Formal Verification of Side Channel Countermeasures Using Self-Composition

J. Bacelar Almeida*, Manuel Barbosa*, Jorge S. Pinto*, Bárbara Vieira*

*CCTC/Departamento de Informática,
Universidade do Minho, Campus de Gualtar,
4710-057 Braga, Portugal*

Abstract

Formal verification of cryptographic software implementations poses significant challenges for off-the-shelf tools. This is due to the domain-specific characteristics of the code, involving aggressive optimisations and non-functional security requirements, namely the critical aspect of countermeasures against side-channel attacks. In this paper we extend previous results supporting the practicality of self-composition proofs of non-interference and generalisations thereof. We tackle the formal verification of high-level security policies adopted in the implementation of the recently proposed NaCl cryptographic library. We formalize these policies and propose a formal verification approach based on self-composition, extending the range of security policies that could previously be handled using this technique. We demonstrate our results by addressing compliance with the NaCl security policies in real-world cryptographic code, highlighting the potential for automation of our techniques.

Keywords: Cryptographic algorithms, program verification, program equivalence, self-composition, side-channel countermeasures.

1. Introduction

Software implementations of cryptographic algorithms and protocols are at the core of security functionality in many IT products. However, the development of this class of software products is understudied as a domain-specific niche in software engineering. The development of cryptographic software is clearly distinct from other areas of software engineering due to a combination of factors. Cryptographic software engineering is *interdisciplinary*, drawing on skills from mathematics, computer science and electrical engineering; it requires developing aggressively *optimised code*, as light as possible in terms of computational and communications load, to compensate for

*Corresponding author
Email addresses: `jba@di.uminho.pt` (J. Bacelar Almeida), `mbb@di.uminho.pt` (Manuel Barbosa), `jsp@di.uminho.pt` (Jorge S. Pinto), `barbarasv@di.uminho.pt` (Bárbara Vieira)

the typically low perceived benefits; finally, it requires writing and optimising code for *heterogeneous architectures*, ranging from embedded processors with very limited computational power, memory and autonomy, to high-end servers with low-latency.

Side-channel countermeasures. One of the most challenging aspects of cryptographic software implementation is the fact that functional correctness is *not* a sufficient condition to guarantee security. It is possible (and likely) that a naive implementation of a theoretically secure cryptographic algorithm is functionally correct, and yet turns out to be insecure. This is because cryptographic algorithms are designed and validated, in theory, by idealizing the computational platform in which they will execute: computation is seen as taking place inside a black box, from which only explicitly released outputs can be extracted. In practice, this is far from the truth, as physical observation of computational platforms can enable an adversary to recover sensitive information, often with very little effort. This type of attack is usually called a *side-channel attack*.

Protection against side-channel attacks is one of the most active areas of research in applied cryptography, involving both hardware and software implementation aspects. On the hardware side, the goal is to devise a platform that aproximates the idealized black-box mentioned above. Smart-cards, for example, incorporate various hardware countermeasures to reduce exposure to side-channel attacks, e.g. by minimizing power consumption fluctuations when different operations are executed by the processor. However, it is not realistic to assume that one can resort to special purpose hardware whenever one needs to employ cryptography. Furthermore, hardware countermeasures are, by design, meant to thwart specific forms of physical data collection, which means that there is always room for new sources of leakage to be uncovered and exploited. Finally, even the most advanced cryptographic hardware cannot protect against side channel leakage caused by bad software implementation choices.

Software side-channel countermeasures aim to minimize the correlation between the sensitive inputs to the algorithm and physically observable variations in the behavior of the underlying computational platform, when the algorithm is executing. In this work we focus on a particular class of countermeasures aiming to eliminate timing dependencies, in both execution and memory access times, that may give rise to so-called timing attacks. Concretely, code is written so as to ensure that the sequence of executed instructions (i.e. the control flow) and the sequence of accessed memory addresses are independent of the sensitive inputs. We refer the interested reader to [15] for details on programming techniques that make this possible without forsaking performance.

We consider deductive formal verification as a means to obtain further guarantees that these side-channel countermeasures are correctly deployed in cryptographic software implementations. In practice, these guarantees are important not only for the end-users of the code, but also for developers working, say, in collaborative projects, in which the eligibility of contributions must be analyzed with respect to well-defined code quality criteria. Transferral of an increased level of assurance to a third party may be necessary, for example, in the context of software certification processes.

A formal verification-based approach. Deductive program verification is the area of Formal Methods that attempts to check properties of software statically with the help of an axiomatic semantics of the underlying programming language and a proof tool.

The area has greatly benefited from recent evolutions, including theoretical developments in the treatment of linked data structures; the adoption of standard interface specification languages for writing *contracts* annotated into the programs; and developments in automated proof technology, in particular SMT solvers. Verification tools for languages like C [8], C# [6], or Java [16] are becoming more and more popular.

Our strategy to formally verify compliance to security policies such as those described above, which enforce the elimination of control flow and memory access dependencies as countermeasures against timing side-channel attacks, is to view them as information flow security restrictions. Information flow security refers to a class of security policies that constrain the ways in which information can be manipulated during program execution. These properties can be formulated in terms of *non-interference* between high-confidentiality input variables and low-confidentiality output variables. A dual formulation permits capturing security policies that constrain information flow from non-trustworthy (or low-integrity) inputs, to trusted (or high-integrity) outputs.

Self-composition [7] is a technique that permits formalizing non-interference properties by considering two copies of the same program: a new program is constructed consisting of the original program composed with itself, with the caveat that the two copies of the original program operate over disjoint parts of the state. Non-interference is specified and verified by defining an appropriate contract for the composed program.

In this work, we build on the results of our previous work, where the applicability of self-composition, and generalizations thereof to the formal verification of cryptographic software, have been demonstrated. More specifically, we have proposed [3, 4] a *composition*-based methodology for proving properties about the semantics of two (possibly identical) programs. To increase the level of automation, *natural invariants* were employed as a device to establish a correspondence between axiomatic properties of programs and their operational semantics. Natural invariants are particularly useful for reasoning about pairs of programs with sufficiently close control structures.

One possible application of this approach is to establish the functional correctness of (cryptographic) programs, by showing that a concrete implementation is functionally equivalent to a specification given as a reference implementation. We have in our previous papers shown that functional correctness can be addressed using a sequence of equivalence proofs, each one corresponding to a simple refinement, so that the control structures of the programs in each pairwise equivalence are sufficiently close. Proofs of non-interference by self-composition are another subclass of problems that can be tackled using the same methodology: self-composition is a particularly convenient case, in which the control structures of both programs are identical.

The applicability of our techniques has been demonstrated on practical examples of cryptographic code, using an off-the-shelf formal verification tool. Our techniques solve some of the automation problems that had previously been identified for the self-composition technique [26]. This work will be revised in more detail in Section 2.

Contributions. In this paper we extend the range of applications of the methods introduced in our previous work, to cope with a set of high-level security policies adopted by the developers of the recently proposed NaCl [9] (read *salt*) cryptographic library. These policies, quoted in Figure 1 from the NaCl specifications, enforce software countermeasures against timing side-channel attacks. The introduction of an instrumented

No data-dependent branches. *The CPU's instruction pointer, branch predictor, etc. are not designed to keep information secret. For performance reasons this situation is unlikely to change. The literature has many examples of successful timing attacks that extracted secret keys from these parts of the CPU. NaCl systematically avoids all data flow from secret information to the instruction pointer and the branch predictor. There are no conditional branches with conditions based on secret information; in particular, all loop counts are predictable in advance. This protection appears to be compatible with extremely high speed, so there is no reason to consider weaker protections.*

No data-dependent array indices. *The CPU's cache, TLB, etc. are not designed to keep addresses secret. For performance reasons this situation is unlikely to change. The literature has several examples of successful cache-timing attacks that used secret information leaked through addresses. NaCl systematically avoids all data flow from secret information to the addresses used in load instructions and store instructions. There are no array lookups with indices based on secret information; the pattern of memory access is predictable in advance. The conventional wisdom for many years was that achieving acceptable software speed for AES required variable-index array lookups, exposing the AES key to side-channel attacks, specifically cache-timing attacks. However, the paper "Faster and timing-attack resistant AES-GCM" by Emilia Käsper and Peter Schwabe at CHES 2009 introduced a new implementation that set record-setting speeds for AES on the popular Core 2 CPU despite being immune to cache-timing attacks. NaCl reuses these results.*

Figure 1: NaCl Security Policies

trace semantics (Section 3.2) plays a key role here, since it makes possible to express these policies as non-interference properties, thus allowing us to bridge an important gap between the general, theoretical formulation of security properties employed in our previous work, and the real-world concerns and coding practices of cryptographers.

We address the problem at the C source code level; our motivation is twofold:

1. Our goal is to enable the formal verification of claims that were, until now, stated and checked in an informal way. Our solution is designed to respond to the concrete needs of cryptographers, by focusing on existing security policies and source-code that are used in real-world applications. In Section 3.3 we provide formal definitions of these security policies, so that their purpose and reach can be better understood. We also use these definitions to precisely justify the guarantees provided by our formal verification approach.
2. Our solution is based on off-the-shelf formal verification tools and the composition-based methodology mentioned above (and reviewed in Section 2). By using existing tools, we are able to anchor the trust that may be deposited in our approach on a well-established and standard class of tools and techniques. At the same time, we demonstrate the applicability of existing technology to novel application areas, namely the formal verification of countermeasures against wider classes of side-channel attacks, and we demonstrate the potential for automating the verification of cryptographic software by self-composition, showing that we can tackle a wider class of programs than previous approaches in the same line.

Our approach can be summarized as follows. We first define the operational semantics of a While language with applicative arrays, which explicitly captures the flavour of side-channel leakage addressed by the NaCl security policies. Concretely, the semantics constructs traces of the memory addresses read or written to by a program, including program and data memory. Based on this, we propose a definition of *secure program* in the sense intended by the NaCl developers. This is essentially a termination-sensitive non-interference requirement stating that the address traces should be independent of secret data. Technically, our security notion can be seen as an extension of the Program Counter Model of [19, 25], where we add the capability to handle a wider range of attacks, including cache timing attacks [23] and branch prediction analysis attacks [1] by extending the model to cover data memory access patterns.

To formally verify that a program meets the previous definition of security it then suffices to proceed with the following two steps:

1. One transforms the original program P into one that explicitly collects in its output state (minimal) additional information about the execution of P; and
2. One then formally verifies (using the composition-based methods introduced in [3, 4]) that this extra information is independent of secret data.

We theoretically validate this technique by showing that a proof of safety (including termination) of a program, and a proof of non-interference for the corresponding transformed program, together imply that the original program is indeed secure with respect to the intended security policy. The details are described in Section 3.

Finally, we discuss how our proof techniques can be deployed using real-world deductive verification tools, namely the `Frama-C` framework. We cover in Section 4 practical examples extracted form the NaCl cryptographic library, highlighting the potential for automation of the program transformation and self-composition proofs using natural invariants. In doing so, we answer questions raised in [25, 26] regarding the feasibility of addressing these problems using off-the-shelf verification tools. Concretely, we show that it is possible to carry out verification directly over the composed program, for a much wider class of programs than was previously achieved. Furthermore, we do not need to transform the input program into a more convenient form that goes around the limitations of the verification framework. In Sections 4 and 5 we further elaborate on the differences and improvements with respect to related work.

Organisation. Our previous work of [3, 4], in particular the application of our methodology for proofs by composition to the self-composition case, are revised with substantial detail in Section 2, where we review the self-composition technique, the notion of natural invariant, and the self-composition lemmas that (embedded in the verification tool) play a central role in the approach. An example (from the NaCl library) is also given. Section 3 then introduces our formal framework supporting the verification of side-channel countermeasures: an instrumented semantics is introduced, and the security policy is formally expressed based on it. Properties of the semantics and of the notion of security introduced are studied, and we prove that the latter notion can indeed be verified by self-composition. Section 4 presents the details of our study of security aspects of the NaCl cryptographic library, based on the results of the previous section. Finally, we discuss related work in Section 5, and conclude the paper in Section 6.

2. Background

In this section we first review the *self-composition* technique [7], which permits using deductive verification to prove non-interference properties. We then review *natural invariants*, a technique introduced in [3, 4] to address some of the difficulties of applying self-composition in practice, namely in the concrete domain of formally verifying cryptographic code. We complete the section with an illustration of these techniques over a concrete example of C code, preceded by a short introduction to the practical formal verification infrastructure that we rely on.

2.1. Proofs by Self-Composition

Information flow properties are usually verified using a special extended type system [30, 20, 5]. Type-based analyses, which track assignments to low security variables, can be too restrictive [7]. An alternative, less conservative approach, based on the language semantics, is to define a program as secure if different terminating executions, starting from states that differ only in the values of high-security variables, result in final states that are equivalent regarding the values of low-security variables.

Formally, let V_H and V_L denote the sets of high-security and low-security variables of program C, and $V'_L = \mathit{Vars}(C) \setminus V_H$. Intuitively, V_L represents the parts of the state (typically program outputs) that are explicitly tagged as being observable by an attacker, whereas the remainder of the state is assumed *not* to be observable; conversely, V_H corresponds to the parts of the state (typically inputs) that are explicitly tagged as containing sensitive information.

We write $(C,\sigma) \Downarrow \tau$ to denote the fact that when executed in state σ, C stops in state τ (states are functions mapping variables to values; $\Downarrow$ is the evaluation relation in a big-step semantics of the underlying language). We consider *termination-insensitive* and *termination sensitive* definitions of security. The former says nothing about information leaked when the initial state causes the program to not terminate. The latter, stronger notion, requires (for deterministic programs) that low-equivalent initial states have consistent termination behavior (either all terminate or none terminate). C is said to be secure if for arbitrary states σ, τ,

$$(\textit{termination-insensitive}) \qquad \sigma \overset{V'_L}{=} \tau \wedge (C,\sigma) \Downarrow \sigma' \wedge (C,\tau) \Downarrow \tau' \implies \sigma' \overset{V_L}{=} \tau'$$
$$(\textit{termination-sensitive}) \qquad \sigma \overset{V'_L}{=} \tau \wedge (C,\sigma) \Downarrow \sigma' \implies (C,\tau) \Downarrow \tau' \wedge \sigma' \overset{V_L}{=} \tau'$$

where $\sigma \overset{X}{=} \tau$ denotes that $\sigma(x) = \tau(x)$ for all $x \in X$, i.e. σ and τ are X-indistinguishable.

The operational definition of non-interference involves two executions of the program. The *self-composition* technique [7] allows this to be reformulated considering a single execution of a transformed program. Given a (deterministic) program C, let C^s be the program that is equal to C except that every variable x is renamed to a fresh variable x^s. Termination-insensitive non-interference can be stated considering a single execution of the self-composed program $C;C^s$ as follows:

If $\sigma(x) = \sigma(x^s)$ for all $x \in V'_L$ and $(C;C^s,\sigma) \Downarrow \sigma'$, then $\sigma'(x) = \sigma'(x^s)$ for all $x \in V_L$.

In other words, C is information-flow secure if starting from a state in which pairs of variables x, x^s may have different values only if x is high-security, any terminating execution of the self-composed program results in a final state in which pairs of variables x, x^s, with x low-security, have necessarily the same value. This allows for a shift to an axiomatic semantics-based definition, as the following partial correctness Hoare triple:

$$\left\{ \bigwedge\nolimits_{x \in V'_L} x = x^s \right\} C;C^s \left\{ \bigwedge\nolimits_{x \in V_L} x = x^s \right\}$$

Note that strengthening this to a total correctness specification yields a notion of non-interference that is stronger than termination sensitive non-interference.

2.2. Natural Invariants

An obvious difficulty in carrying out the verification of self-composed programs comes from the absence of appropriate loop invariants. In what follows we revise the general approach to this problem introduced in [3, 4]. We present the application of these techniques to the self-composition case, although the original work addressed a more general view of proofs by composition that also allowed for proofs of functional correctness. In short, it consists of the following steps:

1. Extracting a specification of a program from its relational semantics. The critical point of the verification process is the automatic construction of appropriate loop invariants that constitute the *natural* specification of the program. Each invariant is turned into a predicate, used to annotate the respective loop in the source code.
2. Identifying and interactively proving additional facts involving the named invariant predicates. These are written as lemmas that capture the non-trivial parts of the proofs required for verification.
3. Augmenting the source file with the previous lemmas, which are justified once-and-for-all by interactive proofs. The availability of these lemmas will allow automatic provers to carry out the verification process, validating the potentially large number verification conditions generated by the self-composition proofs.

When both programs share much of the underlying control structure, as is the case in self-composition proofs, the user may easily guide the interactive verification process by providing as hints the relevant lemmas. The remaining parts can be checked with a high degree of automation.

Relational Specification. For concreteness, we consider a simple While language with integer expressions and arrays. Its syntax is given by:

Operators	op ::=	`+` | `-` | `*` | `/` | `=` | `!=` | `<`
Expressions	e ::=	**n** | `x` | $e\ op\ e$ | `a[`e`]`
Commands	C ::=	**skip** | `x:=`e | `a[`e`]:=`e | **if** e **then** C_1 **else** C_2 | **while** (e) C | C_1 `;` C_2

Instead of a distinct syntactic class for boolean expressions, we adopt the *C* convention of interpreting zero/non-zero integer expressions as truth values. Literals are ranged by **n**, and integer and array variables are ranged by `x` and `a` respectively. Instead of variable declarations, we consider a fixed `State` type that keeps track of all the variable values during execution. Integer variables are interpreted as (unbound) integers, and arrays as functions from integers to integers (no size/range checking). Array operations $\mathrm{acc} : (\mathrm{Z} \to \mathrm{Z}) \times \mathrm{Z} \to \mathrm{Z}$ and $\mathrm{upd} : (\mathrm{Z} \to \mathrm{Z}) \times \mathrm{Z} \times \mathrm{Z} \to (\mathrm{Z} \to \mathrm{Z})$ are axiomatised as usual:

$$\mathrm{acc}(\mathrm{upd}(a,k,x),k) = x \qquad \mathrm{acc}(\mathrm{upd}(a,k',x),k) = \mathrm{acc}(a,k) \qquad \text{if } k \neq k'.$$

The *State* type is defined as the cartesian product of the corresponding interpretation domains (each variable is associated to a particular position). We also consider an equivalence relation $\equiv$ that captures equality on states. Integer expressions are interpreted in a particular state following the standard mathematical meaning by a function $[\![e]\!] : \mathit{State} \to \mathrm{Z}$. The interpretation of division is totalised (division by 0 returns 0), and boolean operations return 0 or 1 (for *false* and *true*). We take the big-step semantics of a program as its *natural specification*. For states σ and σ' we define:

$$\begin{aligned}
\mathsf{spec}_{\mathbf{skip}}(\sigma,\sigma') &= \sigma \equiv \sigma' \\
\mathsf{spec}_{C_1;C_2}(\sigma,\sigma') &= \exists \sigma'', \quad \mathsf{spec}_{C_1}(\sigma,\sigma'') \wedge \mathsf{spec}_{C_2}(\sigma'',\sigma') \\
\mathsf{spec}_{\mathtt{x:=}e}(\sigma,\sigma') &= \sigma' \equiv \sigma\{\mathtt{x} \leftarrow [\![e]\!](\sigma)\} \\
\mathsf{spec}_{\mathsf{a}[e_1]\mathtt{:=}e_2}(\sigma,\sigma') &= \sigma' \equiv \sigma\{\mathsf{a} \leftarrow \mathsf{upd}(\mathsf{a},[\![e_1]\!](\sigma),[\![e_2]\!](\sigma))\} \\
\mathsf{spec}_{\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2}(\sigma,\sigma') &= (([\![e]\!]\sigma \neq 0) \wedge \mathsf{spec}_{C_1}(\sigma,\sigma')) \vee (([\![e]\!]\sigma = 0) \wedge \mathsf{spec}_{C_2}(\sigma,\sigma')) \\
\mathsf{spec}_{\mathbf{while}\ (e)\ C}(\sigma,\sigma') &= \exists n,\ \mathsf{loop}^{n}_{e,\mathsf{spec}_C(\sigma,\sigma')}(\sigma,\sigma') \wedge ([\![e]\!](\sigma') = 0)
\end{aligned}$$

where the relation $\mathsf{loop}^{n}_{B,R}(\sigma,\sigma')$ denotes the loop specification for the body R under condition B and is inductively defined by

$$\begin{aligned}
\mathsf{loop}^{0}_{B,R}(\sigma,\sigma') &\Longleftarrow \sigma \equiv \sigma' \\
\mathsf{loop}^{S(n)}_{B,R}(\sigma,\sigma') &\Longleftarrow \exists \sigma'',\ \mathsf{loop}^{n}_{B,R}(\sigma,\sigma'') \wedge ([\![B]\!](\sigma'') \neq 0) \wedge R(\sigma'',\sigma')
\end{aligned}$$

This relation provides a natural choice for a loop's invariant; we thus call it the *natural invariant* for the loop. The definition makes explicit the *iteration rank* (iteration count) in superscript, as this is often convenient in the proofs (when omitted, it should be considered as existentially quantified). Subscripts will be omitted (both in loop and spec) when the corresponding programs are clear from the context. By construction, spec enjoys the following properties.

Lemma 1 ([4]). *Let $R(\sigma,\sigma')$ be a deterministic relation on states, and B a boolean condition. Then,* $\mathsf{loop}_{B,R}(\sigma,\sigma')$ *is deterministic whenever* $[\![B]\!](\sigma') \neq 0$*, i.e.*

loop synchronisation: $\forall n_1\ n_2\ \sigma_1\ \sigma_2\ \sigma'_1\ \sigma'_2,$
$\sigma_1 \equiv \sigma_2 \wedge \mathsf{loop}^{n_1}_{B,R}(\sigma_1,\sigma'_1) \wedge ([\![B]\!](\sigma'_1) = 0) \wedge \mathsf{loop}^{n_2}_{B,R}(\sigma_2,\sigma'_2) \wedge ([\![B]\!](\sigma'_2) = 0) \Longrightarrow n_1 = n_2;$

loop determinism: $\forall n\ \sigma_1\ \sigma_2\ \sigma'_1\ \sigma'_2,$
$\sigma_1 \equiv \sigma_2 \wedge \mathsf{loop}^{n}_{B,R}(\sigma,\sigma'_1) \wedge \mathsf{loop}^{n}_{B,R}(\sigma,\sigma'_2) \Longrightarrow \sigma'_1 \equiv \sigma'_2.$

Our strategy for reasoning about self-composition proof goals is based on identifying a set of general lemmas that can be proven once-and-for-all, and then included in the annotations provided to the verification platform, allowing other proof obligations to be automatically discharged.

2.3. Self-composition Lemmas

The determinism property is not sufficient to reason about a non-interference property by self-composition: it merely states that the two instances of the program will produce the same outputs when all of their inputs are equal. What is needed is a rephrasing of that property using an equality relation on low-security variables. If the control structure of a cycle does not depend on high-security variables, the determinism property proof can be carried over to non-interference lemmas. More explicitly, we recast each loop synchronisation lemma as follows

$$\forall n_1\ n_2\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2',\quad \pi^B(\sigma_1) \equiv \pi^B(\sigma_2) \wedge \mathsf{loop}^{n_1}_{B,R}(\sigma_1,\sigma_1')$$
$$\wedge\,(\llbracket B\rrbracket(\sigma_1') = 0) \wedge \mathsf{loop}^{n_2}_{B,R}(\sigma_2,\sigma_2') \wedge (\llbracket B\rrbracket(\sigma_2') = 0) \Longrightarrow n_1 = n_2$$

where π^B projects the fragment of the state that influences the control structure (i.e. the loop conditions) – note that this can be obtained by a simple dependency analysis. A non-interference result for each loop follows easily from non-interference in its body:

$$(\forall \sigma_1,\sigma_2,\sigma_1',\sigma_2',\ \sigma_1 \equiv_L \sigma_2 \wedge R(\sigma_1,\sigma_1') \wedge R(\sigma_2,\sigma_2') \Rightarrow \sigma_1' \equiv_L \sigma_2')$$
$$\Rightarrow\quad \forall \sigma_1,\sigma_2,\sigma_1',\sigma_2',\ \sigma_1 \equiv_L \sigma_2 \wedge \mathsf{loop}^{n_1}_{B,R}(\sigma_1,\sigma_1') \wedge (\llbracket B\rrbracket(\sigma_1') = 0)$$
$$\wedge\, \mathsf{loop}^{n_2}_{B,R}(\sigma_2,\sigma_2') \wedge (\llbracket B\rrbracket(\sigma_2') = 0) \Rightarrow \sigma_1' \equiv_L \sigma_2'$$

Observe that proving non-interference for loop-free programs by self-composition can be easily verified by automatic provers. The precondition for this lemma can then be seen as an additional proof-obligation that must be discharged.

2.4. *Verification infrastructure*

In this work we used `Frama-C` [8], a tool for the static analysis of `C` programs, annotated using the ANSI-C Specification Language (ACSL [8]), that contains a multi-prover verification condition generator [14]. `Frama-C` also contains the `gwhy` graphical front-end that allows to monitor individual verification conditions. This is particularly useful when combined with the possibility of using various proof tools, which allows users to first try discharging conditions with one or more automatic provers, leaving the harder conditions to be proved with an interactive proof assistant. We used the Boron release of `Frama-C`. We also employed a set of proof tools that included the Coq proof assistant, and the `Simplify`, `Alt-Ergo`, and `Z3` automatic theorem provers.

A feature of `Frama-C` that was crucial for the work reported here is the declaration of Lemmas. Lemmas resemble axioms in that they can be used to prove verification conditions. The difference is that lemmas originate themselves new goals to be proved. A key idea in the proofs we developed is that once an appropriate lemma has been proved interactively (with Coq) and included in the specification, all other verification conditions can be automatically discharged. The Coq library described in [4] provides support for proving lemmas such as those introduced in the previous subsection. As a rule, this library embeds each lemma and respective proof in a functor parameterised by the basic facts it depends on. All the results needed as inputs for the functors are non-recursive (they concern the loop body only) and can be expected to be proved successfully by an automatic prover.

2.5. *An example*

Consider the code extracted from the the NaCl library presented in Listing 1. Function `select` combines the input values of `r`, `s` and `b` to compute the final values of `p` and `q`. By inspecting the code we can conclude that the final value of `p` does not depend on the value of `q`. This property can be thought of as a noninterference property for confidentiality. So considering, for example, that `p` has low security (L) and `q` has high security (H), we can use self-composition to prove this property.

```
static void select(unsigned int p[64], unsigned int q[64],
        const unsigned int r[64], const unsigned int s[64], unsigned int b) {
        unsigned int j; unsigned int t; unsigned int bminus1;
        bminus1 = b - 1;
        for (j = 0; j < 64; ++j) { t = bminus1 & (r[j] ^ s[j]);
                                   p[j] = s[j] ^ t; q[j] = r[j] ^ t; }
}
```

Listing 1: select function extracted from NaCl core library

```
/*@ predicate loop_body{L1,L2}(integer j1, integer j2, unsigned int *p,
 @    unsigned int *q, unsigned int *r, unsigned int *s, unsigned int bminus) =
 @    \exists unsigned int t;  j2 == j1 + 1 &&
 @    t == (bminus & (\at(r[j1],L2) ^ \at(s[j1],L2))) &&
 @    \at(p[j1],L2) == (\at(s[j1],L2) ^ t) && \at(q[j1],L2) == (\at(r[j1],L2)^t);
 @*/

/*@ inductive loop_predicate{L1,L2}(integer j1, integer j2,
 @      unsigned int *p, unsigned int *q, unsigned int *s, unsigned int *r,
 @  unsigned int bminus){
 @   case base_case{L}:
 @    \forall unsigned int *p,*q,*s,*r,bminus; \forall integer j;
 @     loop_predicate{L,L}(j,j,p,q,s,r,bminus);
 @   case ind_case{L1,L2,L3}:
 @     \forall integer j1, j2, j3; \forall unsigned int *p,*q,*s,*r,bminus;
 @        loop_predicate{L1,L2}(j1,j2,p,q,r,s,bminus) ==>
 @        loop_body{L2,L3}(j2,j3,p,q,r,s,bminus) ==>
 @        loop_predicate{L1,L3}(j1,j3,p,q,r,s,bminus);
 @ }
 @*/
```

Listing 2: natural invariant

The first step is to create a program corresponding to the composition of the original program with its renamed copy. The pre-condition of the resulting program must establish that all input parameters except q are equal, as they correspond to non-high-security inputs; and the post-condition must establish that the final values of the low security output p are also equal, and hence unaffected by free ranging high security input values. Because the code includes a for loop statement, we have to define a loop invariant capturing how the variables change during the loop execution. The natural invariant for the loop is defined as an inductive predicate with a base case and an inductive case. Its specification in ACSL can be found in Listing 2. The predicate only refers to variables handled by the loop. The base case corresponds to the loop initialization and the inductive case relies on the definition of a logical predicate which expresses how the loop variables are related between two successive iterations. The predicate definitions include the notion of state that is introduced by the labels which appear between curly brackets. Notice that, due to the specific characteristics of Frama-C, the explicit inclusion of states in the predicates is only necessary when the variables involved in the loop are pointers or arrays. For integers, for example, the predicate explicitly includes two references of the same variable in different states. To include the loop invariant as an annotation of the source code, we just have to instantiate the inductive predicate described above with the current values of the loop.

```
/*@ loop invariant 0<=j<=64 && loop_predicate{Pre,Here}(0,j,p,q,r,s,bminus1);
/*@ predicate ext_eq{L1,L2}(unsigned int *p1, unsigned int *p2) =
 @    \forall integer i; \at(p1[i],L1)==\at(p2[i],L2);
 @*/

/*@ lemma eq_loop_pred{L1,L2,L3,L4}:
 @    \forall integer j1,j2,j3; \forall unsigned int *p,*q,*r,*s,b;
 @    \forall unsigned int *p1,*q1,*r1,*s1,b1;
 @    ext_eq{L1,L3}(p,p1) ==> ext_eq{L1,L3}(r,r1) ==> ext_eq{L1,L3}(s,s1) ==>
 @    loop_predicate{L1,L2}(j1,j2,p,q,r,s,b) ==>
 @    loop_predicate{L3,L4}(j1,j3,p1,q1,r1,s1,b1) ==>
 @    j2>= 64 ==> j3>=64 ==> ext_eq{L2,L4}(p,p1);
 @*/
```

Listing 3: self-composition lemma

```
  @ loop variant 64 j;
  @*/
   for (j = 0; j < 64; ++j)
      { t = bminus1 & (r[j] ^ s[j]);  p[j] = s[j] ^ t; q[j] = r[j] ^ t; }
```

Finally, the non-trivial part of the proof is isolated in the form of a self-composition lemma, as described in Section 2.3. This simply expresses that executing the loop starting from equivalent states in the non-high security values will lead to a state that is equivalent over the low-security variables. The definition of such a lemma in ACSL can be found in Listing 3. The predicate ext_eq is used for convenience in defining extensional array equality and the lemma itself expresses the property we want to prove.

Hoping for an automatic proof of the lemma would be too ambitious. However, an interactive proof in Coq can be easily done using the Coq library described in [4]. The instantiation of this lemma in the Coq library is accomplished by invoking an appropriate functor. In this particular case, the functor builds the inductive definition of the loop and derives the corresponding lemma. It is parameterized by two modules describing the loop state, which corresponds to the partition that affects the loop condition, and the specification of the loop body. The latter corresponds to the definition of the predicate loop_body defined in Listing 2. Firstly we need to prove a simple lemma which is related to a single execution of the loop body and that is used as input of the functor, stating that the loop body preserves the desired non-interference property. The proof of the self-composition lemma then follows directly from the functor instantiation. Furthermore, note that the proof of the simple theorem concerning the loop body could, itself, be automated. One could generate the corresponding theorem using the ACSL notation, and use an automatic prover to discharge the associated proof obligation.

The proposed methodology relies on a considerable amount of code annotation. But this effort, including natural invariant generation and the corresponding lemmas, is amenable to be automatically generated.

3. Formalisation and Verification of Side Channel Countermeasures

In this section we illustrate how the framework of the previous section can be used to attest adherence to non-functional security policies. We start by explaining in Section 3.1 how the security policies put forward by the developers of the NaCl library can be understood semantically as a non-interference property, that cannot be expressed using a standard semantics. In Section 3.2 we then instrument the semantics of the language (adding memory and control-flow traces) and use it in Section 3.3 to faithfully capture the policies under scrutiny. Section 3.4 applies a simple program transformation to reify the instrumented semantics (by internalising trace information in the programs), which allows expressing security as a standard non-interference property.

3.1. Security Policy as a Semantic Property

Consider a standard big-step operational semantics of the programming language, and let $(C,\sigma) \Downarrow \sigma'$ denote the fact that the program C executed in state σ terminates in state σ'. The NaCl security policy can be expressed as a non-interference-like property based on this semantics. Let C be a program, H a set of high-security variables and $V_L^I = Vars \setminus H$. Then informally, C complies with the NaCl side-channel security policies if

For any two states σ_1, σ_2 such that $\sigma_1 \overset{V_L^I}{=} \sigma_2$, if $(C,\sigma_1) \Downarrow \sigma_1'$ then for some state σ_2' one has that $(C,\sigma_2) \Downarrow \sigma_2'$, *with the same memory trace and control flow for both executions*.

In other words, the memory positions accessed and the execution paths followed are equal for both initial states. This clearly ensures that the control flow and array lookups do not depend on secret information, as prescribed. Naturally, a plain state-based semantics does not allow expressing this property formally (since no trace information is manipulated), which motivates the introduction of an extended instrumented semantics.

3.2. Instrumented Semantics

We consider two additions to the language used before. Firstly, all commands except sequential composition are now labelled. This is equivalent to labeling every atomic statement and every boolean condition. We further assume that all considered programs are *well-labelled*, meaning that all the labels in a program are distinct. Labels can then be thought of as abstractions of the instruction-pointer to the corresponding code. Secondly, a new syntactic class of list-expressions is considered (together with the corresponding variables and assignment statements). Such lists are useless for programming, but they are convenient to capture the NaCl policies under a standard non-interference formulation, so we include them in the language and treat them consistently with the other constructions. Furthermore, our implementation in `Frama-C` of such lists is natural and consistent with this formalisation (see Section 4).

The syntax of the extended language is given as follows.

Operators	op ::=	`+` | `-` | `*` | `/` | `=` | `!=` | `<`
Expressions	e ::=	$\mathbf{n}$ | `x` | $e\ op\ e$ | `a[`e`]`
List expressions	le ::=	`nil` | `cons(`e, le`)`
Commands	C ::=	$[\mathbf{skip}]^l$ | $[\mathtt{x:=}e]^l$ | $[\mathtt{a[}e\mathtt{]:=}e]^l$ | $[\mathtt{xl:=}le]^l$
		| $[\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]^l$ | $[\mathbf{while}\ (e)\ C]^l$ | $C_1\ ;\ \ C_2$

We will use the notation $\mathsf{stmt}^C(l)$ to refer to the statement annotated with label l in program C (recall that labels are assumed to be distinct). Moreover, we remark that by construction every program should use a non-empty set of labels. We denote the leftmost label used in a program C by $\mathsf{firstLabel}(C)$.

$$\frac{}{(\mathbf{n},\sigma) \Downarrow^{e} (\mathbf{n},\varepsilon)} \qquad \frac{}{(\mathtt{x},\sigma) \Downarrow^{e} (\sigma(\mathtt{x}),(\mathtt{x},0))} \qquad \frac{(e,\sigma) \Downarrow^{e} (v,\gamma)}{(\mathtt{a[}e\mathtt{]},\sigma) \Downarrow^{e} (\mathrm{acc}(\sigma(\mathtt{a}),v),(\mathtt{a},v)\cdot\gamma)}$$

$$\frac{(e_1,\sigma) \Downarrow^{e} (v_1,\gamma_1) \qquad (e_2,\sigma) \Downarrow^{e} (v_2,\gamma_2)}{(e_1 \; op \; e_2,\sigma) \Downarrow^{e} (v_1 \; [\![op]\!] \; v_2,\gamma_1\cdot\gamma_2)}$$

$$\frac{}{(\mathtt{nil},\sigma) \Downarrow^{e} (\mathtt{nil},\varepsilon)} \qquad \frac{(e,\sigma) \Downarrow^{e} (v,\gamma_1) \qquad (le,\sigma) \Downarrow^{e} (lv,\gamma_2)}{(\mathtt{cons}(e,le),\sigma) \Downarrow^{e} (\mathtt{cons}(v,lv),\gamma_1\cdot\gamma_2)}$$

$$\frac{}{([\mathbf{skip}]^{l},\sigma) \Downarrow (\sigma,l,\varepsilon)} \qquad \frac{(e_1,\sigma) \Downarrow^{e} (v_1,\gamma_1) \qquad (e_2,\sigma) \Downarrow^{e} (v_2,\gamma_2)}{([\mathtt{a[}e_1\mathtt{]:=}e_2]^{l},\sigma) \Downarrow (\sigma[\mathtt{a} \leftarrow \mathrm{upd}(\sigma(\mathtt{a}),v_1,v_2)],l,(\mathtt{a},v_1)\cdot\gamma_1\cdot\gamma_2)}$$

$$\frac{(e,\sigma) \Downarrow^{e} (v,\gamma)}{([\mathtt{x:=}e]^{l},\sigma) \Downarrow (\sigma[\mathtt{x} \leftarrow v],l,(\mathtt{x},0)\cdot\gamma)} \qquad \frac{(le,\sigma) \Downarrow^{e} (lv,\gamma)}{([\mathtt{xl:=}le]^{l},\sigma) \Downarrow (\sigma[\mathtt{xl} \leftarrow lv],l,(\mathtt{xl},0)\cdot\gamma)}$$

$$\frac{(e,\sigma) \Downarrow^{e} (v,\gamma) \qquad (C_1,\sigma) \Downarrow (\sigma_1,\delta_1,\gamma_1)}{([\mathbf{if}\; e \;\mathbf{then}\; C_1 \;\mathbf{else}\; C_2]^{l},\sigma) \Downarrow (\sigma_1,l\cdot\delta_1,\gamma\cdot\gamma_1)} \; \textit{if } v \neq 0$$

$$\frac{(e,\sigma) \Downarrow^{e} (v,\gamma) \qquad (C_2,\sigma) \Downarrow (\sigma_2,\delta_2,\gamma_2)}{([\mathbf{if}\; e \;\mathbf{then}\; C_1 \;\mathbf{else}\; C_2]^{l},\sigma) \Downarrow (\sigma_2,l\cdot\delta_2,\gamma\cdot\gamma_2)} \; \textit{if } v = 0$$

$$\frac{(e,\sigma) \Downarrow^{e} (v,\gamma) \qquad (C,\sigma) \Downarrow (\sigma_1,\delta_1,\gamma_1) \qquad ([\mathbf{while}\;(e)\;C]^{l},\sigma_1) \Downarrow (\sigma_2,\delta_2,\gamma_2)}{([\mathbf{while}\;(e)\;C]^{l},\sigma) \Downarrow (\sigma_2,l\cdot\delta_1\cdot\delta_2,\gamma\cdot\gamma_1\cdot\gamma_2)} \; \textit{if } v \neq 0$$

$$\frac{(e,\sigma) \Downarrow^{e} (v,\gamma)}{([\mathbf{while}\;(e)\;C]^{l},\sigma) \Downarrow (\sigma,l,\gamma)} \; \textit{if } v = 0 \qquad \frac{(C_1,\sigma) \Downarrow (\sigma_1,\delta_1,\gamma_1) \qquad (C_2,\sigma_1) \Downarrow (\sigma_2,\delta_2,\gamma_2)}{(C_1;C_2,\sigma) \Downarrow (\sigma_2,\delta_1\cdot\delta_2,\gamma_1\cdot\gamma_2)}$$

Figure 2: Evaluation semantics

To capture the memory locations accessed during the execution of a program, the operational semantics is instrumented in order to keep track of the sequence of performed accesses – the *memory trace*, ranged by γ. Each element of the memory trace consists of a pair $(\mathtt{v},\textit{offset})$ where $\mathtt{v}$ is the variable identifier and *offset* is the index of the accessed memory location (0 for non-array variables). The control-flow is also made explicit by computing the sequence of labels executed during the computation — the *control-flow trace*, ranged by δ. We will then consider judgements of the form $(C,\sigma) \Downarrow (\sigma',\delta,\gamma)$ meaning that program C executed in state σ terminates in state σ', having followed the control-flow path δ and performed memory accesses γ. An auxiliary judgment is used for expressions: $(e,\sigma) \Downarrow^{e} (\mathbf{n},\gamma)$ means that expression e evaluated in state σ returns the value $\mathbf{n}$, having performed accesses γ. When the traces in the final configuration are not important they will be omitted as in $(C,\sigma) \Downarrow \sigma'$. Figure 2 presents the big-step rules for both expressions and programs, where ε denotes the empty sequence, $\cdot$ denotes concatenation of sequences, and the singleton sequence is identified with its element (e.g. $l\cdot\delta$ denotes the addition of l in front of δ).

We now state a few useful lemmas (proofs can be found in Appendix A). A first observation is that the control-flow trace constrains significantly the memory access trace of any given program. If an execution path is fixed, only indices for array accesses are allowed to vary. Let us denote by $\mathsf{projFst}(\gamma)$ the function that projects the first component of a memory trace γ, returning a list of variable identifiers.

Lemma 2. *Let C be a program, e an expression, and σ_1, σ_2 states.*

1. *If $(e,\sigma_1) \Downarrow^e (v_1,\gamma_1)$ and $(e,\sigma_2) \Downarrow^e (v_2,\gamma_2)$, then $\mathsf{projFst}(\gamma_1) = \mathsf{projFst}(\gamma_2)$.*

2. *If $(C,\sigma_1) \Downarrow (\sigma_1', \delta, \gamma_1)$, $(C,\sigma_2) \Downarrow (\sigma_2', \delta, \gamma_2)$, then $\mathsf{projFst}(\gamma_1) = \mathsf{projFst}(\gamma_2)$.*

Another way of looking at the previous lemma is to state that the differences between two memory traces γ_1, γ_2 obtained through the same execution path concern only the sequences of indexes accessed in one or more arrays. Denoting by $\mathsf{projArr}^{\mathsf{a}}(\gamma)$ the function that returns the list of indexes accessed in an array a, we have:

Lemma 3. *Let C be a program such that $(C,\sigma_1) \Downarrow (\sigma_1', \delta, \gamma_1)$ and $(C,\sigma_2) \Downarrow (\sigma_2', \delta, \gamma_2)$. Then, $\gamma_1 = \gamma_2$ if and only if for all array variables $\mathtt{a}$ in C, $\mathsf{projArr}^{\mathtt{a}}(\gamma_1) = \mathsf{projArr}^{\mathtt{a}}(\gamma_2)$.*

Control-flow traces are also severely constrained: there are specific points where different executions may diverge, which correspond exactly to the boolean conditions tests performed by the program (*if* and *while* statements).

Lemma 4. *Let C be a program such that $(C,\sigma_1) \Downarrow (\sigma_1', \delta_1, \gamma_1)$ and $(C,\sigma_2) \Downarrow (\sigma_2', \delta_2, \gamma_2)$. Then, $\delta_1 = \delta_2$ if and only if $\mathsf{tests}^C(\delta_1) = \mathsf{tests}^C(\delta_2)$.*

Unction $\mathsf{tests}^C(\cdot)$ extracts the outcomes of these tests from a given execution trace.

$$\mathsf{tests}^C(\varepsilon) = \varepsilon$$

$$\mathsf{tests}^C(l \cdot \delta) = \begin{cases} \mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) \text{ is not an } \textit{if} \text{ nor a } \textit{while} \\ 1 \cdot \mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) = [\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2]^l, \\ & \delta = l' \cdot \delta'' \text{ and } l' = \mathsf{firstLabel}(C_1) \\ 0 \cdot \mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) = [\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2]^l, \\ & \delta = l' \cdot \delta'' \text{ and } l' = \mathsf{firstLabel}(C_2) \\ 1 \cdot \mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) = [\textbf{while } (e)\ C]^l, \\ & \delta = l' \cdot \delta' \text{ and } l' = \mathsf{firstLabel}(C) \\ 0 \cdot \mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) = [\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2]^l \\ & \text{and either } \delta = \varepsilon, \text{ or } \delta = l' \cdot \delta' \text{ and } l' \neq \mathsf{firstLabel}(C) \end{cases}$$

3.3. Formal Security Definition

The NaCl side-channel security policies (Figure 1) can now be expressed as a non-interference-like property.

Definition 5. *Let C be a program, H high-security variables and $V_L' = Vars \setminus H$. We say that C is NaCl-secure if*

$$\sigma_1 \overset{V_L'}{=} \sigma_2 \wedge (C,\sigma_1) \Downarrow (\sigma_1', \delta_1, \gamma_1) \implies$$
$$\textit{For some } \sigma_2', \delta_2, \textit{and } \gamma_2, \quad (C,\sigma_2) \Downarrow (\sigma_2', \delta_2, \gamma_2) \wedge (\delta_1 = \delta_2 \wedge \gamma_1 = \gamma_2).$$

A weaker termination-insensitive variant is also considered, namely

$$\sigma_1 \overset{V_L'}{=} \sigma_2 \wedge (C,\sigma_1) \Downarrow (\sigma_1', \delta_1, \gamma_1) \wedge (C,\sigma_2) \Downarrow (\sigma_2', \delta_2, \gamma_2) \implies (\delta_1 = \delta_2 \wedge \gamma_1 = \gamma_2).$$

Analogously, an expression e is said to be NaCl-secure if

$$\sigma_1 \overset{V_L'}{=} \sigma_2 \wedge (e,\sigma_1) \Downarrow^e (v_1,\gamma_1) \implies \textit{for some } v_2 \textit{ and } \delta_2, \quad (e,\sigma_2) \Downarrow^e (v_2,\gamma_2) \wedge \gamma_1 = \gamma_2.$$

The following proposition captures a convenient compositional property of our security notion.

Proposition 6 (Compositionality). *Let C_1 and C_2 be NaCl-secure programs, and let e_1 and e_2 be NaCl-secure expressions. Then,*

- *e_1 op e_2, and $a[e_1]$ are NaCl-secure expressions;*
- *$C_1;C_2$, $[\textbf{\textit{while}}\ (e_1)\ C_1]^l$ and $[\textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ C_1\ \textbf{else}\ C_2]^l$ are NaCl-secure programs;*

PROOF. By structural induction on expressions and programs.

The above property has implications on both the scalability and modularity of our techniques. We rely on it to conduct the formal verification exercise in a gradual way, starting from leaf functions, and tackling each function independently. This allows us to tame the complexity of each verification step and combine the results to obtain a global security guarantee. Furthermore, the results one obtains for a verified component (such as the NaCl library) are established once-and-for-all, and can be reused as an intermediate result in subsequent verification exercises, e.g. verifying different client applications that may come to use the NaCl library.

3.4. Verification of Security

Although Definition 5 nicely captures the NaCl side-channel security policies, it is not a convenient formalization for our verification purposes: we aim to apply self-composition, and so we require a specification that expresses security directly over the program state. To this end, we now introduce a program transformation that internalises into the program state sufficient information from the instrumented semantics. The transformed programs explicitly manipulate control-flow and memory access trace information.

Figure 3 contains the definition of the transformation $\langle\cdot\rangle$ for both expressions and programs. The transformation makes use of fresh list variables `control` and $\texttt{xl}^a$ (for each array variable `a`). Informally, given an expression e and a command C, $\langle e\rangle$ is a program that stores the indexes of arrays accessed during the evaluation of e (in the corresponding variables $\texttt{xl}^a$), and $\langle C\rangle$ is similar to C but also keeps track of all conditional tests performed and of all array access indexes (in variables `control` and $\texttt{xl}^a$). The following proposition relates in precise terms the final values of these variables of the transformed program, and the memory and execution traces of the original.

Proposition 7. *Let C be a program such that $(C,\sigma) \Downarrow (\sigma',\delta',\gamma')$. Consider moreover that $\overline{\sigma}^0$ is the environment that assigns to variable `control` and xl^a (for every array variable `a` in C) the empty sequence ε. Then, $(\langle C\rangle, \sigma \uplus \overline{\sigma}^0) \Downarrow \overline{\sigma}$, where:*

- $\overline{\sigma} = \sigma' \uplus \overline{\sigma}'$, *with* $\mathsf{dom}(\overline{\sigma}^0) = \mathsf{dom}(\overline{\sigma}')$,
- $\overline{\sigma}'(\mathit{control}) = \mathsf{tests}^C(\delta')$,
- $\overline{\sigma}'(xl^a) = \mathsf{projArr}^a(\gamma')$.

$$\begin{aligned}
\langle \mathbf{n} \rangle = \langle \texttt{nil} \rangle = \langle \texttt{x} \rangle = \langle \texttt{xl} \rangle &= [\mathbf{skip}]^{l} && (l \text{ a fresh label})\\
\langle \texttt{a[}e\texttt{]} \rangle &= \langle e \rangle \,;[\texttt{xl}^{a}\texttt{:=}\texttt{cons}(e,\texttt{xl}^{a})]^{l} && (l \text{ a fresh label})\\
\langle e_1 \; op \; e_2 \rangle &= \langle e_1 \rangle \,;\langle e_2 \rangle\\
\langle \texttt{cons}(e,le) \rangle &= \langle e \rangle \,;\langle le \rangle\\
\\
\left\langle [\mathbf{skip}]^{l} \right\rangle &= [\mathbf{skip}]^{l}\\
\left\langle [\texttt{x:=}e]^{l} \right\rangle &= \langle e \rangle \,;[\texttt{x:=}e]^{l}\\
\left\langle [\texttt{xl:=}le]^{l} \right\rangle &= \langle le \rangle \,;[\texttt{xl:=}le]^{l}\\
\left\langle [\texttt{a[}e_1\texttt{]:=}e_2]^{l} \right\rangle &= \langle e_1 \rangle \,;\langle e_2 \rangle \,;[\texttt{xl}^{a}\texttt{:=}\texttt{cons}(e_1,\texttt{xl}^{a})]^{l'}\,;[\texttt{a[}e_1\texttt{]:=}e_2]^{l} && (l' \text{ a fresh label})\\
\left\langle [\mathbf{if}\; e \;\mathbf{then}\; C_1 \;\mathbf{else}\; C_2]^{l} \right\rangle &= \langle e \rangle \,;[\texttt{control:=cons}((e\;\texttt{!=}\;\texttt{0}),\texttt{control})]^{l'}\,;\\
&\quad\; [\mathbf{if}\; e \;\mathbf{then}\; \langle C_1 \rangle \;\mathbf{else}\; \langle C_2 \rangle]^{l} && (l' \text{ a fresh label})\\
\left\langle [\mathbf{while}\;(e)\; C]^{l} \right\rangle &= \langle e \rangle \,;[\texttt{control:=cons}((e\;\texttt{!=}\;\texttt{0}),\texttt{control})]^{l'}\,;\\
&\quad\; \Big[\mathbf{while}\;(e)\;\langle C \rangle \,;\langle e \rangle \,;[\texttt{control:=cons}(e,\texttt{control})]^{l'_1}\Big]^{l} && (l', l'_1 \text{ fresh labels})\\
\langle C_1 ; C_2 \rangle &= \langle C_1 \rangle \,;\langle C_2 \rangle
\end{aligned}$$

Figure 3: Transformation for internalising trace information

PROOF. By structural induction on the derivation of $(C,\sigma) \Downarrow (\sigma',\delta,\gamma)$. It is clear from the definition of the transformation that the inserted code only affects variables introduced by it, hence the partition of the final state is immediate. Moreover, every conditional test performed during the execution is explicitly stored in variable `control` (notice that, for the case of *while* loops, the transformation inserts code before the loop and at the end of the loop body). Finally, every evaluated expression of the original program is preceded by the execution of the transformation of that same expression.

Theorem 8. *Let C be a program, H high-security variables, $\langle V \rangle$ the set of variables introduced by transforming C to $\langle C \rangle$, and $\langle V'_L \rangle = Vars(C) \setminus H \,\cup\, \langle V \rangle$. The program C is (termination-insensitive) secure with respect to Definition 5 if for states σ_1, σ_2,*

$$\sigma_1 \overset{\langle V'_L \rangle}{=} \sigma_2 \,\wedge\, (\langle C \rangle, \sigma_1) \Downarrow \sigma'_1 \,\wedge\, (\langle C \rangle, \sigma_2) \Downarrow \sigma'_2 \implies \sigma'_1 \overset{\langle V \rangle}{=} \sigma'_2$$

PROOF. Follows directly from Proposition 7 and Lemmas 3 and 4.

The formulation given by Theorem 8 can be readily verified by the self-composition technique, as explained in Section 2. A similar result could be derived for the termination-sensitive variant of security, but that would not be directly usable with self-composition. In our approach we separately handle the proof of termination, which together with the previous result trivially yields the termination-sensitive variant.

```
int crypto_verify(const unsigned char *x, const unsigned char *y)
{
   int differentbits = 0, i = 0;

   while (i < 16) { differentbits |= x[i] ^ y[i]; i++; }
   return (1 & ((differentbits - 1) >> 8)) - 1;
}
```

Listing 4: NaCl implementation of crypto_verify function

4. Case Study: NaCl Cryptographic Library

The high-level security policies adopted in the implementation of the NaCl cryptographic library, which serve as motivation for this work, were introduced in Section 1 and formalized in the previous section. We now present examples of how the techniques proposed in this paper can be used in practice to formally verify compliance to these policies, using off-the-shelf verification tools. We selected two additional examples from the core of the NaCl library, aiming to highlight various aspects of our contributions. We begin with a simpler one, which we can describe in more detail to adequately illustrate the practical implementations aspects of our work. We then move on to discuss a more complex example to further justify our contributions. Overall, we have successfully applied these techniques to the formal verification of all of the core functions in the NaCl library (aprox. 560 loc). Nevertheless, and even though we argue that most of the annotation work required to carry out the exercise can be automated, we have manually annotated the programs. The discharge of the resulting verification conditions, with the exception of the loop-related lemmata that we explicitly factor out in the self-composition proofs, was fully handled by automatic provers.

4.1. A simple example

The selected function[1] is called crypto_verify and is presented in Listing 4. It may be surprising to know that the high-level specification for this function is that it compares the contents of two 16-byte arrays x and y, whose contents are high-security and must not be leaked. The introduced optimizations aim to ensure both control flow and data memory access independence, as prescribed by the NaCl security policies. As a side note, we remark that we have also verified that this function is functionally correct with respect to a (readable) reference implementation, using the methodology proposed in [4, 3]. We do not include the details in this paper due to space constraints.

As explained at the end of the previous Section, we establish (termination-insensitive) security by splitting our formal verification exercise in two independent steps. The first step is to verify safety (and termination) for all valid inputs. The second step is to apply the program reification and formal verification tasks that permit applying Theorem 8 and establishing that the program is indeed secure according to Definition 5.

[1]The actual implementation in the NaCl library totally unfolds the while loop, but this would not be as convenient for ilustrative purposes.

Safety and termination verification. This step can be easily achieved in `Frama-C` by annotating the code with appropriate pre-conditions, imposing the validity of input arrays in the proper range, and adding some simple lemmas that allow the tool to recognize the correct output range of the bit-wise operations used. These lemmas are required because a sufficiently expressive axiomatic semantics for these operations is typically not included in off-the-shelf formal verification tools such as `Frama-C`, since such operations are rarely used in general-purpose software.

Establishing (termination-insensitive) security. To apply Theorem 8, we establish security by first constructing a reified version of the program, and then performing a self-composition proof that it displays the required non-interference properties. The transformed program is created according to the rules described in Figure 3, and outputs a set of lists containing the relevant traces collected during the program's execution.

Recall that the list type introduced in the instrumented semantics of Section 3 is essentially an artifact to enable the application of our proof technique. They are not dynamic data structures offered by the underlying programming language, but rather constructions that may exist merely at the logical level. Furthermore, since the values of the constructed lists cannot influence the semantics of operations over other data types, they enable a more elegant formalisation and an easier justification of our theoretical results. Luckily, we can take advantage of a feature of `Frama-C` that enables the direct transposition of this logical data type onto code annotations: the ability to use *ghost code* in annotations enables us to include all the extra code introduced by our transformation as comments to the original program. Furthermore, using ghost code, we have the guarantee that the semantics of the original program are preserved, and cannot be affected by the values of said lists as required by our formalisation. This restriction is imposed as a necessary condition by the deductive verification tool.

In short, the fact that we do not require a concrete implementation of the list type is a central aspect to the practical side of our work. On one hand, it eliminates a potential gap between our theoretical and practical approaches. On the other hand, as noted in [25], if we could not adopt this strategy, the formal verification exercise would be rendered considerably more complex, and probably, out of reach of our framework.

In Listing 5 we show the result of applying the transformation in Figure 3 to the program in Listing 4. Note the declaration of C functions that allow the construction of the lists within ghost code. The semantics of these functions is axiomatised to capture the necessary list constructors. At the end of execution, the final state of the ghost list variables is essentially a logical term evidencing a sequence of `cons` operations. Our experience shows that this implementation is highly suitable for being passed down to automatic provers. To complete the verification exercise, we must establish that this reified program indeed displays the non-interference property specified in Theorem 8. Here we directly apply, in a black-box way, the approach to performing proofs by self-composition presented in Section 2 and proposed in [3, 4]. Therefore, to deal with the loop structures, we annotate the programs with natural invariants and the associated lemmata. In particular, in order to enable the automatic discharge of all proof obligations, the following lemma needs to be included:

```
/*@ lemma eq_loop_pred{L1,L2,L3,L4}:
  @ \forall int i1,i2,i3,diffbits1,diffbits2,unsigned char *x,*y,*x1,*y1;
```

```
/*@ axiomatic list{ type list;
  @              logic list null;
  @              logic list cons(integer n, list s); }                    */

/*@ ghost int mem_control, mem_x, mem_y;
  @ axiomatic lmem{ logic list lmem_control{L} reads mem_control;
  @              logic list lmem_x{L} reads mem_x;
  @              logic list lmem_y{L} reads mem_y; }                      */

/*@ assigns mem_control;
  @ ensures lmem_control{Here} == cons(condition,lmem_control{Pre});      */
void append_control(int condition);
/*@ assigns mem_x;
  @ ensures lmem_x{Here} == cons(x,lmem_x{Pre});                          */
void append_x(int x);
/*@ assigns mem_y;
  @ ensures lmem_y{Here} == cons(y,lmem_y{Pre});                          */
void append_y(int y);

void crypto_verify(const unsigned char *x, const unsigned char *y) {
   int differentbits = 0, i = 0;

   //@ ghost append_control(i<16);
   while (i < 16) {
      differentbits |= x[i] ^ y[i]; //@ ghost append_x(i); ghost append_y(i);
      i++;
      //@ ghost append_control(i<16);
   }
   return (1 & ((differentbits - 1) >> 8)) - 1;
}
```

Listing 5: Transformed version of crypto_verify function

```
  @ \forall list l1_x,l2_x,l1_y,l2_y, l1_control, l2_control,l1_x1, l2_x1;
  @ \forall list l1_y1, l2_y1,l1_control1,l2_control1;
  @   l1_x == l1_x1 ==> l1_y == l1_y1 ==> l1_control == l1_control1 ==>
  @   loop_pred{L1,L2}(i1, i2, x, y, 0, diffbits1, l1_x, l2_x,
  @                    l1_y, l2_y, l1_control, l2_control) ==>
  @   loop_pred{L3,L4}(i1, i3, x1, y1, 0, diffbits2, l1_x1, l2_x1,
  @                    l1_y1, l2_y1, l1_control1, l2_control1) ==>
  @   i2 == i3 ==> l2_x == l2_x1 && l2_y == l2_y1 && l2_control == l2_control1; */
```

The rest of the Frama-C input can be found in Appendix Appendix B. Note that the pre-conditions include only the necessary restrictions to complete the proof, and need not refer to all the non-high parts of the initial state. As stated above, the discharging of the proof obligations generated by this example, bar the lemma presented above, was handled without assistance by the automatic provers targeted by Frama-C. Furthermore, although we have manually added these annotations, we emphasise that *all* of the annotations required for this verification exercise could have been generated automatically by a tool implementing the specification described in Section 2.

The only caveat to the automation potential of this approach, which is highlighted by this example, resides therefore in the justification of self-composition lemmas such as that presented above. As explained in Section 2, we address this problem by relying on a Coq library [4] that can produce such a justification for a representative class of loop patterns that commonly arise in cryptographic software. This is accomplished by invoking the appropriate functor, which in this case essentially reduces the proof to establishing that the loop body preserves the required non-interference property. In [4]

```
static void mulmod(unsigned int h[17],const unsigned int r[17]) {
  unsigned int hr[17]; unsigned int i; unsigned int j; unsigned int u;
  for (i = 0;i < 17;++i) {
    u = 0;
    for (j = 0;j <= i;++j) u += h[j] * r[i - j];
    for (j = i + 1;j < 17;++j) u += 320 * h[j] * r[i + 17 - j];
    hr[i] = u;
  }
  for (i = 0;i < 17;++i) h[i] = hr[i];
  squeeze(h);
}
```

Listing 6: A snippet of the NaCl sources containing nested loops

we discuss the degree of automation than can also be introduced at this level.

4.2. A more challenging verification example

We now discuss how our techniques allow us to deal with a wider class of programs than previous approaches along similar lines [25, 26]. In particular, we show how we deal with programs with complex control structures, including nested loops, and also how we handle the verification of complete programs: self-contained components involving higher-level functions calling lower-level ones.

Listing 6 contains another snippet from the NaCl library implementation. This function carries out a specific modular multiplication operation. We have proved its adherence to the NaCl side-channel countermeasures using exactly the same approach as for the previous example. Intuitively, the natural invariant for the outer loop refers to the predicates specifying the natural invariants for the inner loops. All loop invariants refer to the contents of the trace lists in a simple way, which is made possible by our formalisation of these lists directly using the ACSL logical types. The end result is that the proof obligations for this more elaborate example are also discharged automatically by the `Frama-C` backend provers. As before, the self-composition lemmas must be discharged interactively, with the assistance of the Coq library. This stands in contrast with the work presented in [25], in which nested loops are excluded.

Another important point in verifying the function in Listing 6 is that it is not a leaf function: it calls auxiliary function `squeeze`, which in turn is a leaf function. To handle function calls in NaCl, and because we have not explicitly captured these language constructions in our formalisation, we slightly abuse the compositionality theorem for our theoretical framework presented in Section 3. In particular, we rely on the fact that the sequential composition of two secure programs is itself secure, and simply verify that all functions, independently, comply with the NaCl security policies. We argue that this is acceptable because of the following facts about the NaCl implementation, allow us to conclude that function calls in NaCl cannot, in themselves, introduce dependencies:

- It relies only on the `char` and `int` data types and arrays thereof, and uses no dynamic memory allocation.
- The relative addresses of all called functions are fixed at compile time.
- Parameter passing in the NaCl library is extremely conservative: all parameters are passed on a call-by-value basis with the exception of byte arrays.

- In NaCl, the base addresses of byte arrays passed by reference are all fully determined at compile time, with constant offsets relative to the start addresses of the memory regions that the caller itself received.

An alternative approach, which we have also implemented, permits formally verifying programs relying on a slightly more flexible parameter passing convention. In particular, we could exclude programs that introduce dependencies when passing the base address of a memory region to a callee function using an offset that depends on a sensitive value. This implies enhancing the reification of a caller function to incorporate in its output traces the start addresses of all memory regions passed by reference to the callees. However, we leave for future work the formalisation and a full description of the implementation of our techniques for these more complex use cases.

4.3. Discussion

By performing formal verification at the source code level, our solution is designed to respond to the concrete needs of cryptographers: we focus on existing security policies formulated over C source-code that are used in real-world applications. As a consequence of this, we are not formally addressing the gap between the guarantees provided by our results, and those that should hold at the machine code level. This is similar to what happens, not only with the currently used informal approach, but also with other formal verification methodologies.

This interesting problem can be addressed by relying on a compiler that is guaranteed to preserve the desired property, namely that the control-flow is independent of secret variables and that the indexed memory accesses follow the same pattern as in the program code. For the concrete case of the security policies and source-code that we address in this paper, we argue with high confidence that such a transferal from source to machine level results is indeed justifiable. This is due to a combination of factors: 1) the source-code is carefully written under coding policies that impose a canonical form for many C constructions, rendering programs with a very "clean" semantics; 2) we have treated short-circuit boolean operators as conditional expressions and these, in turn, were treated like conditional statements (i.e. the condition is added to the control-flow trace); 3) compilation is typically performed by excluding most compiler optimizations in order to ensure a predictable outcome.

We should also emphasize that, even though we believe our results show that our approach outperforms previous solutions in the deployment of self-composition proofs, there are still obvious limitations that should be highlighted. The first class of limitations are those inherent to the deductive verification technology itself. For example, for programs displaying high cyclomatic complexity [2], and despite the optimizations introduced by the existing tools, the number of generated proof obligations tends to increase exponentially. This means that formal verification rapidly becomes impractical. On the other hand, we should also highlight that NaCl code follows strict coding policies that make it *formal verification-friendly*. In particular, it does not use many of the features of the C language that typically complicate matters, including side-effects, pointer casts, or dynamic memory allocation.

[2] Intuitively, programs offering a large number of possible independent execution control-flow paths.

5. Related Work

A good survey of language-based information flow security can be found in [24]. Information flow policies were first introduced by Denning et. al [12] and tend to be formalised as non-interference properties. Information flow type systems have been used to enforce non-interference in different contexts [30, 21, 20, 27, 28]. The main challenge in designing these systems is that they are often too conservative in practice – secure programs may be rejected. Leino and Joshi [17] were the first to propose a semantic approach to secure information flow, with several desirable features: a precise characterisation of security; it applies to all programming constructs with well-defined semantics; it can be used to reason about indirect information leakage through variations in program behaviour (e.g. termination). An attempt to capture this property in program logics using the *Java Modelling Language* (JML) [16] was presented by Warnier et al. [31], who proposed an algorithm, based on strongest postconditions, that generates an annotated source file with specification patterns for confidentiality. Dufay et al. [13] have proposed an extension to JML to enforce non-interference through self-composition, allowing for a simple definition of non-interference for Java programs. However, the generated proof obligations are forbiddingly complex.

Terauchi and Aiken [26] identified problems in the self-composition approach, arguing that automatic tools (like software model checkers) are not powerful enough to verify this property over programs of realistic size. The authors propose a program transformation technique for an extended version of self-composition. Rather than replicating the original code, the renamed version is interleaved and partially merged with it. Naumann [22] extended Terauchi and Aiken's work to encompass heap objects, presented a systematic method to validate the transformations of [26], and reported on the experience of using these techniques with the Spec# and ESC/JAVA2 tools.

Natural invariants [4, 3], as we use them in this paper, provide an explicit rendition of program semantics. In [18] a similar encoding of program semantics in logical form can be found, which advocates the use of second-order logic as appropriate to reason about programs, since it allows to capture the inductive nature of the input-output relations for iterative programs. To some extent, our use of Coq's higher-order logic may be seen as an endorsement of that view. However, we have made an effort to combine this with facilities provided by automatic first-order provers.

Volpano and Smith [29] explored the use of type systems to protect programs against covert termination and timing channels. Specifically, their *timing agreement* theorem refers to a type-system that essentially captures our (termination-sensitive) notion of security (Definition 5). The distinction between both security notions relies on the fact that our definition, being defined by semantic means, is slightly more inclusive (e.g. it will allow a boolean condition such as $x \oplus x == 0$, with x a secret variable and $\oplus$ the *bitwise xor* operation, since it will not affect the control flow of the program). But admittedly, our main motivation for departing from the type-based approach was methodological, since we want to rely on the same set of deductive tools used in other verification tasks of the project. The remainder type-systems presented in [29], as well as consequent proposals (e.g. [2]) relax the constraint of predictable control-flow, hence failing to meet the requirements addressed in this paper.

The *Program Counter security model* (PC-model) proposed by D. Molnar et. al [19] captures the behavior of an attacker capable of observing the sequence of program counter positions during the execution of programs. These sequences are essentially what we have called control-flow traces, and hence their security definition coincides with our own restricted to the "no data-dependent branches" (i.e. ignoring the "no data-dependent array indices" constraint). The primary aim in [19] was not to check conformance of programs with a security property, but rather transform potentially insecure programs into secure ones. In particular, the authors were able to justify several established countermeasures found in the literature.

Svenningsson and Sands [25] have adopted the PC-model and addressed control-flow independence using self-composition. They also considered the issue of declassification, enabling the formal verification that only controlled amounts of leakage can occur (e.g. the leakage of the hamming weight of a secret during a modular exponentiation). Regarding the security notions, our work differs from this in two main aspects, motivated by the concrete real-world use case that we sought to formally verify. On one hand, we consider a more restrictive security notion where we also check for data memory access pattern independence. On the other hand, we do not consider declassification. Our approach to applying self-composition to a transformed version of the original program is close to [25]. However, not only do we present a full theoretical framework to justify our approach, but also and most importantly, our practical implementation approach allows us to go beyond the results reported in [25]. In particular, we have not restricted the class of accepted programs to the so-called *unnested* programs. That broader applicability scope, made possible by the use of natural invariants, was crucial to verify some of the functionality of the *Nacl* library (c.f. Section 4.2).

The security policies we have addressed in this paper can also be seen as integrity-preserving information-flow restrictions. Indeed, it is well known that one can see high variables as untrusted inputs, that (one wants to check) do not interfere with the control flow and addresses accessed by the program. Intuitively, one is showing that attackers manipulating these inputs cannot influence the behavior of the program. This sort of security policy is sometimes addressed through so-called *taint-analysis*. Static taint analysis techniques tend to be based on type systems [10] or on control-dependency graphs (CFG) [11]. Our work can be seen as an alternative approach to taint analysis.

6. Conclusion

We have shown how an off-the-shelf deductive verification platform can be used to validate real-world cryptographic software implementations, using the NaCl cryptographic library as a representative example. Our results focus on security-relevant properties: compliance to security policies aiming to reduce exposure to timing side-channel attacks, formalised as non-interference constraints.

Our approach to proving resistance to certain classes of side-channel attacks, namely timing attacks, extends previous work in several directions. Not only do we extend the range of attacks that were previously addressed, but we also show how reasonably automated verification can be made practical using off-the-shelf formal verification platforms. The general approach we adopt consists of reifying the target program to make explicit in its output the execution traces that may potentially leak information.

We reduce this explicit information to a minimum, proving that our approach is still sound, and then use non-interference and self-composition to verify security.

We have presented these new results as new application scenarios for the general methodology introduced in [3, 4], with promising results. In addition to showing that deductive verification methods are increasingly more amenable to practical use with reasonable degrees of automation, our work answers some open questions raised by previous work, which seemed to indicate that proofs by (self-)composition were not directly applicable in real-world situations, or at least not to sizable formal verification tasks. Our results are promising in that we have been able to achieve our goal using only off-the-shelf verification tools. We also believe that our technique has a high potential for mechanisation, and we aim to pursue this goal in future work.

References

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *ACM symposium on Information, computer and communications security*, ASIACCS '07, pages 312–320. ACM, 2007.

[2] Johan Agat. Transforming out timing leaks. In *In Proc. 27th ACM Symp. on Principles of Programming Languages (POPL*, pages 40–53. ACM Press, 2000.

[3] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Verifying cryptographic software correctness with respect to reference implementations. In *FMICS'09*, volume 5825 of *LNCS*, pages 37–52, 2009.

[4] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Deductive verification of cryptographic software. *ISSE*, 6(3):203–218, 2010.

[5] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.

[6] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS'04*, pages 49–69. Springer, 2004.

[7] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE, 2004.

[8] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specfication Language*. CEA LIST and INRIA, 2008. Preliminary design (version 1.4).

[9] Daniel J. Bernstein. Cryptography in NaCl, 2011. `http://nacl.cr.yp.to`.

[10] Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet. Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In *ICSTW '10*, pages 371–380. IEEE, 2010.

[11] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *CSF'09*, pages 186–199. IEEE, 2009.

[12] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[13] Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with JML. In *Automated Deduction - CADE-20*, pages 116–130. Springer, August 2005.

[14] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[15] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[16] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[17] K. Rustan M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *LNCS*, 1422:254–271, 1998.

[18] Daniel Leivant. Logical and mathematical reasoning about imperative programs. In *POPL*, pages 132–140, 1985.

[19] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of ICISC'05*, volume 3935 of *LNCS*, pages 156–168. Springer, 2006.

[20] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[21] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[22] David A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS'06*, volume 4189 of LNCS, pages 279– 296, 2006.

[23] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2005.

[24] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[25] Josef Svenningsson and David Sands. Specification and verification of side channel declassification. In *FAST'09*, volume 5983 of *LNCS*, pages 111–125. Springer, 2009.

[26] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *SAS'2005*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.

[27] Stephen Tse and Steve Zdancewic. A design for a security-typed language with certificate-based declassification. In *ESOP'05*, volume 3444 of *LNCS*, pages 279–294. Springer, 2005.

[28] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206. IEEE, 2007.

[29] Dennis M. Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *CSFW*, pages 156–169. IEEE Computer Society, 1997.

[30] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer, 1997.

[31] Martijn Warnier and Martijn Oostdijk. Non-interference in JML, 2005. Nijmegen Institute for Computing and Information Sciences, ICIS-R05034.

Appendix A. Proofs

Lemma 2. *Let C be a program, e an expression, and σ_1, σ_2 states.*

1. *If $(e, \sigma_1) \Downarrow^e (v_1, \gamma_1)$ and $(e, \sigma_2) \Downarrow^e (v_2, \gamma_2)$, then* $\mathsf{projFst}(\gamma_1) = \mathsf{projFst}(\gamma_2)$.

2. *If $(C, \sigma_1) \Downarrow (\sigma_1', \delta, \gamma_1)$, $(C, \sigma_2) \Downarrow (\sigma_2', \delta, \gamma_2)$, then* $\mathsf{projFst}(\gamma_1) = \mathsf{projFst}(\gamma_2)$.

PROOF. (1) By structural induction on e. The only case that does not follow directly by induction hypothesis is the *access* of an array element. But, since we are projecting the first components of the memory access traces, the possibly distinct array indexes accessed are irrelevant. (2) Observe that the assumption of distinct labels in C together with the premise that both executions share the control-flow trace δ force the shape of both derivations to be equal (in particular, branching conditions are evaluated to the same truth value). Then, a simple induction on the structure of C allows us to conclude the argument (again, the only case that does not follow immediately from induction hypothesis and (1) is array assignment, and again the first component is state independent).

Lemma 3. *Let C be a program such that $(C, \sigma_1) \Downarrow (\sigma_1', \delta, \gamma_1)$ and $(C, \sigma_2) \Downarrow (\sigma_2', \delta, \gamma_2)$. Then, $\gamma_1 = \gamma_2$ if and only if for all array variables $\mathtt{a}$ in C,* $\mathsf{projArr}^{a}(\gamma_1) = \mathsf{projArr}^{a}(\gamma_2)$.

PROOF. The left-to-right implication is trivial. For the converse, observe that the common execution trace in both final configurations implies, by Lemma 2, that $\mathsf{projFst}(\gamma_1) = \mathsf{projFst}(\gamma_2)$ (in particular, γ_1 and γ_2 have the same length). Now, assume that $\gamma_1 \neq \gamma_2$ and let γ' be the greatest common prefix of γ_1 and γ_2. Since $\gamma_1 \neq \gamma_2$, the length of γ' is strictly smaller than that of γ_1 and γ_2. Consider that the first element where both sequences diverge is now added to this prefix, i.e. $\gamma_1' = \gamma' \cdot (\mathsf{a}, v_1)$ and $\gamma_2' = \gamma' \cdot (\mathsf{a}, v_2)$ (again, by Lemma 2 we know that the first components are equal). By construction, $v_1 \neq v_2$ which implies that $\mathsf{projArr}^{\mathsf{a}}(\gamma_1) \neq \mathsf{projArr}^{\mathsf{a}}(\gamma_2)$.

Lemma 4. *Let C be a program such that $(C, \sigma_1) \Downarrow (\sigma_1', \delta_1, \gamma_1)$ and $(C, \sigma_2) \Downarrow (\sigma_2', \delta_2, \gamma_2)$. Then, $\delta_1 = \delta_2$ if and only if* $\mathsf{tests}^C(\delta_1) = \mathsf{tests}^C(\delta_2)$.

PROOF. The left-to-right implication is trivial. For the converse, assume $\delta_1 \neq \delta_2$ and let δ' be the greatest common prefix of both traces. We firstly observe that δ' is nonempty (its first element is necessarily $\mathsf{firstLabel}(C)$), and that the last label of δ' must be the label of an *if* or *while* statement (in any other case, the control flow is state-independent and thus leads to a common follow-up on both executions). Summarising, we have $\delta_1 = \delta' \cdot \delta_1'$, $\delta_2 = \delta' \cdot \delta_2'$, $\delta' = \delta'' \cdot l'$, l' is a label of an *if* or *while* statement and the greatest common prefix of δ_1' and δ_2' is ε. Since $\delta_1 \neq \delta_2$, it cannot be the case that both δ_1' and δ_2' are empty. Without loss of generality, assume δ_1' is nonempty with l_1' as its first element. Since δ_1' and δ_2' have ε as its greatest common prefix, l_1' cannot be the first element of δ_2', and hence $\mathsf{tests}^C(l' \cdot \delta_1') \neq \mathsf{tests}^C(l' \cdot \delta_2')$. It follows then that $\mathsf{tests}^C(\delta_1) \neq \mathsf{tests}^C(\delta_2)$.

Appendix B. Annotated self-composed crypto_verify_transformed function

```
/*@ predicate body{L1,L2}(unsigned char *x,unsigned char *y,
  @                        integer diffbits1, integer diffbits2,
  @                        list l1x, list l2x, list l1y, list l2y,
  @                        list l1ctrl, list l2ctrl, integer i1, integer i2) =
  @  i2==i1+1 && (diffbits2==(diffbits1|(\at(x[i1],L1)^\at(y[i1],L1)))) &&
  @  l2ctrl==cons(i2<16?1:0,l1ctrl) && l2x==cons(i1,l1x) && l2y==cons(i1,l1y);   */

/*@ inductive loop_pred{L1,L2}(integer i1, integer i2, unsigned char *x,
  @                            unsigned char *y, integer diffbits1,
  @                            integer diffbits2, list l1_x, list l2_x,
  @                            list l1_y, list l2_y,
  @                            list l1_control, list l2_control){
  @  case base_case{L}:
  @   \forall list lx, ly, lcontrol, integer i, diffbits, unsigned char *x,*y;
  @    loop_pred{L,L}(i,i,x,y,diffbits,diffbits, lx,lx,ly,ly,lcontrol,lcontrol);
  @  case ind_case{L1,L2,L3}:
  @   \forall unsigned char *x,*y, list l1_x, l2_x, l3_x, l1_y, l2_y, l3_y;
  @          list l1_control, l2_control, l3_control, integer i1,i2,i3;
  @          integer diffbits1, diffbits2, diffbits3;
  @   loop_pred{L1,L2}(i1, i2, x, y, diffbits1, diffbits2, l1_x, l2_x,
  @                    l1_y, l2_y, l1_control, l2_control) ==>
  @   body{L2,L3}(x, y, diffbits2, diffbits3, l2_x, l3_x,
  @              l2_y, l3_y, l2_control, l3_control,i2,i3) ==>
  @   loop_pred{L1,L3}(i1, i3, x, y, diffbits1, diffbits3, l1_x, l3_x,
  @                    l1_y, l3_y, l1_control, l3_control); }                   */

/*@ requires lmem_control == lmem_control1
  @          && lmem_x == lmem_x1 && lmem_y == lmem_y1;
  @ ensures lmem_control == lmem_control1
  @          && lmem_x == lmem_x1 && lmem_y == lmem_y1;                           */
void crypto_verify(const unsigned char *x, const unsigned char *y,
                   const unsigned char *x1, const unsigned char *y1,
                   int result, int result1) {
   int differentbits = 0, differentbits1 = 0, i = 0, i1 = 0;

   /*@ ghost append_control(i<16);
     @ ghost L1:
     @ loop invariant 0<=i<=16 &&
     @  loop_pred{L1,Here}(0,i,x,y,0,differentbits,lmem_x{L1},lmem_x,
     @                     lmem_y{L1},lmem_y,lmem_control{L1},lmem_control);    */
   while (i < 16) {
      F(i) //@ ghost append_x(i); ghost append_y(i);
      i++; //@ ghost append_control(i<16);
   }
   result = (1 & ((differentbits - 1) >> 8)) - 1;
   /*@ ghost append_control1(i1<16);
     @ ghost L2:
     @ loop invariant 0<=i1<=16 &&
     @  loop_pred{L2,Here}(0,i1,x1,y1,0,differentbits1,lmem_x1{L2},lmem_x1,
     @                     lmem_y1{L2},lmem_y1,lmem_control1{L2},lmem_control1);*/
   while (i1 < 16) {
      F1(i1) //@ ghost append_x1(i1); ghost append_y1(i1);
      i1++; //@ ghost append_control1(i1<16);
   }
   result1 = (1 & ((differentbits1 - 1) >> 8)) - 1;
}
```