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 low-level optimizations 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 propose a formal verification framework to address these policies, extending the range of attacks that could previously be handled using self-composition. We demonstrate our techniques by addressing functional correctness and compliance with security policies for a practical use case.

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 the typically low perceived benefits; finally, it requires writing and optimising code

*Corresponding author
**Principal 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)

for *heterogeneous architectures*, ranging from embedded processors with very limited computational power, memory and autonomy, to high-end servers with low-latency.

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 (such as Hoare logic [17]) and a proof tool. An alternative static approach is *software model checking* [18], which is path-sensitive and focused on full automation, and has been applied to try to solve problems similar to the ones that concern us here.

Deductive program verification 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 significant developments in automated proof technology, in particular SMT solvers. Consequently, verification tools for languages like C [8], C# [6], or Java [19] have in recent years become more and more popular.

This paper follows a series of publications where the authors have explored the application of deductive program verification techniques to prove diverse properties of cryptographic software [2, 3]. Summarising these background results:

- We proposed a *composition*-based methodology for proving the functional equivalence of programs, inspired by the *self-composition* technique [7] for proving information flow properties of programs. Our approach targets the general problem of proving the correctness of concrete implementations with respect to specifications given as reference implementations.

- We employed *natural invariants* as a device to establish a correspondence between axiomatic properties of programs and their operational semantics. This takes us beyond the usual scope of deductive verification, enabling automatic proofs relying on a battery of lemmas, interactively proved once-and-for-all.

- We showed how natural invariants are useful for reasoning about pairs of programs with similar control structures, which in practice is a useful technique to enable proofs about program equivalence. It also allows for the automation of the self-composition technique, previosuly identified as a major problem [29].

- We employed an off-the-shelf verification tool to reason about functional correctness, and safety and security properties of a `C` implementation of the RC4 encryption scheme, included in the well-known open-source library `openSSL`.

In this paper we revise and extend this background to demonstrate the power of our methodology in formally addressing a pressing issue in the implementation of cryptographic software: minimising exposure to *side-channel attacks*.

Contributions. We focus our attention on the recently proposed NaCl [10] (read *salt*) cryptographic library. We analyse a set of high-level security policies adopted by the developers with the goal of minimizing exposure to side-channel attacks in the C implementation of this library . We propose to formally verify compliance with these security policies using the methodology outlined above. To this end, we propose a formal framework that can be summarised as follows:

- We define the operational semantics of a While language with applicative arrays, which explicitly captures the flavour of side-channel leakage addressed by the security policies. Concretely, the semantics constructs traces of the memory addresses read or written to by a program, including program and data memory.
- 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 [22, 28], where we add the capability to handle a wider range of attacks, including cache timing attacks [26] and branch prediction analysis attacks [1].
- We propose a generic technique for formally verifying that a program meets the previous security definition using self-composition and deductive verification, in two steps: 1) one first transforms the original program into one that explicitly collects (minimal) additional information about the execution of the original program in its output state; and 2) one then formally verifies (using the methodology outlined above) that this extra information is independent of secret data. Regarding this global structure, our approach is similar to that proposed in [28].
- We prove that a proof of safety (including termination) of a program and a proof of non-interference for the corresponding transformed program together imply the original program is indeed secure with respect to the intended security policy.

We then discuss how our proof technique can be deployed using real-world deductive verification tools, namely the `Frama-C` framework. We apply this technique to a practical example extracted form the NaCl cryptographic library. We highlight the practical relevance of our results by formally verifying two properties that are representative of recurring problems in cryptographic software:

1. The NaCl implementation complies with the claimed security policies; and
2. the NaCl implementation (which is blatantly unreadable due to the side-channel countermeasures and optimizations) is functionally correct with respect to its high-level specification.

In both exercises we use self-composition and natural invariants, and answer questions raised in [28, 29] 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, eliminating the need to first transform it into a more convenient form that goes around the limitations of the verification framework.

Organisation. In the next Section we briefly discuss and provide motivation for the properties of cryptographic software covered in this paper. In Section 3 we revise the methodology for proofs by composition of [2, 3]. Then Section 4 introduces our framework for the formal verification of side-channel countermeasures, and Section 5 presents the details of a case study extracted from the NaCl cryptographic libary. Finally, in Section 6 we discuss related work, and conclude the paper in Section 7.

2. On the Correctness and Security of Cryptographic Software

Correctness. The standard scenario in deductive verification is that specifications are written as contracts on the function and procedure interfaces. This may involve properties of the output values (typically written in first-order logic), as well as relations between input and output values. For cryptographic software, however, specifications are typically given as operational descriptions (i.e. as algorithms). This is the case, not only in symmetric-key techniques such as ciphers, message authentication codes and cryptographic hash functions; but also in the implementation of algebraic calculations supporting public-key techniques such as digital signature and encryption schemes.

When producing an implementation, the programmer will follow the operational description but is free to introduce optimisations or internal reorganisations, say for improving efficiency or maintainability, or satisfying non-functional security properties (as described below), as long as the input-output behaviour is preserved. To some extent, the specification acts as a reference implementation: verifying functional correctness is reduced to proving program equivalence. This is a difficult (and undecidable) problem, unless we can rely on the fact that implementation and specification share most of their internal structure, which is the case here: the sort of equivalence proof required for cryptographic software correctness corresponds to what is usually known in software engineering as *code refactoring*.

Safety. A widespread verification approach that aims at increasing the level of assurance that can be placed on software implementations is to confine the analysis to a restricted class of properties that rule out the occurrence of some recognisable "bad things". This less ambitious form of verification is called a *safety analysis*, and covers common vulnerabilities arising from coding errors, such as de-referencing invalid pointers, accessing containers with invalid indexes, or calculation errors due to overflows. The advantage of focusing on such simple properties, which are sufficient for a wide class of application scenarios, is that a significant degree of automation can be achieved, minimising user intervention and the impact on development time.

In formal verification tools safety verification is often seen as a prerequisite to more advanced verification tasks, ensuring that the tool will produce valid results for the program under analysis. This is the case of `Frama-c`, adopted in this work.

Security. The notion of a secure implementation of a cryptographic algorithm is an interesting one. At first sight it would appear that functional correctness (in terms of safety and input/output behavior) should imply than an implementation is secure. However, it is well known that physical attacks targeting the computational platform where the algorithm is executing, e.g. side-channel attacks, can succeed in breaking an apparently correct implementation. This means that non-functional security requirements must typically be associated to cryptographic implementations, depending on the specific computational environment in which they are expected to execute.

In this paper we consider a particular instance of such requirements that is adopted in the implementation of the NaCl cryptographic library [10]. Figure 1 contains a quotation from the NaCl documentation, where two high-level security policies are claimed for the library implementation in C. The strategy we adopt to formally verify

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

compliance to these policies is to formalise 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. In Section 6 we provide an overview of developments in this area related to the work in this paper.

Information flow properties are usually verified using a special extended type system [32, 23, 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 $V'_L = \mathit{Vars}(C) \setminus V_H$. 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,

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

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

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

3. Proofs by Composition

In this section we first review *self-composition*, a technique for proving non-interference based on deductive verification, and then a generalisation that can be used to prove *program equivalences*, motivated by the notion of "correctness with respect to a reference implementation", as explained in Section 2. The difficulties of applying self-composition in practice are well-known, and apply also to proofs of equivalence. To overcome these difficulties, we will introduce in Section 3.2 the notion of *natural invariant*, and a technique that establishes a correspondence between program annotations and an underlying formalisation of the operational semantics of the programs.

3.1. Self-Composition and Equivalence by Composition

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:

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

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:

$$\left\{\bigwedge_{x\in V_L'} x = x^s\right\}\ C;C^s\ \left\{\bigwedge_{x\in V_L} x = x^s\right\}$$

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

This method can be extended to handle program equivalence [3]. Let V be the union of the sets of variables occurring in two programs C_1 and C_2. We want to capture the idea that if the programs are executed from indistinguishable states with respect to V, they terminate in states that are also indistinguishable. Thus every execution of the composed program $C_1;C_2^s$, starting from a state in which the values of corresponding variables are equal, must terminate in a state with the same property. This can be expressed as the following Hoare logic total correctness specification:

$$\left[\bigwedge_{x\in V} x = x^s\right]\ C_1;C_2^s\ \left[\bigwedge_{x\in V} x = x^s\right]$$

Note that in fact any arbitrary relation can be considered instead of equivalence:

$$[R_1(\sigma,\sigma^s)]\ C_1;C_2^s\ [R_2(\sigma,\sigma^s)]$$

where σ and σ^s denote state partitions associated with C_1 and C_2^s respectively.

3.2. Natural Invariants

In both scenarios identified above, an obvious difficulty in carrying out the verification comes from the absence of appropriate loop invariants. In what follows we revise our general approach to this problem [2, 3]. 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, typically corresponding to basic *refactoring steps* that are recurrently used in the development of cryptographic software. 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 verification conditions generated by a potentially large number of (self-)composition proofs.

When both programs share much of the underlying control structure, the user may easily guide the interactive verification process by providing hints on the relevant code refactorings. 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	$::=$ $\mathbf{n}$ | `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 $\mathbf{n}$, 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 $\mathrm{acc} : (Z \to Z) \times Z \to Z$ and $\mathrm{upd} : (Z \to Z) \times Z \times Z \to (Z \to Z)$ 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 $[\![e]\!] : \mathit{State} \to Z$. 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:

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

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 – we will see that this will often be 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 ([3]). *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: $\forall n\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2'$,
$\sigma_1 \equiv \sigma_2 \wedge \mathsf{loop}^n_{B,R}(\sigma,\sigma_1') \wedge \mathsf{loop}^n_{B,R}(\sigma,\sigma_2') \Longrightarrow \sigma_1' \equiv \sigma_2'$.

Proposition 2 ([3]). spec *is a morphism that preserves $\equiv$ and is deterministic. More precisely, for every program fragment C and states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$,*

- *If $\sigma_1 \equiv \sigma_2$, $\sigma_1' \equiv \sigma_2'$ and $\mathsf{spec}_C(\sigma_1,\sigma_1')$ then $\mathsf{spec}_C(\sigma_2,\sigma_2')$.*
- *If $\mathsf{spec}_C(\sigma,\sigma_1')$ and $\mathsf{spec}_C(\sigma,\sigma_2')$ then $\sigma_1' \equiv \sigma_2'$.*

Our strategy for reasoning about multiple executions (for self-composition or to justify interesting refactorings) 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.

Self-composition Lemmas. The determinism property is not relevant 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 the program 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

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

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:

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

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

Justifying Loop Refactorings. Justifying code refactorings is difficult only when the refactorings affect loops. For the sake of presentation, we restrict our attention to specifications obtained from single loops with loop-free bodies. That is, we consider natural invariants of the form $\mathsf{loop}_{B,\mathsf{spec}(C)}(\sigma,\sigma')$ where C contains no loops. This is sufficient to cover the refactorings needed for the case-study of Section 5.

The simplest loop refactoring that can be addressed using our technique is *loop unrolling*, which detaches instances of the loop body. This transformation is justified by the following property that results from direct inversion of the definition of loop:

$$\forall n\ n'\ \sigma\ \sigma',\ \mathsf{loop}^n(\sigma,\sigma'1) \wedge n' < n \implies \exists \sigma'',\ \mathsf{loop}^{n'}(\sigma,\sigma'') \wedge \mathsf{loop}^{n-n'}(\sigma'',\sigma')$$

Simple transformations like these are in fact better handled directly at the annotation level, rather than through explicit lemmas. A single natural invariant can be defined for the original loop, and then used in the annotations of the unrolled loop, in order to establish the necessary relation between the execution of the two programs.

Loop body refactorings. To justify more significant code refactorings such as loop body changes that preserve the functionality of the program, we need to rely on an explicit lemma. Consider the equivalence between two programs that have the same control structure, each with its own loop but where the loop-bodies follow different strategies to achieve the same goal. Let us denote the natural invariants of these loops by loop_1 and loop_2. Since the loops share the same control structure (loop condition and the state on which it depends), one can prove *mixed* synchronisation lemmas like

$$\forall n_1\ n_2\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2', \\ \pi^B(\sigma_1) \equiv \pi^B(\sigma_2) \wedge \mathsf{loop}_1^{n_1}(\sigma_1,\sigma_1') \wedge \neg[\![C]\!](\sigma_1') \wedge \mathsf{loop}_2^{n_2}(\sigma_2,\sigma_2') \wedge \neg[\![C]\!](\sigma_2') \Longrightarrow n_1 = n_2$$

The proof is a straightforward generalisation of the single loop version. Once this has been established, one can prove the following main lemma and use it to justify the loop body refactoring (relation Ψ relates states from the original loop and its refactoring):

$$\forall n\ \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2',$$
$$\text{BodyRefactor}_\Psi(\mathsf{body}_1, \mathsf{body}_2) \Rightarrow \sigma_1\ \Psi\ \sigma_2 \wedge \mathsf{loop}_1^n(\sigma_1, \sigma_1') \wedge \mathsf{loop}_2^n(\sigma_2, \sigma_2') \implies \sigma_1'\ \Psi\ \sigma_2'$$

BodyRefactor_Ψ will correspond to simple properties concerning the loop bodies which, as was the case with the self-composition lemmas, are all non-recursive and can thus be regarded as additional proof-obligations, easily discharged by automatic provers:

$$\text{BodyRefactor}_\Psi(\mathrm{R}_1, \mathrm{R}_2) = \forall \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2'.\ \sigma_1\ \Psi\ \sigma_2 \wedge R_1(\sigma_1, \sigma_1') \wedge R_2(\sigma_2, \sigma_2') \Longrightarrow \sigma_1'\ \Psi\ \sigma_2'.$$

4. Formalisation and Verification of Side Channel Countermeasures

In this section we illustrate how the previously presented framework might be used to attest adherence to non-functional security policies such as the ones put forward by the developers of the NaCl library. To do so, we first instrument the semantics of the language to faithfully capture the security policy under scrutiny. Later, we will show how a simple transformation reifies the instrumented semantics, turning the security verification problem into a standard non-interference problem. In Section 5 we provide additional details on how adherence to such a policy can be verified using the framework presented earlier, with off-the-shelf verification tools.

4.1. Instrumented Semantics

We consider two simple additions to the language used before, namely:

- Commands, with the exception of sequential compositions, 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.
- A new syntactic class of list-expressions is considered (together with the corresponding variables and assignment statements). Its use is postponed to the last part of this section, where it is shown how the security policy can be captured by a standard non-interference property.

The syntax of the extended language is given as follows.

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

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

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

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

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

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

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

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

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

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

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

Figure 2: Evaluation semantics

by construction every program should use a non-empty set of labels. We denote the leftmost label used in a program C by $\mathsf{firstLabel}(C)$.

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 $(\mathtt{v},\mathit{offset})$ 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 that can be made is that, for a given program C, the execution trace constrains significantly the correspondent memory access trace. In fact, 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 3. *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 4. *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, $\mathsf{projArr}^a(\gamma_1) = \mathsf{projArr}^a(\gamma_2)$.*

The execution traces of a program C are 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 5. *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 $\mathsf{tests}^C(\delta_1) = \mathsf{tests}^C(\delta_2)$.*

Here, function $\mathsf{tests}^C(\cdot)$ extracts the outcomes of these tests from a given execution trace (for convenience, seen here as snoc-lists).

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

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

4.2. Security Definition and Verification

The security policy can now be expressed as a non-interference-like property.

Definition 6. *Let C be a program, H high-security variables and $V_L' = Vars(C) \setminus H$. We say it is secure, in the sense that it complies with the NaCl side-channel security policies in Figure 1, if*

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

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

Figure 3: Transformation for internalising trace information

A weaker termination insensitive variant is also considered, namely

$$\sigma_1 \overset{V'_L}{=} \sigma_2 \ \wedge (C, \sigma_1) \Downarrow (\sigma'_1, \delta_1, \gamma_1) \ \wedge \ (C, \sigma_2) \Downarrow (\sigma'_2, \delta_2, \gamma_2) \implies (\delta_1 = \delta_2 \ \wedge \ \gamma_1 = \gamma_2).$$

In order to apply standard non-interference verification techniques to our definition, we reduce the security verification problem to a proper non-interference problem by means of a program transformation. The transformation internalises a minimum amount of information from the semantics that is sufficient to guarantee that no secret information is leaked to the control-flow and memory access traces.

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 $\mathtt{xl}^a$ (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 $\mathtt{xl}^a$), and $\langle C \rangle$ is a program that is similar to C but also keeps track of all the conditional tests performed during the execution (in variable `control`), and of all array access indexes (in $\mathtt{xl}^a$). 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 $\mathtt{xl}^a$ (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* $\mathsf{dom}(\overline{\sigma}^0) = \mathsf{dom}(\overline{\sigma}')$,
- $\overline{\sigma}'(\texttt{control}) = \mathsf{tests}^C(\delta')$,
- $\overline{\sigma}'(xl^a) = \mathsf{projArr}^a(\gamma')$.

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 6 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 4 and 5.

The formulation given by Theorem 8 can be readily verified by the self-composition technique, as explained in Section 3. 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.

5. 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 2. We now present an example of how the techniques proposed in this paper can be used in practice to formally verify compliance to these policies. We selected a small example from the core of the NaCl library, which we will describe in detail. The function we selected is called crypto_verify and is presented in Listing 1, top.

It may be surprising to know that the high-level specification for this function is to compare 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 compliance to the security policies we have described. We will show how we can, not only formally verify that this function complies with said policies, but also that it is functionally correct. To establish the former, we will adopt the trace-based approach described in the previous section. To establish functional correctness, we will use the framework introduced in Section 3 to prove functional equivalence with a readable and non-optimized reference implementation of the same algorithm. This can be found in Listing 1, bottom.

```
int crypto_verify(const unsigned char *x, const unsigned char *y)
{
        int differentbits = 0;
#define F(i) differentbits |= x[i] ^ y[i];
        F(0) F(1) F(2)  F(3)  F(4)  F(5)  F(6)  F(7)
        F(8) F(9) F(10) F(11) F(12) F(13) F(14) F(15)
        return (1 & ((differentbits - 1) >> 8)) - 1;
}

int crypto_verify(const unsigned char x[16], const unsigned char y[16])
{
        int res = 0, i = 0;
        while (i < 16) {
                if (x[i] != y[i]) res = (-1);
                i++;
        }
        return res;
}
```

Listing 1: NaCl implementation (top) and reference implementation (bottom) of crypto_verify function

5.1. Verification Infrastructure

In this work, we have used `Frama-c` [8], a tool for the static analysis of `C` programs that contains a multi-prover verification condition generator [15]. We also employed a set of proof tools that included the Coq proof assistant, and the `Simplify`, `Alt-Ergo`, and `Z3` automatic theorem provers. `C` programs are annotated using the ANSI-C Specification Language (ACSL [8]). Both `Frama-c` and ACSL are work in progress; we have used the Boron release of `Frama-c`.

`Frama-c` contains the `gwhy` graphical front-end that allows to monitor individual verification conditions. This is particularly useful when combined with the possibility of exporting the conditions to various proof tools, which allows users to first try discharging conditions with one or more automatic provers, leaving the harder conditions to be studied with the help of an interactive proof assistant. An additional feature of `Frama-c` that we have found useful is the declaration of Lemmas. Like axioms, lemmas can be used to prove goals, but unlike axioms, which require no proof, lemmas originate themselves new goals. In the proofs we developed, it was often the case that once an appropriate lemma was provided (and proved interactively with Coq), all the verification conditions could be automatically discharged. The Coq library described in [3] provides support for proving lemmas such as those introduced in Section 3. 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.

5.2. Verifying compliance to side-channel-related security policies

The first observation we make is that for this simple example one can, not only check by inspection that the NaCl implementation indeed complies with stated policies, but also that the reference implementation is in fact insecure under our definition. Indeed, the NaCl implementation has completely static control flow and array indices, whereas one can infer from the control flow of the reference implementation which (if

```
/*@ 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) {
      F(i) //@ ghost append_x(i); ghost append_y(i);
      i++;
      //@ ghost append_control(i<16);
   }
   return (1 & ((differentbits - 1) >> 8)) - 1;
}
```

Listing 2: Transformed version of crypto_verify function

any) bytes caused the function to detect a difference in its inputs. In order to apply the formal verification approach introduced in the previous section to this example, the first step is to verify safety (and termination) of the NaCl implementation for all valid inputs. This 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 used bit-wise operations. This is needed because a sufficiently expressive axiomatic semantics for these operations is typically not included in off-the-shelf formal verification tools, as such operations are rarely used in general-purpose software.

The second step in our methodology involves reifying the program according to the transformation defined in the previous section. Here we take advantage of a feature of `Frama-c` that is extremely advantageous to our goals. Indeed, 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. This means, in particular, that we do not require a concrete implementation of the list type, which we can model using the logical extensions available in ACSL. Furthermore, using ghost code, we have the guarantee that the semantics of the original program are preserved, as this is imposed as a necessary condition by the deductive verification tool.

In Listing 2 we show the result of transforming a slightly refactored version of the original function. In this version, we use a while loop, since this allows for both a more

compact presentation and a more in-depth demonstration of our technique. Note that this version of the crypto_verify function is actually one of the refactoring steps required to prove functional correctness of the NaCl implementation as we will describe later in the paper, where compliance to the security policies has already been achieved.

We emphasize that the annotated program incorporates the formalism we have introduced in the previous section in a direct way, and that this is made possible by the capability of implementing the necessary lists as logical types. Note the declaration of C functions that allow the construction of the lists within ghost code, and whose semantics is axiomatized so that, at the end of execution, the final state of the ghost variables is essentially a logical term evidencing a sequence of cons operations. Our experience shows that this implementation is highly suitable for passing down to automatic provers.

For this simple example, the proof of non-interference by self-composition is fully automatic. We include the `Frama-c` input in Listing 3. 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. In order to complete this proof automatically, `Frama-c` requires the following lemma to be included:

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

This is an instance of the self-composition lemmas, introduced in Section 3. The instantiation of this lemma in our Coq library is accomplished by invoking the appropriate functor, as shown in Listing 4. The listing also illustrates the definitions contained in the modules `Body` and `SCI` (modules `State`, `Cond`, and `Loop` analogously contain the relevant definitions). Note that the body definition does not include the update to the loop iterator, as this is handled separately in the generic loop pattern in our formalization. Also note that the relation on states matches the pre- and post- conditions in the annotated C code for the self-composition presented above.

5.3. Verfying functional correctness

As mentioned in section 2, our approach to proving functional correctness consists of proving a sequence of program equivalences, starting from a reference implementation, and ending in the target program. Each equivalence corresponds to a simple refactoring that can be addressed using our technique. Listings 1 (top), 5, 6, and 1 (bottom), in that order, comprise the sequence of refactorings we have used for our case study. All of the required equivalence proofs, except for the last one, correspond to loop body refactorings such as those described in Section 3. The final step is a simple unfold of all the loop iterations, also described in the same Section.

We will present only the details related to our use of natural invariants, since the proof-by-composition technique itself is essentially the same as that adopted for the

```
/*@ 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;
}
```

Listing 3: Annotated self-composed crypto_verify_transformed function

```
Module VerifyNaCl := BuildSelfComp State Cond Body Loop SCI.

VerifyNaCl.self_comp_loop :
 forall (r1 r2 : nat) (c : C.St) (x1 x2 y1 y2 : S.St) (d1 d2 : C.St),
    SCI.selfcompProp x1 x2 →
     Loop.loopN r1 x1 c y1 d1 → ~ C.cond d1 →
      Loop.loopN r2 x2 c y2 d2 → ~ C.cond d2 →
       SCI.selfcompProp y1 y2 ∧ C.eqSt d1 d2


Module Body.
Definition body (s1: State.St) (c1: Cond.St) (s2: State.St) : Prop :=
  match s1, c1, s2 with
   (i1,x1,y1,diffB1,lc1,lx1,ly1), n, (i2,x2,y2,diffB2,lc2,lx2,ly2)
      ⇒ diffB2 = bw_or diffB1 (bw_xor (arrSel x1 i1) (arrSel y1 i1))
      ∧ eqA x2 x1 ∧ eqA y2 y1 ∧ i2 = i1
      ∧ lx2 = i1::lx1 ∧ ly2 = i1::ly1 ∧ lc2 = (lt_int_bool i2 16)::lc1
  end.
...
End Body.

Module SCI.
Definition selfcompProp (s1 s2: State.St) : Prop :=
  match s1, s2 with
   | (i1,x1,y1,difB1,lc1,lx1,ly1),(i2,x2,y2,difB2,lc2,lx2,ly2)
     ⇒ i1 = i2 ∧ eqA x1 x2 ∧ eqA y1 y2 ∧ lc1=lc2 ∧ lx1=lx2 ∧ ly1=ly2
  end.
...
End SCI.
```

Listing 4: Instantiation of self-composition lemma in the Coq library

security verification presented before. The natural invariants for the loop in the original program and in refactoring #1 can be found in Listing 7, along with the loop-refactoring lemma (c.f. Section 3) and the contract for the composed program.

The instantiation of this lemma in our Coq library is similar to the one presented for the self-composition example. This is because the two loops have the same control structure, which means that in our formalization the two examples are identical with the caveat that two different body definitions are required, and a different equivalence relation on states is defined. The relevant definitions are shown in Listing 8.

The verification of equivalence for the other refactorings follows in the same lines as this. Overall, the previous formal verification exercises in `Frama-c` implied the (fully automatic) discharge of over 600 proof obligations.

6. Related Work

A good survey of language-based information flow security can be found in [27]. Information flow policies were first introduced by Denning et. al [13] and tend to be formalised as noninterference properties. Information flow type systems, have been used to enforce noninterference in different contexts [32, 24, 23, 30, 31]. The main challenge in designing these systems is that they are often too conservative in practice, so that secure programs may be rejected. Leino and Joshi [20] were the first to

```
int res = 0, i = 0;
int differentbits = 0;
while (i < 16) {

  differentbits |= x[i] ^ y[i];

  i++;
}
if (differentbits!=0) res = (-1);
return res;
```

Listing 5: Refactoring step #1

```
int res, i = 0;
int differentbits = 0;
while (i < 16) {

  differentbits |= x[i] ^ y[i];

  i++;
}
res = (1&((differentbits-1)>>8))-1;
return res;
```

Listing 6: Refactoring step #2

```
/*@ predicate body_loop1{L1,L2}(unsigned char *x, unsigned char *y, integer i1,
  @                            integer i2,integer res1, integer res2) =
  @   i2 == i1 + 1 && ((\at(x[i1],L1) != \at(y[i1],L1) && res2 ==  1) ||
  @                    (\at(x[i1],L1) == \at(y[i1],L1) && res2 ==  res1));      */

/*@ predicate body_loop3{L1,L2}( unsigned char *x, unsigned char *y,
  @  integer i1, integer i2,integer diffbits1 , integer diffbits2) =
  @   i2 == i1 +1  && diffbits2 == (diffbits1 | (\at(x[i1],L1) ^ \at(y[i1],L1)));*/

/*@ lemma eq_loops{L1,L2,L3,L4}:
  @  \forall integer i, int bits1, res, unsigned char *x,*y, *x1, *y1;
  @  \forall integer j; \at(x[j],L1)==\at(x1[j],L3) ==>
  @  \forall integer j; \at(y[j],L1)==\at(y1[j],L3) ==>
  @  loop_pred{L1,L2}(0,i,x,y,0,res) ==> loop_pred3{L3,L4}(0,i,x1,y1,0,bits1) ==>
  @    ((res ==0 && bits1==0) || (res == 1 && bits1!=0));                        */

/*@ requires (\forall integer j; 0<=j<16 ==> x[j]==x1[j]) &&
  @          (\forall integer j; 0<=j<16 ==> y[j]==y1[j]);                       */
void crypto_verify(unsigned char *x, unsigned char *y,
                   unsigned char *x1, unsigned char *y1, int res, int res1);
```

Listing 7: Lemma for first refactoring step

```
Definition body (s1: State.St) (c1: Cond.St) (s2: State.St) : Prop :=
match s1,c1,s2 with (i1,x1,y1,res1,difB1),n,(i2,x2,y2,res2,difB2) ⇒
  (res2 = -1 ∧ (arrSel x2 i1) <> (arrSel y2 i1)
   ∨ (res2 = res1 ∧ (arrSel x2 i1) = (arrSel y2 i1))
  ∧ eqA x2 x1 ∧ eqA y2 y1 ∧ i2 = i1 ∧ difB2 = difB1)
end.

Definition body (s1: State.St) (c1: Cond.St) (s2: State.St) : Prop :=
match s1,c1,s2 with (i1,x1,y1,res1,difB1),n,(i2,x2,y2,res2,difB2) ⇒
  difB2 = bw_or difB1 (bw_xor (arrSel x1 i1) (arrSel y1 i1))
  ∧ res2 = res1 ∧ eqA x2 x1 ∧ eqA y2 y1 ∧ i2 = i1
end.

Definition selfcompProp (s1 s2:VerifyState.St) : Prop :=
match s1, s2 with
 | (i1,x1,y1,res1,difB1),(i2,x2,y2,res2,difB2) ⇒
   i1 = i2 ∧ eqA x1 x2 ∧ eqA y1 y2
   ∧ (res1 = 0 ∧ difB2=0 ∨ res1 = -1 ∧ difB2 <> 0)
end.
```

Listing 8: Coq definitions (functional correctness)

propose a semantic approach to checking secure information flow, with several desirable features: a more precise characterisation of security; it applies to all programming constructs whose semantics are well-defined; and it can be used to reason about indirect information leakage through variations in program behaviour (e.g., whether or not the program terminates). An attempt to capture this property in program logics using the *Java Modelling Language* (JML) [19] was presented by Warnier et al. [33], who proposed an algorithm, based on the strongest postcondition calculus, that generates an annotated source file with specification patterns for confidentiality in JML. Dufay et al. [14] have proposed an extension to JML to enforce non-interference through self-composition. This extended annotation language allows for a simple definition of non-interference for Java programs. However, the generated proof obligations are complex, which limits the general applicability of the approach.

Terauchi and Aiken [29] identified problems in the self-composition approach, arguing that automatic tools (software model checkers like SLAM [4] and BLAST [16]) are not powerful enough to verify this property over programs of realistic size. To compensate for this, the authors propose a program transformation technique for an extended version of the self-composition approach. Rather than replicating the original code, the renamed version is interleaved and partially merged with it. Naumann [25] extended Terauchi and Aiken's work to encompass heap objects, presented a systematic method to validate the transformations proposed in [29], and reported on the experience of using these techniques with the Spec# and ESC/JAVA2 tools.

Natural Invariants provide an explicit rendition of program semantics. In [21] 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 the strengths of higher-order logic reasoning with facilities provided by automatic first-order provers.

Relational Hoare Logic [9] has been used to prove the soundness of program analyses and optimising transformations. Its scope is thus similar to our proofs-by-composition setting. The main difference is the fact that we do not need to move away from traditional Hoare Logic, which allows us to rely on off-the-shelf verification tools.

Svenningsson and Sands [28] first proposed proving resistance against side-channel attacks using self-composition. The authors present a semantic model of security for programs based on the *program counter model* [22] which captures the behavior of an attacker capable of observing the sequence of program counter positions. A technique to achieve declassification through timing channels is also proposed. Our approach extends the previous work in several directions. We enrich the formal framework to capture more powerful side-channel attacks, namely cache timing attacks, and we provide a formal proof that our program reification technique is sound and guarantees side-channel attack resistance under a termination-sensitive flavour of non-interference. We also address the practical problems associated with this approach reported in [28], demonstrating that our methodology permits handling real-world examples without modifying the target program.

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, where the goal is to check that such low-integrity inputs 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 [11] or on control-dependency graphs (CFG) [12]. Our work can be seen as an alternative approach to taint analysis.

7. Conclusion

We have shown how an off-the-shelf deductive verification platform can be used to validate real-world cryptographic software implementations, using the the NaCl cryptographic library as a representative example. Our results focus on three security-relevant properties, with increasing degrees of verification complexity: (1) safety properties and termination; (2) implementation of security policies aiming to reduce exposure to side-channel attacks, formalised as non-interference; and (3) functional equivalence with respect to a reference implementation.

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 side-channel attacks that were previously addressed, but we also show how reasonably automated verification can be made practical using off-the-shelf formal verification frameworks. The general approach we adopt consists of reifying the target program to make explicit in its output state 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 presented these new results as new application scenarios for the general methodology introduced in [2, 3], with promising results. We have also confirmed that the same method introduced to prove non-interference can be applied to the more general case of *equivalence proofs*, to prove the correctness of real implementations with respect to reference implementations. We 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] 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.

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

[4] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL'02*, pages 1–3. ACM, 2002.

[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] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL'04*. ACM, 2004.

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL'02*, pages 58–70. ACM, 2002.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[18] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):1–54, 2009.

[19] 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.

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

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

[22] 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 *ICISC'05*, volume 3935 of *LNCS*, pages 156–168. Springer, 2006.

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

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

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

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

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

Appendix A. Proofs

Lemma 3. *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 4. *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, $\mathsf{projArr}^{\mathtt{a}}(\gamma_1) = \mathsf{projArr}^{\mathtt{a}}(\gamma_2)$.*

PROOF. The left-to-right implication is trivial. For the converse, observe that the common execution trace in both final configurations implies, by Lemma 3, 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. $\gamma_1' = \gamma' \cdot (\mathtt{a}, v_1)$ and $\gamma_2' = \gamma' \cdot (\mathtt{a}, v_2)$ (again, by Lemma 3 we know that the first components are equal). By construction, $v_1 \neq v_2$ which implies that $\mathsf{projArr}^{\mathtt{a}}(\gamma_1) \neq \mathsf{projArr}^{\mathtt{a}}(\gamma_2)$.

Lemma 5. *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 suffix of both traces. We firstly observe that δ' is nonempty (its last element is necessarily $\mathsf{firstLabel}(C)$), and that the first 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 $\delta_1 = \delta_1' \cdot \delta'$, $\delta_2 = \delta_2' \cdot \delta'$, $\delta' = l' \cdot \delta''$ and the common suffix 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 last element. Since $\delta_1' \neq \delta_2'$, l_1' cannot be the last element of δ_2', and hence $\mathsf{tests}^C(\delta_1' \cdot l') \neq \mathsf{tests}^C(\delta_2' \cdot l')$. It follows then that $\mathsf{tests}^C(\delta_1) \neq \mathsf{tests}^C(\delta_2)$.