in this area. It has no impact on its execution, and we assume that a prior analysis introduced the atomic annotations. They are used to syntactically identify the high-security parts of the program, that are also identified by a previous analysis of the program. The compiler does not need to perform any taint analysis to check the atomic annotations. Informally, secret (i.e., private) branching conditions must appear inside of atomic annotations, while public (i.e., non-secret) branching condition must appear outside of them. Atomic annotations are then used to indicate where to optimise differently the program in order to preserve the CR$^\#$ policy.

Last, some common compiler passes, such as the common sub-expression elimination (abbreviated to CSE) preserve the CCT policy, but are likely not to preserve the CR$^\#$ policy. We explain how to modify them so that they preserve the CR$^\#$ policy. Our approach consists in introducing a minimal amount of padding in the program. We detail the associated proof methodology we designed to tackle the main issue that CR$^\#$ leakages are not non-cancelling. In order to facilitate our proofs, we split our optimisations into elementary transformations, that are individually proved CR$^\#$ preserving. As a

consequence, most of this work could be reused to study any optimisation (e.g., constant propagation, partial redundancy elimination) that does not insert new branchings and is able to maintain the consumed resources of some parts of a program using padding.

All results presented in this paper have been mechanically verified using the Coq proof assistant. The complete development is available as a supplementary material. This paper makes the following contributions.

- We formalise the notion of a security policy called CR$^\#$.
- We introduce an annotation called atomic, allowing to flexibly identify the high-security parts of the program.
- We adapt common compiler optimisations so that they preserve the CR$^\#$ policy; this relies on padding insertion followed by padding minimisation.
- We define elementary passes that are used to build the previous optimisations and we prove that these passes preserve the CR$^\#$ policy.
- We prove that padding minimisation preserves the CR$^\#$ policy.

This paper is organised as follows. First, Section II illustrates through a motivating example how we adapt the CSE optimisation so that it becomes CR-preserving. Then, we motivate our CR$^\#$ policy in Section III, before giving precise definitions in Section IV. Section V explains how to decompose three common compiler optimisations into general elementary transformations that preserve the CR$^\#$ policy, which is proved in Section VI. Moreover, Section VII is devoted to padding minimisation, our trickiest transformation to prove CR$^\#$-preserving. Related work is described in Section VIII, followed by conclusions.

II. Preserving the CR Policy through Compilation

This section first gives examples of CR-secure programs. Then, it explains through the example of CSE how to adapt a transformation to make it CR-preserving. It relies on a policy stronger than CR, that we call the CR$^\#$ policy.

A. Example of CR-secure Programs

Figure 2 presents some code snippets written in `C` syntax. In the first one, if the condition `cond` is secret, then this snippet is considered unsecure by the CCT policy. In general, branchings on secrets are unsecure because an attacker able to measure their execution time could determine which branch was executed, and thus the secret condition. However, because both branches consume the same amount of resources (i.e., six accesses to variables, two additions, two multiplications, and two assignments), we better consider that these branches are indistinguishable from the perspective of such an attacker. This program is then an example of CR-secure program.

The code snippet in Figure 2a presents some redundant computations, and it is a good candidate for the CSE optimisation. Figure 2b shows the optimised code snippet, which is not CR-secure. Indeed, both branches perform five accesses to variables and two assignments, but the `then` branch performs two additions and one multiplication, while the `else` branch performs one addition and two multiplications. Hence, these branches are no longer balanced. This illustrates how a simple transformation can break the CR policy.

B. A CSE Optimisation that Preserves the CR Policy

Preserving the CR policy requires to carefully keep track of all the branches of the program. Any optimisation performed only in one branch could unbalance the whole program. Our approach consists in preserving the balance by introducing a minimal amount of padding in every unbalanced branch. We illustrate it with the example of the CSE optimisation.

We add two steps to the CSE optimisation to make it CR-preserving. First, our new CSE adds padding to balance the consumption of resources between the branches, using a new padding instruction called $\boldsymbol{\delta}$. It is parametrized by an integer n and executing $\boldsymbol{\delta}(n)$ consumes n resources. Second, our CSE performs a pass called $\mathcal{M}$ that minimises the padding by factorising and removing as many $\boldsymbol{\delta}$ instructions as possible, as long as the CR policy is preserved. More precisely, any modification of the resource consumption stemming from CSE is compensated by an adequate $\boldsymbol{\delta}$ instruction.

In the `then` branch of Figure 2c, our CSE factorises a redundant evaluation and modifies the resource consumption: the optimised program has one less multiplication and one less variable access. So, our CSE compensates these changes by adding the instruction $\boldsymbol{\delta}(T_1)$ in the `then` branch, with $T_1 = K^{mult} + K^{var}$, where the K^{mult} and K^{var} constants are statically computed and represent the cost of respectively a multiplication and a variable access. The added $\boldsymbol{\delta}$ instruction consumes the same amount of resources spared by CSE, hence preserving the resource consumption of the whole program. Similarly, the padding instruction $\boldsymbol{\delta}(T_2)$ is added in the `else` branch, with $T_2 = K^{add} + K^{var}$. Figure 2d shows the final code snippet, where the padding of both branches is reduced by $T = \min(T_1, T_2)$. This last step minimises the padding while keeping both branches balanced.

Our modified version of the CSE optimisation introduces padding to preserve the balance of costs between branches. Then, it minimises the inserted padding as much as possible, while preserving the CR policy. However, this behaviour may not always be desired, as it makes the output program less efficient in terms of consumed resources. Indeed, some branches could be secret dependent and balanced, while some other branches could be non secret-dependent. The former branches would need a careful and restrained optimisation, similar to the example above. However, the latter branches could benefit from a more aggressive optimisation, without requiring any padding. To distinguish between both cases, we introduce a syntactic annotation, called *atomic*, which delimit the areas of the program to be carefully optimised. The padding insertion and minimisation passes are then only performed in these areas.

The proof that our modified CSE preserves the CR policy consists of the individual proofs of each of its steps. This proof effort led us to define a stronger security policy, that we call CR$^\#$, and whose definition depends on the atomic

```
if (cond) {
 x = a*b;
 y = (a*b)+c+d;
} else {
 x = a+b;
 y = (a+b)*c*d;
}
```

(a) A balanced program with common subexpressions `a*b` and `a+b`

```
if (cond) {
 x = a*b;
 y = x+c+d;
} else {
 x = a+b;
 y = x*c*d;
}
```

(b) The unbalanced optimised program (with CSE)

```
if (cond) {
 δ(T1);
 x = a*b;
 y = x+c+d;
} else {
 δ(T2);
 x = a+b;
 y = x*c*d;
}
```

(c) The padded optimised program

```
if (cond) {
 δ(T1 − T);
 x = a*b;
 y = x+c+d;
} else {
 δ(T2 − T);
 x = a+b;
 y = x*c*d;
}
```

(d) The padded optimised program with minimal padding

Fig. 2: Example of branching programs, where $T_1 = K^{mult} + K^{var}$, $T_2 = K^{add} + K^{var}$ and $T = \min(T_1, T_2)$.

annotations. This CR$^\#$ policy is discussed in the next section. Some proofs are trickier than others, and we define in this paper (see Section VI and Section VII) two other policies that are stronger than the CR$^\#$ policy but facilitate our proofs. In particular, we decompose $\mathcal{M}$ into two steps, and each of them is proved using a different policy. We also prove that any of these policies implies the CR$^\#$ policy. Interestingly, these policies are not peculiar to CSE and could be reused to prove other optimisations.

III. The CR$^\#$ Policy

This section first motivates the choice of our security policy. Our work is based on the constant-resource (CR) security property presented in [20], which we describe first. We then present a stronger property, called CR$^\#$, and motivate this definition. CR$^\#$ is the security property we will focus on during the rest of the paper.

A. The CR Policy

In [20], the authors present the CR policy. It captures the fact that a given notion of resources consumed during the execution of a program does not reveal any information on the secret values of this program.

More formally, let us first consider a language $\mathcal{L}$, and its big-step semantics judgement $\langle p, \sigma\rangle \Downarrow \sigma', q$ instrumented to observe the resource consumption of an execution. We read it as follows: the execution of a program $p \in \mathcal{L}$ from an initial state σ to a final state σ' consumes q resources, where q is an integer. States map variable identifiers to values, and every variable is marked as either secret or public.

Next, we consider a notion of indistinguishability between semantic states. Two states σ_1 and σ_2 are indistinguishable, written as $\sigma_1 \sim \sigma_2$, if every public variable has the same value in both states.

Then, the CR policy is defined as follows. A program is CR if any pair of executions whose initial states only differ on secret values consume the same amount of resources. This captures the idea that resource consumption does not reveal any information on the secrets of a CR program.

Definition III.1 (CR security)**.** *Let p be a program, and states σ_1 and σ_2 such that $\sigma_1 \sim \sigma_2$. Suppose that we have two executions of p: $\langle p, \sigma_1\rangle \Downarrow \sigma_1', q_1$ and $\langle p, \sigma_2\rangle \Downarrow \sigma_2', q_2$. The program p is CR-secure (written as CR(p)) when $q_1 = q_2$.*

B. Secure Compilation of CR Programs

Our goal is to implement program transformations that preserve the CR policy, then prove that these transformations always preserve the policy. Formally, we say that a transformation $\mathcal{T}$ is CR-preserving if for any program p, we have $\text{CR}(p) \implies \text{CR}(\mathcal{T}(p))$.

A first possibility to prove that a transformation $\mathcal{T}$ is CR is to use a type system, as in [20], where the type system is designed so that any well-typed program p, denoted as $\vdash p$, is CR-preserving: $\vdash p \implies \text{CR}(p)$. We could then prove that $\mathcal{T}$ preserves such a type system. Formally, we would prove the following property: $\vdash p \implies \vdash \mathcal{T}(p)$, stating that the type system enforces the CR policy on both source and transformed programs. This approach would work in the context of a simple type system and a simple language. However, we argue that it would not scale to a more realistic compiler, such as CompCert or LLVM, as type-preserving compilers typically do not scale to realistic languages. As far as we know, no realistic compiler includes a type system to verify the preservation of non-interference properties. Another drawback of this approach is that the compiler must explicitly know the security level (i.e., secret or public) of every variable.

Our methodology to prove that a transformation is CR-preserving does not rely on a type system, but rather on an extended language and its instrumented semantics. Firstly, we extend the language $\mathcal{L}$ with a syntactic annotation, that we call an *atomic* annotation (see Section I). Secondly, we extend the semantics of $\mathcal{L}$, by instrumenting it with leakages that track the branchings encountered during the execution. The information leaked is a partial control-flow of the program, represented by a list of booleans values, that contains some of the guards evaluated during the execution. The choice of leaked control-flow depends on the atomic annotations, as secret branching conditions only appear inside of atomic annotations. Last, we extend the CR policy to facilitate our proofs. We introduce

a stronger policy, called CR$^{\#}$, that is defined with respect to this newly introduced leakage.

We use the CR$^{\#}$ policy in the following way. Firstly, we annotate any program p into $p^{\#}$, so that CR$(p) \implies$ CR$^{\#}(p^{\#})$. This will be discussed in IV. Secondly, we prove that the transformation $\mathcal{T}$ we are focusing on preserves the CR$^{\#}$ policy. Formally, for any program p, CR$^{\#}(p) \implies$ CR$^{\#}(\mathcal{T}(p))$. Last, as CR$^{\#}$ is stronger than CR, we directly have for any program p, CR$^{\#}(p) \implies$ CR(p).

The CR$^{\#}$ policy is a proof artefact. However, we argue that it is a relevant security property by itself. This idea will be discussed in the following section, that also contains precise definitions of our example language, of its extension with atomic annotations, of its instrumented semantics and of the CR$^{\#}$ property, along with several examples.

IV. Formal Semantics and CR$^{\#}$ Policy

We focus on a While language that highlights the salient features of our approach. This section first defines While. We formalise our CR$^{\#}$ policy and its preservation by program transformations. Then, we discuss the relation between the CCT, CR and CR$^{\#}$ policies. Last, useful semantic properties of While are detailed.

A. Instrumented Semantics of the While Language

The syntax of While (see Figure 3a) contains common features, such as expressions over integers and usual arithmetic operations, `skip` statements, assignments, sequences of statements, and branches within `if` and `while` statements. We add two peculiar statements, a padding instruction $\boldsymbol{\delta}$ parametrized by a quantity of consumed resources (i.e., the execution of $\boldsymbol{\delta}(n)$ where n is a constant consumes n resources) and an atomic annotation, denoted using a box notation: $\boxed{p}$ is the atomic version of p.

The semantics of While is defined in Figure 3 using a big-step style, that we instrument with the amount of consumed resources and a boolean leakage emitted by an execution. A boolean leakage is a list of boolean values, representing the values of all the boolean guards encountered during the execution. As such, it fully describes the execution path. This notion of boolean leakage is commonly used to capture the definition of the CCT policy [5]. We use notation $l_1 \cdot l_2$ to denote concatenation of boolean leakages, and empty leakage is denoted by ϵ. Moreover, the semantics is parametrized by a set of constants denoted by $K^{\cdots}$ and representing the unitary costs of basic syntactic constructs. We assume that our notion of resource is additive, as shown in rule OP_BIN of Figure 3b, where the evaluation of the arithmetic operation $\diamond$ consumes $K^{\diamond}$ resources. Rule SEQ of Figure 3c highlights the additivity of resource consumption. In the rules, a state σ maps variable identifiers to values. If a variable identifier is not defined in a state, evaluating the variable returns the default value 0. A *leakage* is a pair (q, l) with q the consumed resources and l the emitted boolean leakage.

Executing `skip` does not consume any resource and emits an empty boolean leakage. Executing a padding

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle int \rangle \mid \langle ident \rangle \mid \langle exp \rangle \ \diamond \ \langle exp \rangle \\
\langle stmt \rangle & ::= & \texttt{skip} \mid \boldsymbol{\delta}\texttt{(} \ \langle int \rangle \ \texttt{)} \\
& \mid & \langle ident \rangle \ \texttt{:=} \ \langle exp \rangle \mid \langle stmt \rangle \ \texttt{;} \ \langle stmt \rangle \\
& \mid & \texttt{if(} \ \langle exp \rangle \ \texttt{)} \ \{ \ \langle stmt \rangle \ \} \ \texttt{else} \ \{ \ \langle stmt \rangle \ \} \\
& \mid & \texttt{while(} \ \langle exp \rangle \ \texttt{)} \ \{ \ \langle stmt \rangle \ \} \mid \boxed{\langle stmt \rangle}
\end{array}
$$

(a) Syntax

$\boxed{\langle e, \sigma \rangle \Downarrow n, q}$ evaluating e in state σ yields value n and consumes q resources

$$\textsc{const}\ \frac{}{\langle n, \sigma \rangle \Downarrow n, K^{int}} \qquad \textsc{var}\ \frac{\sigma[id] = n}{\langle id, \sigma \rangle \Downarrow n, K^{var}}$$

$$\textsc{op\_bin}\ \frac{\langle e_1, \sigma \rangle \Downarrow n_1, q_1 \qquad \langle e_2, \sigma \rangle \Downarrow n_2, q_2}{\langle e_1 \diamond e_2, \sigma \rangle \Downarrow n_1 \diamond n_2, K^{\diamond} + q_1 + q_2}$$

(b) Evaluation of expressions

$\boxed{\langle s, \sigma \rangle \Downarrow \sigma', q, l}$ executing s in state σ leads to state σ' and emits a leakage (q, l)

$$\textsc{skip}\ \frac{}{\langle \texttt{skip}, \sigma \rangle \Downarrow \sigma, 0, \epsilon} \qquad \textsc{padding}\ \frac{}{\langle \boldsymbol{\delta}(n), \sigma \rangle \Downarrow \sigma, n, \epsilon}$$

$$\textsc{assign}\ \frac{\langle e, \sigma \rangle \Downarrow n, q}{\langle id\texttt{:=}e, \sigma \rangle \Downarrow \sigma[id \leftarrow n], K^{asn} + q, \epsilon}$$

$$\textsc{seq}\ \frac{\langle p_1, \sigma \rangle \Downarrow \sigma', q_1, l_1 \qquad \langle p_2, \sigma' \rangle \Downarrow \sigma'', q_2, l_2}{\langle (p_1 \texttt{;}\ p_2), \sigma \rangle \Downarrow \sigma'', q_1 + q_2, l_1 \cdot l_2}$$

$$\textsc{if\_true}\ \frac{\langle e, \sigma \rangle \Downarrow n, q_e \qquad n \neq 0 \qquad \langle p_1, \sigma \rangle \Downarrow \sigma', q, l}{\langle \texttt{if(}e\texttt{)}\ \{p_1\}\ \texttt{else}\ \{p_2\}, \sigma \rangle \Downarrow \sigma', q_e + q, [true] \cdot l}$$

$$\textsc{if\_false}\ \frac{\langle e, \sigma \rangle \Downarrow n, q_e \qquad n = 0 \qquad \langle p_2, \sigma \rangle \Downarrow \sigma', q, l}{\langle \texttt{if(}e\texttt{)}\ \{p_1\}\ \texttt{else}\ \{p_2\}, \sigma \rangle \Downarrow \sigma', q_e + q, [false] \cdot l}$$

$$\textsc{while\_true}\ \frac{\langle e, \sigma \rangle \Downarrow n, q_e \qquad n \neq 0 \qquad \langle p, \sigma \rangle \Downarrow \sigma', q, l \qquad \langle \texttt{while(}e\texttt{)}\{p\}, \sigma' \rangle \Downarrow \sigma'', q', l'}{\langle \texttt{while(}e\texttt{)}\{p\}, \sigma \rangle \Downarrow \sigma'', q_e + q + q', [true] \cdot l \cdot l'}$$

$$\textsc{while\_false}\ \frac{\langle e, \sigma \rangle \Downarrow n, q_e \qquad n = 0}{\langle \texttt{while(}e\texttt{)}\{p\}, \sigma \rangle \Downarrow \sigma, q_e, [false]} \qquad \textsc{atomic}\ \frac{\langle p, \sigma \rangle \Downarrow \sigma', q, l}{\langle \boxed{p}, \sigma \rangle \Downarrow \sigma', q, \epsilon}$$

(c) Execution of statements, instrumented with leakages and resource consumption

Fig. 3: Syntax and big-step instrumented semantics of While

instruction $\boldsymbol{\delta}(n)$ only consumes n resources. Executing $\texttt{if}(e)\ \{s_1\}\ \texttt{else}\ \{s_2\}$ first evaluates e into n, which consumes q_e resources. If n is true (i.e., differs from zero), then s_1 is evaluated, which consumes q resources and emits a boolean leakage l. Therefore, the execution of the whole branch consumes $q_e + q$ resources. The boolean leakage is l, to which we append the value $true$.

Last, the execution of $\boxed{p}$ executes p, and erases the boolean leakage emitted by p (i.e., returns an empty boolean leakage). Indeed, contrary to the CCT policy that forbids branches depending on secret values, in our CR$^\#$policy, branches may depend on secrets as long as they are balanced. However, such branches must be inside of an atomic annotation, hence the leakage erasure in atomic annotations.

B. Semantic Definition of the CR$^\#$ Policy

We consider an attacker capable of observing the amount of resources consumed during any execution; he can exploit the consumption of resources as a side-channel attack, to deduce information about the program. Our objective is to prevent him to learn anything about the secret data manipulated by the program. In other words, we do not want the resource consumption to leak any secret data. To that purpose, we define a policy called CR$^\#$; it characterises two different executions of a program from two initial states sharing the same values of public variables. If the attacker can not distinguish both executions, then the program is considered to be CR$^\#$-secure. Moreover, we consider a program where secret (i.e., non public) input values are already known.

Definition IV.1 (Indistinguishability)**.** *Two states σ_1 and σ_2 are indistinguishable w.r.t. a list Γ of public identifiers (written as $\Gamma \vdash \sigma_1 \sim \sigma_2$) if for any public identifier $id \in \Gamma$, we have $\sigma_1[id] = \sigma_2[id]$.*

Two executions starting from two indistinguishable states will also be called two indistinguishable executions.

Definition IV.2 (CR$^\#$ security)**.** *Let p be a program, Γ be a list of public identifiers and states σ_1 and σ_2 such that $\Gamma \vdash \sigma_1 \sim \sigma_2$. Suppose that we have an execution of p from each state: $\langle p, \sigma_1\rangle \Downarrow \sigma_1', q_1, l_1$ and $\langle p, \sigma_2\rangle \Downarrow \sigma_2', q_2, l_2$. The program p is CR$^\#$-secure w.r.t. Γ (written as $\Gamma \vdash$ CR$^\#(p)$) when $(q_1, l_1) = (q_2, l_2)$.*

In the rest of this paper, we fix a list of public identifiers Γ once for all. As Γ is never modified, we will abuse notations by omitting it, thus denoting a CR$^\#$ program p with CR$^\#(p)$.

Our CR$^\#$ policy expects two indistinguishable executions to emit the same boolean leakage and consume the same amount of resources. Expecting two executions to emit the same boolean leakage resembles the definition of the CCT policy (i.e., the program must not branch on secrets), and forces two indistinguishable executions to follow the same control-flow. The CR$^\#$ policy relaxes this constraint with our atomic annotations. By erasing the boolean leakage emitted inside of atomic annotations, the CR$^\#$ policy no longer requires indistinguishable executions to follow the same control-flow. Instead, it allows the control flow to differ inside of atomic annotations, thus allowing secret-dependent branchings.

However, our CR$^\#$ policy always expects two indistinguishable executions to have an equivalent resource consumption. As a consequence, all secret-dependent branches in a program will have to be balanced, or to balance each other, in order to maintain a constant global resource consumption for these executions.

C. Examples of CR$^\#$-secure programs

Figure 4 presents examples of CR$^\#$-secure programs. For a program without atomic statement, the CR$^\#$ policy is equivalent to the CCT policy. Therefore, P_1 is CR$^\#$ if and only if its condition $\texttt{b}$ is public. Both branches of P_2 are balanced with a cost of 3. P_2 is CR$^\#$ if and only if its condition is public, just like P_1. However, $\boxed{P_2}$ is always CR$^\#$-secure, as its branches are balanced and inside atomic statements. P_3 contains two unbalanced atomic branches that balance each other. Indeed, whatever the initial value of b, the total amount of consumed resources is 7. Therefore, P_3 is always CR$^\#$-secure. This example illustrates the fact that CR$^\#$ leakages are not non-cancelling: a CR$^\#$-secure program can be composed of several non-CR$^\#$ programs, that balance each other. P_4 is similar to P_3, but only its second branch is atomic. If b is public, then P_4 is CR$^\#$-secure. However, if b is secret, P_4 may never be CR$^\#$-secure because of the first branch, even if as previously, branches balance each other. It illustrates that only atomic branches can be used to balance each other.

Last, a transformation $\mathcal{T}$ is CR$^\#$-preserving when given a CR$^\#$ program P, then $\mathcal{T}(P)$ is a CR$^\#$ program. The underlying hypothesis is that P starts from two indistinguishable states and so does $\mathcal{T}(P)$. So, there is no relation between these two pairs of states. In the same way, there is no relation between the two leakages observed during the two executions of P and $\mathcal{T}(P)$. This definition expresses the preservation of our CR$^\#$ policy, but it is too general to be proved in a simple way. For that reason, we define in Section VI and Section VII less general preservation properties that fit to our program transformations and are easier to prove.

Definition IV.3 (CR$^\#$ preservation)**.** *A transformation $\mathcal{T}$ is CR$^\#$-preserving when, for any public input Γ and any program p, $\Gamma \vdash$ CR$^\#(p) \implies \Gamma \vdash$ CR$^\#(\mathcal{T}(p))$.*

D. Relations between CCT, CR and CR$^\#$

We highlight here some interesting consequences of the definition of the CR$^\#$ policy. Firstly, we can express the CR policy with the CR$^\#$ policy. For a program p, we have $\text{CR}(p) \iff \text{CR}^\#(\boxed{p})$. In other words, CR is an instance of CR$^\#$, obtained by annotating the whole program with an atomic annotation. Secondly, we can also express the CCT policy with the CR$^\#$ policy. For a program p without any atomic annotation, then CR$^\#(p)$ forbids any secret-dependent branch in p. For such a program p, we then have the equivalence $\text{CCT}(p) \iff \text{CR}^\#(p)$.

As a consequence, it is relevant to consider the CR$^\#$ policy as a flexible mix between the CR and the CCT policies.

P_1: $\texttt{if}(b)\ \{\boldsymbol{\delta}(1)\}\ \texttt{else}\ \{\boldsymbol{\delta}(2)\}$

P_2: $\texttt{if}(b)\ \{\boldsymbol{\delta}(1);\ \boldsymbol{\delta}(2)\}\ \texttt{else}\ \{\boldsymbol{\delta}(3)\}$

P_3: $\boxed{\texttt{if}(b)\ \{\boldsymbol{\delta}(2)\}\ \texttt{else}\ \{\boldsymbol{\delta}(3)\}};\ \boldsymbol{\delta}(4);\ \boxed{\texttt{if}(b)\ \{\boldsymbol{\delta}(4)\}\ \texttt{else}\ \{\boldsymbol{\delta}(3)\}}$

P_4: $\texttt{if}(b)\ \{\boldsymbol{\delta}(2)\}\ \texttt{else}\ \{\boldsymbol{\delta}(3)\};\ \boldsymbol{\delta}(4);\ \boxed{\texttt{if}(b)\ \{\boldsymbol{\delta}(4)\}\ \texttt{else}\ \{\boldsymbol{\delta}(3)\}}$

P_1 is CR$^\#$-secure iff b is public. P_2 is CR$^\#$-secure iff b is public, but $\boxed{P_2}$ is CR$^\#$-secure. P_3 is a balanced CR$^\#$-secure program with unbalanced atomic annotations. P_4 is CR$^\#$-secure iff b is public.

Fig. 4: Example of CR$^\#$-secure programs.

The CR$^\#$ policy can observe both behaviours, depending on the added atomic annotations. It behaves as CR inside of annotations, and as CCT outside of them.

Last, if we consider a program p which is CR-secure, then it is always safe to assume that there exists a way to annotate it (denoted $p^\#$) so that we have CR$^\#(p^\#)$. Indeed, $\boxed{p}$ is always a valid candidate. However, as atomic annotations also restrict the compiler, finding candidates that contain fewer annotations yields more effective optimisations.

E. Semantic Properties of While

We conclude this section by stating two semantic properties of While that are required to prove the preservation of our security policy. First, the semantics is deterministic, both for the output state and the emitted leakage.

Lemma IV.1 (Determinism)**.** *Let p be a program and σ an initial state. If we have two executions $\langle p, \sigma\rangle \Downarrow \sigma_1, q_1, l_1$ and $\langle p, \sigma\rangle \Downarrow \sigma_2, q_2, l_2$, we then have $\sigma_1 = \sigma_2$ and $(q_1, l_1) = (q_2, l_2)$.*

The second property focuses on the non-cancellation of the boolean leakage emitted by a sequence of programs. It states that if two executions of a sequence of programs $(s;\ p)$ emit the same boolean leakage, then the two associated executions of s emit the same boolean leakage, and so do the two associated executions of p. This property is only verified by the emitted boolean leakage, but is not verified by the whole leakage (that includes the resource consumption).

Lemma IV.2 (Non-cancellation)**.** *If we have the following executions :*

(S_1) $\langle s, \sigma_1\rangle \Downarrow \sigma_1', q_1, l_1$
(P_1) $\langle p, \sigma_1'\rangle \Downarrow \sigma_1'', q_1', l_1'$
(S_2) $\langle s, \sigma_2\rangle \Downarrow \sigma_2', q_2, l_2$
(P_2) $\langle p, \sigma_2'\rangle \Downarrow \sigma_2'', q_2', l_2'$
(EQ) $l_1 \cdot l_1' = l_2 \cdot l_2'$,

then we have $l_1 = l_2$ and $l_1' = l_2'$.

Proof. We prove this lemma by reasoning by induction on the execution (S_1). We focus on some representative cases; the other cases use similar reasoning.

- Case SEQ. s is then a sequence of statements, say $s = (s_1;\ s_2)$. We apply the induction hypothesis on statements s_1 and $(s_2;\ p)$ to find that both executions of s_1 emit the same boolean leakage, and both executions of $(s_1;\ s_2)$ emit the same boolean leakage. We then apply the induction hypothesis on s_2 and p to conclude.
- Case IF. s is then an if-branch, say $s = \texttt{if}(e)\ \{s_1\}\ \texttt{else}\ \{s_2\}$. From the hypothesis (EQ), we can deduce that e evaluates to the same value in both executions of s. We consider that e evaluates to $true$. We then necessarily have two executions of s_1. We conclude by applying the induction hypothesis on s_1 and p.

□

V. Three CR$^\#$-Preserving Transformations

This section explains how to adapt in a CR$^\#$-preserving way three optimisations that are likely not to preserve the CR$^\#$ policy: constant folding, CSE and dead-store elimination. Constant folding replaces any expression evaluating to a constant value by the value itself. Every expression $e_1 \diamond e_2$, where e_1 and e_2 evaluate respectively to n_1 and n_2 is replaced by $n_1 \diamond n_2$. This transformation modifies expressions (e.g., `1+2` becomes `3`), hence the resource consumption, and is thus likely to break to CR$^\#$ policy. For example, this transformation transforms `x:=1+2` into `x:=3`, whose evaluation cost is lower. If the instruction `x:=1+2` appears in a balanced secret-dependent if-branch, the balance could be broken.

CSE computes available expressions at every program point. Any available expression is replaced by the variable storing the result of its evaluation. CSE also introduces assignments to store intermediary results, hence modifying the resource consumption. Dead-store elimination aims at removing any occurrence of an assignment that is not used later. Again, removing an instruction reduces the resource consumption and may break our security policy.

The above examples share a similar structure: the composition of a data-flow analysis, which does not modify the program, and of elementary transformations performing the optimisation. They consist mainly of substituting an expression

with another expression, introducing an assignment instruction, and removing an assignment instruction.

We implement each elementary transformation by compensating any modification in the resource consumption with a padding $\boldsymbol{\delta}$ instruction. Introducing an adequate padding requires to compute the resource consumption of the evaluation of an expression. The resource consumption of the execution of an expression does not depend on the current state. We thus introduce a cost function $\mathcal{Q} : exp \to Z$, that statically computes the cost of an expression.

Given an expression e, the substitution transformation $\mathcal{S}$ replaces the expression in the right-hand side of an assignment with e. In order to preserve the resource consumption, $\mathcal{S}$ uses $\mathcal{Q}$ to add padding, hence $\mathcal{S}$ is parametrized by an expression e and a statement, and we denote $\mathcal{S}_e$ the substitution $\mathcal{S}$ with parameter e. $\mathcal{S}_e : stmt \to stmt$ is defined as follows: $\mathcal{S}_e(id\texttt{:=}e') = id\texttt{:=}e\texttt{; }\boldsymbol{\delta}\texttt{(}\mathcal{Q}(e') - \mathcal{Q}(e)\texttt{)}$.

The insertion transformation stores the result of a temporary computation e into a fresh variable. It inserts an assignment before an arbitrary statement. The insertion is compensated with a negative padding. The way optimisations such as CSE are designed ensures that this padding will be compensated by a greater positive padding, hence the absence of negative padding in the final optimised program. The insertion transformation $\mathcal{I}_e : stmt \to stmt$ is defined as follows: $\mathcal{I}_e(p) = tmp\texttt{:=}e\texttt{; }\boldsymbol{\delta}\texttt{(}-K^{asn}-\mathcal{Q}(e)\texttt{); }p$, where *tmp* is a fresh variable identifier never used in p. The removal transformation replaces an assignment with an adequate padding. $\mathcal{R} : stmt \to stmt$ is defined as $\mathcal{R}(id\texttt{:=}e) = \boldsymbol{\delta}\texttt{(}K^{asn} + \mathcal{Q}(e)\texttt{)}$.

VI. Using Leakage Preservation to Prove CR$^{\#}$ Preservation

In order to prove that the previously defined transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ are CR$^{\#}$-preserving, we can not use standard induction reasoning. This is a consequence of the fact that CR$^{\#}$ leakages are not non-cancelling. Our solution is to proceed in two steps and conduct simpler proofs. First, we define a property called leakage preservation that is more constrained than CR$^{\#}$-preservation, and we prove that each transformation is leakage preserving. Then, we prove once for all that leakage preservation implies CR preservation. Moreover, we apply this proof scheme to the first pass of the $\mathcal{M}$ transformation introduced in Section II-B to minimise padding.

A. Leakage Preservation implies CR$^{\#}$ Preservation

We define a transformation as leakage preserving when it does not modify the leakage (i.e., resource consumption and emitted boolean leakages). It is then more constrained than the CR$^{\#}$ preservation. We define leakage preservation as a backward property that is required by theorem VI.2: given a property of the transformed program, it states a property of the source program.

Definition VI.1 (Leakage preservation)**.** *A transformation $\mathcal{T}$ is leakage preserving if, for any program p, given an execution $\langle \mathcal{T}(p), \sigma\rangle \Downarrow \sigma'$, q, l of the transformed program, there exists an state σ'' such that we have $\langle p, \sigma\rangle \Downarrow \sigma''$, q, l.*

Theorem VI.1. *Transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ are leakage preserving.*

Proof. This is a direct consequence of the definition of the transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$. □

Theorem VI.2. *Any leakage preserving transformation is CR$^{\#}$-preserving.*

Proof. Let $\mathcal{T}$ be a leakage-preserving transformation, we want to prove that $\mathcal{T}$ is CR$^{\#}$-preserving. Let p be a CR$^{\#}$-secure program, we need to prove that $\mathcal{T}(p)$ is CR$^{\#}$-secure as well. To this end, we assume having two indistinguishable executions of the transformed program $\mathcal{T}(p)$, and we then need to prove that both executions emit the same leakage. As $\mathcal{T}$ is leakage preserving, we can find two similar executions of p, which are indistinguishable as well; this highlights the fact that we need leakage preserving to be a backward property. Next, as we know that p is CR$^{\#}$-secure, we can deduce that both executions of p emit the same leakage. As these leakages are identical to the one emitted by the two executions of $\mathcal{T}(p)$, this concludes the proof. □

B. Leakage Preservation of the Normalisation Transformation

This section defines the normalisation transformation $\mathcal{N}$ and shows that it is a leakage-preserving pass. The transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ may introduce many $\boldsymbol{\delta}$ instructions, that increase the overall resource consumption, and are then factorised and minimised by the $\mathcal{M}$ pass. $\mathcal{M}$ is designed to minimize $\boldsymbol{\delta}$ instructions locally to every atomic block, and will not try to balance out $\boldsymbol{\delta}$ instructions across different atomic blocks. We decompose $\mathcal{M}$ into a normalisation pass $\mathcal{N}$ followed by a deletion pass $\mathcal{D}$ and we prove that $\mathcal{N}$ is leakage preserving, contrary to $\mathcal{D}$ (that is discussed in Section VII). The $\mathcal{N}$ pass repeatedly performs the four following basic operations until convergence, in order to merge and factorise as many $\boldsymbol{\delta}$ instructions as possible. These four operations preserve the resources consumed during an execution, and are defined as rewrite rules.

(1) *Move upwards*. First, $\boldsymbol{\delta}$ instructions are moved as upward as possible, in order to further group them together. Formally, $\mathcal{N}$ performs the following operation:

$$p\texttt{; }\boldsymbol{\delta}\texttt{(}n\texttt{)} \Rrightarrow_{\mathcal{N}} \boldsymbol{\delta}\texttt{(}n\texttt{); }p$$

(2) *Merge*. Then, $\boldsymbol{\delta}$ instructions are merged:

$$\boldsymbol{\delta}\texttt{(}n\texttt{); }\boldsymbol{\delta}\texttt{(}m\texttt{)} \Rrightarrow_{\mathcal{N}} \boldsymbol{\delta}\texttt{(}n+m\texttt{)}$$

(3) *Factorise ticks out of branches*. Next, whenever a $\boldsymbol{\delta}$ instruction appears in an if-branch, it is factorised when it appears on the opposite branch as well. Formally:

$$\texttt{if(}b\texttt{) \{}\boldsymbol{\delta}\texttt{(}n_1\texttt{); }p_1\texttt{\} else \{}\boldsymbol{\delta}\texttt{(}n_2\texttt{); }p_2\texttt{\}} \Rrightarrow_{\mathcal{N}}$$

$$\boldsymbol{\delta}\texttt{(}n\texttt{); if(}b\texttt{) \{}\boldsymbol{\delta}\texttt{(}n_1-n\texttt{); }p_1\texttt{\} else \{}\boldsymbol{\delta}\texttt{(}n_2-n\texttt{); }p_2\texttt{\}}$$

where n is the minimum between n_1 and n_2. Any resulting $\boldsymbol{\delta}(0)$ instruction is deleted. For example, we have:

$$\texttt{if}(b)\ \{\boldsymbol{\delta}(3);\ p\}\ \texttt{else}\ \{\boldsymbol{\delta}(5);\ s\} \Rrightarrow_{\mathcal{N}} \boldsymbol{\delta}(3);\ \texttt{if}(b)\ \{p\}\ \texttt{else}\ \{\boldsymbol{\delta}(2);\ s\}$$

(4) *Move out of atomic*. Whenever a $\boldsymbol{\delta}$ appears in an atomic annotation, outside of a branch, we move it out. This is the main step of the normalisation pass; it reduces the amount of $\boldsymbol{\delta}$ instructions inside of atomic annotations, to prepare for the deletion pass $\mathcal{D}$. Formally:

$$\boxed{\boldsymbol{\delta}(n);\ p} \Rrightarrow_{\mathcal{N}} \boldsymbol{\delta}(n);\ \boxed{p}$$

$\mathcal{N}$ performs these operations, along with recursive calls for the sequence of two instructions, if-branch, while loop, and atomic constructs. We can additionally notice that $\mathcal{N}$ does not move any $\boldsymbol{\delta}$ instruction outside of a loop. We argue that it is not necessary to try to do so, for the following reason. If a loop is present outside an atomic annotation, the following $\mathcal{D}$ transformation will optimise it identically (see Section VII). If a loop is present inside an atomic annotation, it may appear in a secret-dependent branch. This practice may be dangerous, but is still tolerated by our CR$^\#$ policy: such a choice is the programmer's responsibility. However, we prefer not to optimise the loop in this case, as this situation is not realistic.

Lemma VI.3. *$\mathcal{N}$ is leakage preserving.*

Proof. Let p be a program and $\langle \mathcal{N}(p), \sigma\rangle \Downarrow \sigma', q, l$ an arbitrary execution of the transformed program. We need to show that p has the same execution: $\langle p, \sigma\rangle \Downarrow \sigma', q, l$. We proceed by induction on p and examine the numerous possible cases, that all strictly preserve the leakage. □

Theorem VI.4. *$\mathcal{N}$ is CR$^\#$-preserving.*

Proof. By lemmas VI.2 and VI.3. □

VII. A Cornerstone non Leakage-Preserving Transformation

This section is devoted to our last and trickiest transformation to prove CR$^\#$ preserving, the second pass $\mathcal{D}$ of the $\mathcal{M}$ minimisation of $\boldsymbol{\delta}$ instructions (introduced in Section II-B). $\mathcal{D}$ deletes unnecessary $\boldsymbol{\delta}$ instructions while preserving balanced branches in atomic annotations. First, this section defines $\mathcal{D}$. Then, it justifies why it is CR$^\#$-preserving. Once again, we decompose the proof in two parts and define a stronger property than CR$^\#$ preservation.

A. Deletion Pass $\mathcal{D}$

The $\mathcal{D}$ pass is defined in Figure 5 using rewrite rules. Our CR$^\#$ policy considers that balance between branches must only hold inside of atomic annotations. So, deleting a $\boldsymbol{\delta}$ instruction outside of an atomic annotation has no effect on any balanced branch. The transformation $\mathcal{D}$ thus deletes any occurrence of a $\boldsymbol{\delta}$ instruction outside of an atomic annotation. However, $\boldsymbol{\delta}$ instructions inside of atomic annotations are not deleted, in order to preserve the balance between potential balanced secret-dependent branches. For example, we have $\boldsymbol{\delta}(2);\ \boxed{\boldsymbol{\delta}(3)} \Rrightarrow_{\mathcal{D}} \boxed{\boldsymbol{\delta}(3)}$. Let us recall that they result from the factorisation of $\boldsymbol{\delta}$ instructions by the previous pass $\mathcal{N}$ of the $\mathcal{M}$ minimisation.

$$\frac{}{\texttt{skip} \Rrightarrow_{\mathcal{D}} \texttt{skip}} \qquad \frac{}{\boldsymbol{\delta}(n) \Rrightarrow_{\mathcal{D}} \texttt{skip}}$$

$$\frac{p_1 \Rrightarrow_{\mathcal{D}} p_1' \qquad p_2 \Rrightarrow_{\mathcal{D}} p_2'}{p_1;\ p_2 \Rrightarrow_{\mathcal{D}} p_1';\ p_2'}$$

$$\frac{p_1 \Rrightarrow_{\mathcal{D}} p_1' \qquad p_2 \Rrightarrow_{\mathcal{D}} p_2'}{\texttt{if}(c)\ \{p_1\}\ \texttt{else}\ \{p_2\} \Rrightarrow_{\mathcal{D}} \texttt{if}(c)\ \{p_1'\}\ \texttt{else}\ \{p_2'\}}$$

$$\frac{p \Rrightarrow_{\mathcal{D}} p'}{\texttt{while}(c)\{p\} \Rrightarrow_{\mathcal{D}} \texttt{while}(c)\{p'\}} \qquad \frac{}{\boxed{p} \Rrightarrow_{\mathcal{D}} \boxed{p}}$$

Fig. 5: Definition of the $\mathcal{D}$ transformation, with rewrite rules.

B. Proving that the Deletion Pass is CR$^\#$ Preserving

Contrary to all the passes presented so far, $\mathcal{D}$ deletes some $\boldsymbol{\delta}$ instructions, hence explicitly modifying the resource consumption. So, $\mathcal{D}$ is not leakage preserving. Instead, we rely on the property that $\mathcal{D}$ only removes $\boldsymbol{\delta}$ instructions that are outside of atomic annotations. As a consequence, its impact on resource consumption must not depend on secret input values, as all secret dependent if-branches appear inside an atomic annotation.

More precisely, for a given program, let us consider two executions emitting the same boolean leakage, meaning that outside of atomic annotations, both executions follow the same path during the execution. Transforming both executions with $\mathcal{D}$ will have the same impact on resource consumption. Indeed, outside of atomic annotations, as both executions follow the same path, the transformation will have a similar impact on resource consumption. Moreover, inside of atomic annotations, the program is not modified, and neither are the executions. So, we define a new policy called LPo for "leakage preservation with constant resource offset", which captures this behaviour.

Definition VII.1 (LPo)**.** *A transformation $\mathcal{T}$ is LPo if, for any program p and two of its executions $\langle p, \sigma_1\rangle \Downarrow \sigma_1', q_1, l$ and $\langle p, \sigma_2\rangle \Downarrow \sigma_2', q_2, l$, then there exists a resource offset $r \in \mathbb{Z}$ (i.e., a same offset in resource consumption) such that we have $\langle \mathcal{T}(p), \sigma_1\rangle \Downarrow \sigma_1', q_1 + r, l$ and $\langle \mathcal{T}(p), \sigma_2\rangle \Downarrow \sigma_2', q_2 + r, l$.*

LPo is not sufficient to imply that the transformation is CR$^\#$-preserving. We further require the transformation to satisfy a property called "termination preservation". It states that if an execution of a transformed program from an initial state σ terminates, then an execution of the source program from the same initial state σ also terminates. Similarly to leakage preservation, termination preservation is a backward property.

Definition VII.2 (Termination preservation)**.** *A transformation $\mathcal{T}$ is termination preserving if, for any program p, supposing that we have an execution $\langle \mathcal{T}(p), \sigma_1 \rangle \Downarrow \sigma_2, q, l$ of the transformed program, then there exists an output state σ' and a leakage (q', l') such that we have the execution $\langle p, \sigma_1 \rangle \Downarrow \sigma', q', l'$.*

The following lemma states that any transformation complying to the two previous properties is CR$^\#$-preserving. We then prove that $\mathcal{D}$ complies to both properties, hence to CR$^\#$ preservation.

Theorem VII.1. *Any LPo and termination preserving transformation is CR preserving.*

Proof. Let $\mathcal{T}$ be a transformation, that is LPo and termination preserving. Let p be a CR$^\#$-secure program, we need to prove that $\mathcal{T}(p)$ is CR$^\#$-secure as well. To this end, we assume having two indistinguishable executions E_1 and E_2 of $\mathcal{T}(p)$, and we then need to prove that they emit the same leakage. As $\mathcal{T}$ is termination preserving, we can find two executions of p emitting unknown leakages. However, as p is CR$^\#$-secure, these unknown leakages are equal. Let (q, l) be such a leakage.

As E_1 and E_2 emit the same boolean leakage l, we use the fact that $\mathcal{T}$ is LPo to find a resource offset $r \in \mathbb{Z}$ such that we have two executions of $\mathcal{T}(p)$ with similar initial states and emitting the same leakage $(q + r, l)$. Finally, as While is deterministic (see lemma IV.1), these two executions of $\mathcal{T}(p)$ are exactly the executions E_1 and E_2. As E_1 and E_2 emit the same leakage $(q + r, l)$, this concludes the proof. □

Lemma VII.2. *$\mathcal{D}$ is termination preserving.*

Proof. By induction on the execution of the transformed program. □

Lemma VII.3. *$\mathcal{D}$ is LPo.*

Proof. Let p be a program, we assume to have two executions of p emitting the same boolean leakage l, and consuming respectively q_1 and q_2 resources. We need to find an offset $r \in \mathbb{Z}$, such that the associated executions of $\mathcal{D}(p)$ emit the same boolean leakage l, and consume respectively $q_1 + r$ and $q_2 + r$ resources. We reason by induction on the semantics of one of the source executions. We focus on some of the interesting cases.

- Case TICK. We have $p = \boldsymbol{\delta}(n)$ and $\mathcal{D}(p) = \texttt{skip}$. We choose $r = -n$.
- Case SEQ. We have $p = p_1; p_2$, two executions of p_1 and two executions p_2. As the boolean leakage is non-cancelling (see lemma IV.2), we conclude that both executions of p_1 emit the same boolean leakage l_1, and both executions of p_2 emit the same boolean leakage l_2. Then we use the induction hypothesis and these two pairs of executions to conclude.
- Case IF. We have $p = \texttt{if}(e)\ \{p_1\}\ \texttt{else}\ \{p_2\}$. As both executions have the same boolean leakage, e evaluates to the same value in both executions. When e evaluates to true (resp. false), we have two executions of p_1 (resp. p_2), producing the same boolean leakage. We then use the induction hypothesis on the executions of p_1 (resp. p_2) to conclude.

□

Theorem VII.4. *$\mathcal{D}$ is CR-preserving.*

Proof. By lemmas VII.2 and VII.3, and theorem VII.1. □

VIII. Related Work

A. Timing non-interference

Our work focuses on a timing non-interference policy, which was first introduced in [3]. In [3], the authors define a type system in which well-typed programs do not leak secret information. This is a direct adaptation of [26] but it adds control of timing leaks on high branches. Their type system is undecidable because they rely on an undecidable semantic judgement to check that high branches have equal timing costs. But their approach can be refined with a more conservative judgement to become executable and directly adapted to enforce the CR policy. When high branches are not time-balanced but the program is typable with respect to the type system of [26], they also show how to repair the program by suitably padding both branches. Using this approach in our setting could help repairing CR after a unsecure compiler transformation but will require running the type checking of [26] after each compilation pass, even on low-level languages where taint analysis is often too conservative to succeed. Our approach avoids running a taint analysis inside the compiler, thanks to our atomic annotations.

In [20], the authors introduce a timing non-interference policy called CR. We extended this policy to also include control-flow leakages. They present a type system used to verify that an implementation respects their policy. They also show how this type system can automatically remove vulnerabilities from a program. In other works [16], [13], the authors use a similar notion of resource consumption to establish precise bounds for worst and best cases resource usage. The main difference with our work is that we focus on the preservation of a variation of this security policy. Our current paper does not put to much emphasis on enforcing CR and CR$^\#$ at source level because the work of [3] can be easily adapted to do it, by positioning carefully atomic annotations at source level.

The work that is closest to ours is [6], where the authors use another relaxation of the CCT policy called time balancing and defined as negligibly influenced by secrets. To ensure that a program respects this policy, a global timing counter is added to measure the timing differences between branchings. Then, the Boogie deductive verifier checks for each program the constraints required by the policy. This work is not formally verified with a proof assistant but a tool was implemented to verify that the Amazon's s2n implementation of TLS respects the time-balancing policy. Similarly to our work, the author use padding or dummy instructions in order to balance branches in the program. Our work differs as we use

an instrumented operational semantics to model the timing behavior of a program. Their policy also differs, as it allows pairs of executions with different execution time, as long as this difference is bounded by a given constant value.

In [24], the authors study a different but close policy, where the strength of a side-channel leak is measured by a notion of entropy, and then propose methods to reduce the entropy of the leakage emitted by a program. These methods rely on the insertion of padding instructions to increase the execution time of branches of the program, which is similar to the padding we use. The property they study can be seen as a generalisation of the CR policy, as the latter expects a constant leakage, i.e., a leakage of null entropy.

B. Preservation of side-channel security through compilation

Our work focuses on the preservation of side-channel security during compilation, as our setting assumes the source programs to be secure. This is a standard approach that we used in [7]. We formally verified the problem of CCT preservation in the CompCert compiler, using the Coq proof assistant. We presented a modified version of the CompCert compiler, and proved it preserves the CCT policy, applying the proof methodology previously presented in [9], and using advanced 2-simulations relying on non-cancellation. Our approach differs here as the leakage we consider does not satisfies the non-cancellation property.

Other works consider the problem of CCT preservation. Jasmin [4] is a programming framework, allowing the programmer to write programs in the Jasmin programming language. The Jasmin compiler then compiles programs down to efficient assembly code.

In [12], the authors present FaCT, a domain specific language addressing the challenge of writing human-readable constant-time cryptographic code. The language provides high-level constructs, that are compiled by the FaCT compiler down to constant-time LLVM bitcode. It is designed to make cryptographic libraries easier to implement. Indeed, a FaCT developer can focus on the correctness of the implementation, and then rely on the compiler to apply usual recipes (such as bitwise operations) yielding a constant-time compiled program. The FaCT compiler relies on a static information-flow type system. The type system allows to annotate variables as secret or public, then reject unsafe programs. However, they only rely on empirical evaluation, using dudect [21], to ensure that the generated code has a constant-time behaviour.

In [19], the authors present a flow-sensitive dependent type system for shared-memory programs, which enforces a strong policy: timing-sensitive non-interference for concurrent program. Then in [23], the authors study the preservation of this policy during compilation. Similarly to our work, the authors formally verify the preservation of their policy through compilation from a while language. However, as our CR$^{\#}$ security policy does not rely on a type system at target level, the proof of preservation of our policy is easier to achieve, and our approach avoids the use of an information-flow type system in the compilation chain.

In [11], the authors consider the preservation of Information-Flow during compilation. The property they study does not deal with timing side-channels, but rather considers an attacker capable of observing an arbitrary amount of information during an execution. Their policy ensures that a compiled program doesn't leak more information than the associated source program, for instance by optimizing away code erasing secret values in the memory. They also present proof principles designed to prove that a transformation preserves their security policy. It is not clear how their policy can express the CR policy we study here. In recent work [14], the authors consider a similar problem with a different approach. They introduce an opaque annotation, and a security policy enforcing that any observation occurring in an opaque area must be preserved through compilation, thus allowing to prevent unwanted dead code elimination. They study the preservation of their policy from C code to machine code. Interestingly, their opaque annotation is similar to our atomic annotation, as it disables aggressive optimizations, while the compiler does not need to know the security taint of the variables.

C. Hardware instructions as mitigation.

Our paper introduces a source level annotation that can restrain the compiler. However, these annotations may also be interesting at low level machine code, to restrain hardware mechanisms such as cache or speculative execution, and thus make branch balancing more reliable. Ideally, we would like to implement atomic annotations in a way that disables cache, pipeline and speculative execution inside of atomic blocks. To the best of our knowledge, such precise control over all these hardware mechanisms is not possible on modern architectures. Still, memory fence instructions are a first step toward our goal, as they prevent speculative execution at a given program point.

Recent work [25] presents Blade, an automatic tool able to repair a program vulnerable to speculative execution attacks, by eliminating speculation-based leakage. Their approach is based on the insertion of a minimal amount of *protect* annotations in the source code, which may then be implemented as memory fence instructions at low level machine code.

In [8], the authors formalize a notion of speculative semantics and the speculative constant-time policy, which captures programs whose every possible speculative execution respects the constant-time policy. They then implement their methods in the Jasmin verification framework, and use it to implement speculatively constant-time cryptographic primitives. These implementations also rely on memory fence instructions to prevent speculative execution until prior instructions have completed. The speculative semantics of the extended Jasmin language also depends on a fence instruction that exists at source level.

IX. Conclusion

We formalised the CR$^{\#}$ security policy, a stronger policy than the CR policy which is used by some cryptographic practitioners. We also formalised a methodology to adapt

program transformations so that they become CR$^{\#}$ preserving; it relies on adding padding to balance secret branchings and minimisation of padding. We proved that different transformations used by compiler optimisations are CR$^{\#}$ preserving. Last, we introduced an annotation called atomic, used to delimit the high-security parts of the program. This annotation indicates where to restrict the compiler optimisations in order to preserve the CR$^{\#}$ policy.

As future work, we will extend our security policy to handle memory accesses. We suggest to reuse the work done for CCT preservation, where it is forbidden to access memory when the address depends on a secret, and also to prohibit it in secret-dependent if-branches. We also plan to study more realistic cost models, that can account for non-local behaviors, such as cache misses and branch prediction. In a longer term, we intend to apply our methodology to more realistic languages such as those of the CompCert compiler.

ACKNOWLEDGMENTS

This work is supported by a European Research Council (ERC) Consolidator Grant for the project "VESTA", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 772568).

REFERENCES

[1] Abate, C., Blanco, R., Ciobâcă, Ş., Durier, A., Garg, D., Hritcu, C., Patrignani, M., Tanter, É., Thibault, J.: Trace-relating compiler correctness and secure compilation. In: Müller, P. (ed.) ESOP. LNCS, vol. 12075, pp. 1–28. Springer (2020)

[2] Abate, C., Blanco, R., Garg, D., Hritcu, C., Patrignani, M., Thibault, J.: Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In: Comp. Sec. Foundations Symposium, CSF. pp. 256–271. IEEE (2019)

[3] Agat, J.: Transforming out timing leaks. In: POPL. p. 40–53. ACM (2000)

[4] Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-assurance and high-speed cryptography. In: CCS. pp. 1807–1823. ACM (2017)

[5] Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: USENIX Security Symp. pp. 53–70 (2016)

[6] Athanasiou, K., Cook, B., Emmi, M., MacCarthaigh, C., Schwartz-Narbonne, D., Tasiran, S.: Sidetrail: Verifying time-balancing of cryptosystems. In: Verified Software, Theories, Tools, and Experiments (VSTTE). pp. 215–228. Springer (2018)

[7] Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D., Trieu, A.: Formal verification of a constant-time preserving C compiler. POPL **4**, 1–30 (2019)

[8] Barthe, G., Cauligi, S., Grégoire, B., Koutsos, A., Liao, K., Oliveira, T., Priya, S., Rezk, T., Schwabe, P.: High-assurance cryptography in the spectre era (2021)

[9] Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In: Computer Security Foundations Symp. (CSF). pp. 328–343. IEEE (2018)

[10] Bernstein, D.J.: Cache-timing attacks on AES (2005), `http://cr.yp.to/papers.html`

[11] Besson, F., Dang, A., Jensen, T.: Information-flow preservation in compiler optimisations. In: 2019 IEEE 32nd Computer Security Foundations Symposium (CSF). pp. 230–23012. IEEE (2019)

[12] Cauligi, S., Soeller, G., Johannesmeyer, B., Brown, F., Wahby, R.S., Renner, J., Grégoire, B., Barthe, G., Jhala, R., Stefan, D.: FaCT: a DSL for timing-sensitive computation. In: PLDI. pp. 174–189 (2019)

[13] Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational cost analysis. ACM SIGPLAN Notices **52**(1), 316–329 (2017)

[14] Cohen, A., De Grandmaison, A., Guillon, C., Heydemann, K., Vu, S.T.: Secure optimization through opaque observations (2021)

[15] D'Silva, V., Payer, M., Song, D.X.: The correctness-security gap in compiler optimization. In: Symp. on Security and Privacy Workshops. pp. 73–87. IEEE (2015)

[16] Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: POPL. pp. 357–370 (2011)

[17] Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: POPL. pp. 179–192. ACM (2014)

[18] Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009)

[19] Murray, T., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional verification and refinement of concurrent value-dependent noninterference. In: Computer Security Foundations Symp. (CSF). pp. 417–431. IEEE (2016)

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

[21] Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Design, Automation & Test in Europe Conf. (DATE). pp. 1697–1702. IEEE (2017)

[22] Simon, L., Chisnall, D., Anderson, R.J.: What you get is what you C: controlling side effects in mainstream C compilers. In: EuroS&P. pp. 1–15. IEEE (2018)

[23] Sison, R., Murray, T.: Verifying that a compiler preserves concurrent value-dependent information-flow security. arXiv preprint arXiv:1907.00713 (2019)

[24] Tizpaz-Niari, S., Černỳ, P., Trivedi, A.: Quantitative mitigation of timing side channels. In: International Conference on Computer Aided Verification. pp. 140–160. Springer (2019)

[25] Vassena, M., Disselkoen, C., Gleissenthall, K.v., Cauligi, S., Kıcı, R.G., Jhala, R., Tullsen, D., Stefan, D.: Automatically eliminating speculative leaks from cryptographic code with blade. Proceedings of the ACM on Programming Languages **5**(POPL), 1–30 (2021)

[26] Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of computer security **4**(2-3), 167–187 (1996)

[27] Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: POPL. pp. 427–440. ACM (2012)