Verified Translation Validation of Static Analyses

Gilles Barthe, Sandrine Blazy, Vincent Laporte, David Pichardie, Alix Trieu

Verified Translation Validation of Static Analyses

Gilles Barthe*, Sandrine Blazy†, Vincent Laporte*, David Pichardie‡ and Alix Trieu†
*IMDEA Software Institute
Madrid, Spain
Email: gilles.barthe@imdea.org, vlaporte@imdea.org
†CNRS IRISA - University of Rennes 1 - Inria
Rennes, France
Email: sandrine.blazy@irisa.fr, alix.trieu@irisa.fr
‡CNRS IRISA - ENS Rennes - Inria
Rennes, France
Email: david.pichardie@ens-rennes.fr

***Abstract*—Motivated by applications to security and high efficiency, we propose an automated methodology for validating on low-level intermediate representations the results of a source-level static analysis. Our methodology relies on two main ingredients: a relative-safety checker, an instance of a relational verifier which proves that a program is "safer" than another, and a transformation of programs into defensive form which verifies the analysis results at runtime. We prove the soundness of the methodology, and provide a formally verified instantiation based on the Verasco verified C static analyzer and the CompCert verified C compiler. We experiment with the effectiveness of our approach with client optimizations at RTL level, and static analyses for cache-based timing side-channels and memory usage at pre-assembly levels.**

***Index Terms*—verified compilation; Coq proof assistant; program analysis; constant-time programming**

I. Introduction

Static analysis based on abstract interpretation [1] is a principled approach for proving program properties and ensuring program safety. Traditionally, abstract interpretation is performed at source level, partly because source programs have a more explicit control flow and contain more information than intermediate representations or machine code. However, there are many scenarios where it is preferable for analyses to consider intermediate representations or machine code. In particular, analyses of intermediate representations are useful in compilers for detecting opportunities to optimize programs. Likewise, analyses of assembly code or machine code are more appropriate for applications in security, because of the correctness-security gap in compilers [2], [3].

In principle, the need for analyzing lower-levels could be addressed directly, by building abstract interpreters that operate over intermediate representations of interest. However, building an abstract interpreter for a realistic language is a challenging engineering task. It involves implementing symbolic algorithms (e.g., fixpoint computation), numerical computations used by different abstract domains (including a memory abstract domain) that communicate together, and finally a sufficiently precise abstract semantics that keeps track of different kinds of properties, including symbolic equalities between Boolean expressions, values contained in memory cells (including points-to information), and alignment of memory accesses. Moreover, the analysis of lower-level representations may turn out to be less precise than the analysis of the original source programs.

A. Contributions

We propose a new methodology for carrying the results of an abstract interpreter for a source language to lower-level representations. Our methodology does not impact the efficiency of generated code, and it does not require to develop new abstract interpreters for lower-level representations. Instead, our methodology exploits two well-known paradigms:

- inlining enforceable properties: enforceable properties are a general class of program properties that can be enforced using runtime monitors [4]; inlining these monitors yields defensive forms, i.e., programs instrumented with runtime checks for enforcing properties of interest [5].
- relative safety: relative safety is an instance of relational verification. Its goal is to establish safety of a program p_1, under the knowledge that a program p_2 is safe [6]. Relative safety plays an increasingly prominent role, in particular in the context of cross-version verification of real-life software [7]–[9].

Our approach combines these two ideas in a novel way that overcomes efficiency issues with defensive programs; specifically, defensive programs are *only used* as proof artefacts, and the guarantees we get are on the unmodified code generated by the compiler. In more detail, we instrument a source-level analyzer so that it produces defensive programs which verify the results of the analysis using runtime checks, and we define an algorithm that takes as inputs a target program and the annotations for which validation is sought, and returns a defensive target program.

Both defensive programs will fail with a safety violation whenever an annotation fails during execution. The relative-safety checker is applied on the defensive form of the compilation of the original program, and on the compilation of the defensive form of the original program—using in the first case the annotations that must be validated, and in the second case the output of the static analyzer. If the relative safety checker accepts both programs, and the compiler preserves

safety, then the annotations are correct. Note that our relative-safety checker needs not to be adapted for any specific property supported by the source-level analyzer, so we only need a single relative-safety checker per target language, rather than per class of properties verified.

We instrument our approach on top of the CompCert compiler [10], and use it in combination with the Verasco static analyzer [11] for proving properties of low-level programs. Furthermore, we implement and formally verify a relative-safety checker based on a simple but effective notion of product program. We prove the correctness of the relative-safety checker and the transformation into defensive form, and obtain a soundness proof of the overall approach.

We validate our methodology with three use cases. The first use case is optimizing compilation. We demonstrate that lowering the results of the points-to analysis from Verasco and exploiting these results in the common subexpression elimination (CSE) optimization pass of CompCert leads to substantial improvements on the number of loads that can be eliminated: on the set of examples considered, the original CSE from CompCert eliminates less than $60\,\%$ of the loads, whereas our modified optimization eliminates $85\,\%$ of the loads. The second use case is security and more specifically analysis of timing side-channels. One standard methodology used for thwarting timing and cache attacks, especially but not only in cryptographic implementations, is the "constant-time" approach, which mandates that branches and memory accesses do not depend on secrets. We demonstrate that lowering the results of the points-to analysis from Verasco and exploiting these results in an information-flow type system for "cryptographic constant-time" developed by Barthe and others [12] yields major improvements, avoiding the need to rewrite programs extensively and allowing to deal with much larger programs. Our third use case is a resource analysis for stack usage. We show that our method can be used for lowering resource annotations and provides an alternative to the direct approach developed by Carbonneaux and co-workers [13].

B. Summary of contributions

In this paper, we make the following contributions:

- a general methodology to validate the compilation of high-level assertions (such as the properties inferred by a static analyzer), and a proof of its soundness;
- a realization of this methodology, building on the CompCert verified compiler and on the Verasco verified static analyzer;
- a defensive transformation for points-to assertions for C-like and low-level languages. In the case of the low-level language, the defensive form is formally verified (there is no need to verify the defensive form for the source language);
- a relative-safety checker for a low-level language that relies on a simple but effective notion of product program and on a weakest precondition (WP) calculus. The relative-safety checker is also formally verified;

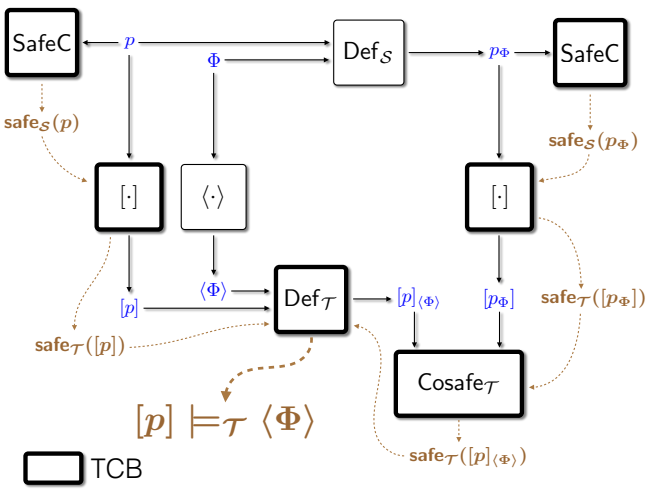

Figure 1. Overview of our methodology

- experimental evaluations on three use cases: the CSE optimization, a "cryptographic constant-time" analysis, and a stack usage resource analysis.

The paper is organized as follows. First, Section II presents our methodology. Then Sections III and IV detail the two main components of our methodology, respectively a relative-safety checker and a generator of defensive forms of programs. Section V shows various intermediate results of the methodology applied to one example program. Section VI describes the experimental evaluation of our methodology. Related work is described in Section VII, followed by conclusions.

Our development is available at the companion website http://www.irisa.fr/celtique/ext/csf17/; in this document links to the Coq formalization for a given definition or theorem are marked with the following symbol: ↗.

II. Technical overview

In this section, we outline our methodology and describe its instantiation based on the verified CompCert compiler and the verified Verasco static analyzer.

A. Methodology

Our methodology is illustrated in Figure 1. Consider a compiler $[\cdot] : \mathsf{Prog}_{\mathcal{S}} \to \mathsf{Prog}_{\mathcal{T}}$ mapping programs from a source language $\mathcal{S}$ to programs in a target language $\mathcal{T}$, and a translation $\langle\cdot\rangle : \mathsf{Spec}_{\mathcal{S}} \to \mathsf{Spec}_{\mathcal{T}}$ mapping properties of source programs to properties of target programs. Our goal is to define a lightweight and automated method for checking that compiled programs satisfy properties in the image of $\langle\cdot\rangle$, i.e., to check $[p] \models_{\mathcal{T}} \langle\Phi\rangle$, where $\models_{\mathcal{T}}$ denotes validity.

We assume given notions of safety $\mathsf{safe}_{\mathcal{S}}$ and $\mathsf{safe}_{\mathcal{T}}$ for source and target programs, and suppose that the compiler preserves safety, i.e., for every source program p, if $\mathsf{safe}_{\mathcal{S}}(p)$ then $\mathsf{safe}_{\mathcal{T}}([p])$. We also assume algorithms that compute defensive programs for source and target levels. These algorithms take as inputs a program p together with a property Φ, and return

a so-called defensive program p_Φ that augments the original program p with additional instructions for checking at runtime that the property Φ is valid. Given a target program q and a target property Ψ, we let q_Ψ denote the defensive form of q with respect to Ψ. We assume that the defensive form on target programs is precise, i.e. for every target program q and target property Ψ, if $\mathsf{safe}_{\mathcal{T}}(q_\Psi)$ then $q \models_{\mathcal{T}} \Psi$.

B. Associated proofs

Our methodology is based on the following observation: thanks to compiler correctness, proofs of $[p] \models_{\mathcal{T}} \langle\Phi\rangle$ can be decomposed into a safety proof $\mathsf{safe}_{\mathcal{S}}(p_\Phi)$, and a relative safety proof $\mathsf{safe}_{\mathcal{T}}([p_\Phi]) \Rightarrow \mathsf{safe}_{\mathcal{T}}([p]_{\langle\Phi\rangle})$. The first proof can be obtained automatically from a safety checker safeC for source programs—in other words, safeC is a Boolean-valued function such that for every source program p, $\mathsf{safeC}(p) = \mathsf{true}$ implies $\mathsf{safe}_{\mathcal{S}}(p)$.

For the second proof, observe that relative safety is a relational property and can be established via the construction and verification of a product program. Informally, product programs are executable representations of relative safety proofs between target programs. Product programs are written in an extension of $\mathcal{T}$ with assert statements and havoc statements, and their validity is captured by the following conditions: if $q_\times$ is a valid product program for q_1 and q_2, and $q_\times$ does not raise assertion failures, then safety of q_1 entails safety of q_2.

In line with recent developments [14], the validity of product programs is verified in two steps. First, we develop a product-program checker $\mathsf{productC}$, which takes as inputs two target programs q_1 and q_2 and a product program $q_\times$ and performs structural verifications to check whether $q_\times$ is a well-formed product program of q_1 and q_2. The correctness of $\mathsf{productC}$ is given by the following implication: if $\mathsf{productC}(q_1, q_2, q_\times) = \mathsf{true}$, and $\mathsf{safe}_{\mathcal{T}}(q_1)$, then either $\mathsf{safe}_{\mathcal{T}}(q_2)$ or $q_\times$ raises assertion failures. Second, we develop an assertion verifier $\mathsf{unfailC}$ for product programs; it checks that the product program will not raise any assertion failure at runtime. The combination of these two algorithms provides a (partial) method for checking relative safety between two programs q_1 and q_2, given a product program $q_\times$. Indeed, relative safety follows from $\mathsf{productC}(q_1, q_2, q_\times) = \mathsf{true}$ and $\mathsf{unfailC}(q_\times) = \mathsf{true}$. To complete the method, we implement a product program generator $\mathsf{productG}$ that takes as inputs target programs q_1 and q_2 and computes a product program $q_\times$. This generator together with the method for checking relative safety are represented in Figure 1 by the box called cosafe_τ.

In summary, we propose an algorithmic method for formally verifying properties of target programs, based on the following components: defensive forms for source and target programs, and generator of product programs, and three checkers: a safety checker safeC for source programs, a product program checker $\mathsf{productC}$ relating target and product programs, and a verifier $\mathsf{unfailC}$ for product programs.

C. Instantiation of the methodology

We develop a verified instantiation of our method on top of the CompCert compiler. In this setting, safety has a precise meaning: a program is safe if it has no undefined behavior according to the CompCert semantics.

Our source language is the C-like language C#minor; C#minor is the intermediate form considered by the verified static analyzer Verasco, and therefore we can use Verasco for safety checking. Our target language is RTL (after CompCert optimization passes). RTL is a natural trade-off, for two reasons. First, choosing RTL over lower-level representations (which assume finitely many registers) simplifies the instantiation, both for the defensive form of target programs and also for the construction and verification of product programs. Second, most of the complexity of the compiler is found in its middle-end; choosing RTL eliminates the need to prove correctness of translation of annotations of middle-end optimizations. Moreover, it is relatively direct (and done in this paper) to prove correctness of the translation of annotations from RTL to assembly-level.

We focus on a restricted class of properties, namely points-to assertions. These assertions capture aliasing relationships and are essential for detecting and validating optimizations on intermediate representations and for carrying information-flow analyses on assembly programs. Pleasingly, the CompCert memory model is shared across the different intermediate languages, and therefore so are points-to assertions; in particular, the function $\langle\cdot\rangle$ is extremely simple. It should be noted however that dealing with points-to annotations requires some ingenuity because symbolic pointers computed by the static analysis refer to call stacks, which must therefore be accounted in the code of defensive programs. We solve this issue by introducing (only) in the defensive program a shadow stack and ensuring that it remains in line with the stack of the original program.

Regarding to the notations used in Figure 1, given a source program p and its defensive form p_Φ, we use twice the Verasco formally verified static analyzer to prove both properties $\mathsf{safe}_{\mathcal{S}}(p)$ and $\mathsf{safe}_{\mathcal{S}}(p_\Phi)$. Properties $\mathsf{safe}_{\mathcal{T}}([p])$ and $\mathsf{safe}_{\mathcal{T}}([p_\Phi])$ are ensured by the CompCert formally verified compiler. Moreover, the property $\mathsf{safe}_{\mathcal{T}}([p]_{\langle\Phi\rangle})$ is ensured by our relative-safety checker that we detail in section III.

As required by our methodology, we prove that the defensive form at target level is precise, under the additional assumption that the target program is safe. Making this additional assumption does not affect the generality of our approach, since the target program is obtained via CompCert and hence is safe.

The product checker is novel, and optimized for the purpose of proving relative safety between programs that are structurally similar in a strong sense. Finally, the verifier uses standard techniques for generating a set of verification conditions, together with simple (but effective) custom tactics for discharging verification conditions automatically.

D. Trusted computing base

The Trusted Computing Base (TCB) of our methodology includes the generator of target defensive forms, the safety

checker, the relative-safety checker, and the product program checker for validity and non-failure (see Figure 1). On the contrary, the generators of product program and source defensive form are not in the TCB and may be incorrect—as usual, it may limit the usefulness of our approach, but it will not affect its soundness.

An important aspect of our instantiation is mechanization: in order to eliminate the compiler, target defensive form, and checkers from the TCB, they are programmed in a proof assistant, that is also used to formally prove the assumptions used for justifying our method: preservation of safety for the compiler, precision of defensive form on target programs, correctness of the checkers. Thus, our instantiation is foundational, in the sense that the TCB solely consists of the CompCert compiler.

III. Relative-safety checking

One main component of our methodology is a checker for relative safety. This section describes the design and formal verification of a checker for a slightly more powerful property, namely equivalence.

A. Overview

The purpose of this component is to prove the safety of a program $\mathbf{R}$ knowing that another program $\mathbf{L}$ is safe. In our setting, the two programs to compare are two low-level defensive programs that, by construction, are very similar. Therefore, we prove a much stronger property: a simulation, which in turn implies program equivalence, hence relative safety. Specifically, the relative safety checker generates an obligation for all control-flow instructions and potentially unsafe instructions (instructions in any of these two sets are called "critical"): each time the $\mathbf{R}$ program branches, the $\mathbf{L}$ program must take the same path; this ensures that the programs have the same (local) control flow. Moreover, each time the $\mathbf{R}$ program attempts to perform a potentially-unsafe step, the $\mathbf{L}$ program can perform the same potentially-unsafe step. Knowing that the $\mathbf{L}$ program is safe, we can conclude that the step in $\mathbf{R}$ is also safe.

In addition, the similarity in the structure of the two programs enables us to do a modular proof. Indeed, they execute the same functions, in the same order, with equal arguments. Therefore, it suffices to check equivalence of each pair of individual functions: checking can thus be performed intra-procedurally, and can be applied even if the call-graph is unknown — e.g., due to function pointers and recursion.

The equivalence proof of two functions L and R is built from a product program f. Such a program — expressed in a simple language called RTL$^{\ominus}$ (see Figure 2) — over-approximates the behaviors of the two initial functions L and R: it features the safe instructions of both L and R, and is decorated with assertions claiming that the critical steps of L are the same as the critical steps in R. This means that the two functions may arbitrarily differ on their safe sides, provided their critical parts match. Since we expect the two functions to have similar loop structures, we consider as critical the instructions whose execution may be stuck — i.e., may have undefined behavior

Instructions:

$i ::=$	$\mathsf{skip}(l)$	no operation (go to l)
	$\mid \mathsf{op}(op_{target}, \vec{r}, r, l)$	arithmetic operation
	$\mid \mathsf{if}(op_{target}, \vec{r}, l_{true}, l_{false})$	if statement
	$\mid \mathsf{return}$	function end
RTL only ↑	$\mid \mathsf{load}(\alpha, \kappa, addr, \vec{r}, r, l)$	memory load
	$\mid \mathsf{store}(\alpha, \kappa, addr, \vec{r}, r, l)$	memory store
	$\mid \mathsf{call}(sig, id, \vec{r}, r, l)$	function call
RTL only ↓	$\mid \mathsf{return}\ r$	function return

Control-flow graphs:

$f, g : l \mapsto i$	finite map

Functions:

$L, R ::= \{\mathsf{param}\ \vec{r}; \mathsf{stack}\ n; \mathsf{start}\ l; \mathsf{graph}\ g\}$

Figure 2. Syntax of the RTL and RTL$^{\ominus}$ languages

in the RTL small-step semantics defined in CompCert — but also those that appear as loop headers or contribute to the call-graph (calls and returns).

If the product is *valid*, i.e., if none of its executions can violate the assertions, then the two functions are equivalent. In this way, we effectively reduce the problem of checking equivalence to the problem of validating that a RTL$^{\ominus}$ program is well-annotated. Of course, since equivalence checking is in general undecidable, the product construction may fail.

B. The language for product programs

The equivalence checker operates on programs written in the RTL language of CompCert. Its formal syntax is given in Figure 2. The RTL intermediate language represents programs by control-flow graphs with explicit program points, is structured in functions and instructions, and uses machine-dependent operations and addressing modes (op_{target}). Instructions correspond roughly to elementary instructions of the target processor, but operate over temporaries (i.e., pseudo-registers, where infinitely many pseudo-registers are available) [10]. Each instruction lists explicitly the nodes of its successors.

So as to implement our methodology, we have added to the syntax of load and store instructions a named annotation (represented by the α meta-variable in Figure 2). Such annotations do not affect the semantics of the RTL language: we will later define the meanings of a particular instantiation of these annotations and use them to prove invariants of annotated programs (see Section IV-B).

The language RTL$^{\ominus}$ in which the product programs are expressed is basically a subset of RTL. Thanks to the great similarity of the two programs to compare, the products need not to call functions, to return values, or to access the memory. To model these features, we rely on an extra havoc operator for non-deterministic assignment. Thus only four instructions are used in RTL$^{\ominus}$: skip, operations (including the havoc operator), conditional branches and return without value. Nodes of a product program are decorated with sets of assertions: first-order formulas involving claims about symbolic expressions se.

Symbolic expressions:

$se ::= r \mid id \mid op_{target}(\vec{se})$

Assertions:

$a ::= \mathsf{True} \mid se \mid se_1 = se_2 \mid a_1 \implies a_2 \mid \neg a \mid \forall r, a$

$$\mathcal{SE}_\rho(r) = \rho(r)$$
$$\mathcal{SE}_\rho(id) = id$$
$$\mathcal{SE}_\rho(op_{target}(\vec{se})) = \mathcal{E}(op_{target}; \mathcal{SE}_\rho(\vec{se}))$$
$$[\![\mathsf{True}]\!](\rho) = \text{true}$$
$$[\![se]\!](\rho) = \mathcal{SE}_\rho(se) = 1$$
$$[\![se_1 = se_2]\!](\rho) = \mathcal{SE}_\rho(se_1) = \mathcal{SE}_\rho(se_2)$$
$$[\![a_1 \implies a_2]\!](\rho) = [\![a_1]\!](\rho) \implies [\![a_2]\!](\rho)$$
$$[\![\neg a]\!](\rho) = \neg [\![a]\!](\rho)$$
$$[\![\forall r, a]\!](\rho) = \forall v, [\![a]\!](\rho[r \mapsto v])$$

Figure 3. Syntax and semantics of RTL$^\ominus$ assertions

Their syntax is formally given in Figure 3: expressions are made of registers, identifiers and RTL operations applied to such expressions; assertions claim that a symbolic expression evaluates to true, that two expressions evaluates to the same value, or are composed of assertions using implication, negation, and universal quantification over the value of a given register. Universal quantification is not expected to appear in assertions generated during the product construction. However, it is introduced when we compute weakest preconditions of havoc instructions. The semantics of these expressions is defined by a partial evaluation function $\mathcal{SE}.(\cdot)$, also defined in Figure 3, which itself relies on the evaluation $\mathcal{E}(\cdot;\cdot)$ of RTL operations. Finally, the semantics of assertions is formally defined at the bottom of Figure 3 as predicates over register valuations ρ.

The semantics of a product function f is expressed as a small-step relation between execution states $\langle l; \rho\rangle$ in which l is the program point of the next instruction to execute and ρ is the current valuation of the registers. It is given in Figure 4, where $\mathcal{E}(e; \vec{a})$ denotes the evaluation of operation or condition e on arguments $\vec{a}$. Notice that there is no call-stack nor memory in the state. There is also a particular state $\bullet$ that denotes a terminated execution. Notation $f[l]$ represents the instruction at program point l in control-flow graph f, if any.

Execution starts at node 1, and registers initially have arbitrary values, which may be constrained by some precondition. Execution of instruction return results in the final state. Instruction $\mathsf{skip}(l')$ leaves the registers unchanged and moves to program point l'. To execute instruction $\mathsf{op}(o, \vec{a}, d, l')$, in which o is a RTL operator, this operator is first applied to the values of registers $\vec{a}$ in the current state; this yields a value v that is assigned to register d; execution then moves to program point l'. The havoc operator takes no arguments and evaluates to any (unspecified) value v. To execute instruction $\mathsf{if}(c, \vec{a}, l_{true}, l_{false})$, the condition c is first evaluated on the values of registers $\vec{a}$ in the current state; depending on the

$$\frac{f[l] = \mathsf{return}}{\langle l; \rho\rangle \to \bullet} \qquad \frac{f[l] = \mathsf{skip}(l')}{\langle l; \rho\rangle \to \langle l'; \rho\rangle}$$
$$\frac{f[l] = \mathsf{op}(o, \vec{a}, d, l') \qquad \mathcal{E}(o; \rho(\vec{a})) = v}{\langle l; \rho\rangle \to \langle l'; \rho[d \mapsto v]\rangle}$$
$$\frac{f[l] = \mathsf{op}(\mathsf{havoc}, [], d, l')}{\langle l; \rho\rangle \to \langle l'; \rho[d \mapsto v]\rangle}$$
$$\frac{f[l] = \mathsf{if}(c, \vec{a}, l_{true}, l_{false}) \qquad \mathcal{E}(c; \rho(\vec{a})) = \text{true}}{\langle l; \rho\rangle \to \langle l_{true}; \rho\rangle}$$
$$\frac{f[l] = \mathsf{if}(c, \vec{a}, l_{true}, l_{false}) \qquad \mathcal{E}(c; \rho(\vec{a})) = \text{false}}{\langle l; \rho\rangle \to \langle l_{false}; \rho\rangle}$$

Figure 4. Semantics of product programs

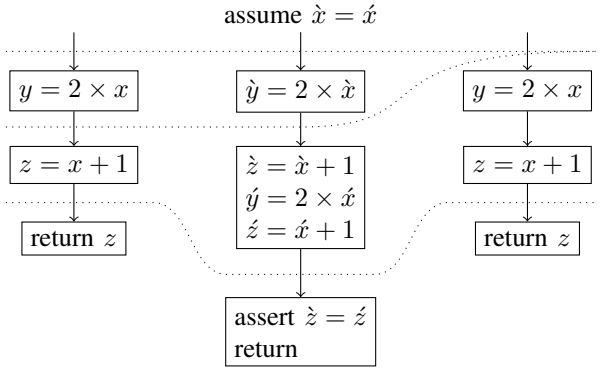

Figure 5. A function and its self-product

Boolean result, execution proceeds, with unchanged registers, either to program point l_{true} or to program point l_{false}.

C. Well-formed products

To be able to reason about product programs without reasoning on the exact procedure that builds them, we first give a formal specification of a well-formed product f of two functions L and R.

The registers of a product program mirror the registers of the original programs; to make this link more visible, we note $\grave{r}$ the register which mirrors register r of the left program, and $\acute{r}$ the register which mirrors register r of the right program. (We assume that for all names x and y, the names $\grave{x}$ and $\acute{y}$ are distinct.)

The property that we intend to prove is asymmetric: if L is safe, then so is R. Similarly, the specification of well-formed products, which justifies this fact, is asymmetric. This specification is built on the notion of *cuts*, acting as synchronization points on critical instructions in the product program (which interleaves the instructions of the two initial functions). A cut is a triple $(\grave{\mathsf{pc}}, \pi, \acute{\mathsf{pc}})$ of related program points, where $\grave{\mathsf{pc}}$ is a program point of function L, $\acute{\mathsf{pc}}$ is a program point of function R, and π is a program point of the product f. Figure 5 gives an example of a product program and shows its associated cuts. It presents a function with parameter x and body $y = 2 \times x; z = y + 1;$ return z — once on the left,

and once on the right — and a possible well-formed product of this function with itself — in the middle. For the sake of readability, we took some liberty with the syntax. The dotted lines represent the cuts.

With this notion of cuts, there is some freedom in how similar the two programs must be. In particular, the control-flow is only required to be locally similar, which makes the relative-safety checker robust to various kinds of loop unrollings.

The entry points must form a cut, and the product has a precondition that states that the function arguments are equal; in the example, $\grave{x} = \acute{x}$.

The path from one cut to the following one corresponds to the parallel execution of the three functions, from the corresponding program points. Along such a path, the L function executes exactly one step. For the R function, there are two cases:

- either the step in L is safe, then R executes an arbitrary number of safe steps (none, one or many, like in the first two cuts in Figure 5);
- or the step in L is critical, then R executes a similar step followed by an arbitrary number of safe steps (e.g., the return step in Figure 5).

In the first case (a left step that is not critical), we require that the product has the following shape: one instruction that models the (safe) instruction from L, followed by a sequence of (safe) instructions that model the instructions from R. This definition captures any kind of reordering and interleaving of non-critical instructions.

In the second case (critical left step), the product obeys some constraints, that are specific to each instruction, and described in Figure 6. These constraints have the following shape:

- the left instruction and the right one must be similar;
- the product is decorated with assertions that claim that the arguments of the instructions are equivalent (in the example, that the returned values are equal);
- one or more instructions model the common instruction;
- the product features register to register copies that capture the property that both original instructions have obtained the same result.

It must be noted that only the shape of the critical instructions must match; they can be applied to dissimilar arguments (for instance to variables with unrelated names).

In both cases, the nodes that are reached at the end of the paths that are related at a cut must also form a cut.

More specifically, for loads, addresses must be equal, the read value is indeterminate (hence the havoc operator), yet both loads read the same value, as shown in Figure 6[1]. For stores, addresses must be equal, written values must be equal, and the statement has no effect. For conditional branches, the branching conditions must be equivalent (cnz is the RTL operation which turns its integer argument into a Boolean value). A function call is similar to the load and store cases at once, but with a more sophisticated address computation: called functions must be equal and arguments must be equal. For returns, the returned values must be equal, if any.

[1]This product construction can be adjusted to express weaker invariants of the memories: for instance that the two memories only agree on a *low* part but may arbitrarily differ on a *high* part. Such an invariant could be useful to prove non-interference of a program. But such a use of a product program is way beyond the scope of this paper.

Left	Right	Product
$x = \text{load}_\kappa\ p$	$y = \text{load}_\kappa\ q$	assert $\grave{p} = \acute{q}$ $\grave{x}$ = havoc $\acute{y} = \grave{x}$
$\text{store}_\kappa(p, u)$	$\text{store}_\kappa(q, v)$	assert $\grave{p} = \acute{q}$ assert $\grave{u} = \acute{v}$
if (x)	if (y)	assert cnz($\grave{x}$) = cnz($\acute{y}$) if $(\grave{x})$
$x = p(\vec{u})$	$y = q(\vec{v})$	assert $\grave{p} = \acute{q}$ assert $\grave{u} = \acute{v}$ $\grave{x}$ = havoc $\acute{y} = \grave{x}$
return	return	return
return x	return y	assert $\grave{x} = \acute{y}$ return

Figure 6. Product of critical instructions

Finally, we need to ensure that both functions eventually progress, i.e., that none is waiting forever. Since critical steps are always synchronized due to the previous rules, problems may only arise with safe steps. On the one hand, the conditional branches and loop headers are considered as critical steps: this ensures that safe paths are free of branches and cycles, hence that the left program does not wait forever. On the other hand, to justify that the right program does not wait forever, cuts have a *height* that counts how many left steps will be executed before the right function makes progress. These heights are computed when checking the well-formedness of the product, along with the property that heights actually decrease on every left step. On Figure 5, the first cut has height one (as the right function waits for one step), and the two other cuts have height zero.

So as to prove that our product construction yields well-formed products, we have implemented a checker for well-formedness and proved it correct.

D. Valid products

The key of relational verification using product programs is to reduce relational properties to properties about a single program: relational invariants (about the running states of two programs) are expressed as invariants of a single program (the product).

The validity of the assertions within the product programs (as program invariants) justifies the simulation between the two initial functions. We thus formally define the validity of products and how this property is automatically checked.

Definition III.1 (Valid state, valid function). Given a RTL$^\ominus$ function f decorated with assertions δ (where δ maps nodes to sets of predicates), an execution state $\langle l; \rho \rangle$ is *valid* when the

valuation of the registers satisfies all assertions a at the current program point l: $\forall a, a \in \delta(l) \implies [\![a]\!](\rho)$. Function f is *valid under precondition* P, noted $f \models^P \delta$, when all reachable states are valid, i.e., when the "valid state" predicate is an invariant under precondition P.

So as to prove the standard verification problem that a function is valid, we implemented a weakest-precondition (WP) calculus for the RTL$^\ominus$ language as well as a verification-condition generator (VC-gen). To this end, we need to infer loop invariants. These invariants must reflect the transformation which links the two programs. In our use cases, the invariants correspond to equality of the variables that are live at the loop headers of the initial programs. We use the liveness analysis that is available in CompCert for RTL functions to automatically infer these invariants. The resulting verification conditions, expressed in the assertion language presented in Figure 3 are then automatically discharged by a simplification procedure, that we implemented and proved correct in Coq.

The formal correctness of the implementation of the checker for product validity is made explicit by the following theorem.

Theorem III.1 (Correctness of the VC-gen). *Given a* RTL$^\ominus$ *function f decorated with assertions δ and a precondition P, the verification-condition generator returns a set of verification conditions such that the conjunction of all these properties implies that the decorated function is valid under the given precondition:*

$$\bigwedge_{a \in \text{VC-gen}(f,\delta,P)} (\forall \rho, [\![a]\!](\rho)) \implies f \models^P \delta.$$

E. Simulation

The correctness theorem of the product construction expresses that the assertions in the product functions capture the relative safety of the two initial programs.

Theorem III.2 (). *Given two programs* $\mathbf{L}$ *and* $\mathbf{R}$*, if they have the same global variables and if for every function L in* $\mathbf{L}$*, there exists a function R in* $\mathbf{R}$*, with same name and signature, such that there exists a well-formed product f of L and R, then, if all such products are valid, there is a simulation between* $\mathbf{L}$ *and* $\mathbf{R}$*.*

The proof sketch is the following. We introduce a simulation relation between states that extends the notion of cuts (program points in related states belong to some cut). Formally, related states have equal memories and equal stack pointers; their program counters (respectively $\grave{\text{pc}}$ and $\acute{\text{pc}}$) as well as their register banks (respectively $\grave{\text{rs}}$ and $\acute{\text{rs}}$) are related through a program point l and a register bank ρ such that: the state $\langle l; \rho \rangle$ is reachable in the product; program points $\grave{\text{pc}}$, $\acute{\text{pc}}$, and l form a cut; and the content of the register banks agree:

$$\forall x, \grave{\text{rs}}(x) = \rho(\grave{x}) \wedge \acute{\text{rs}}(x) = \rho(\acute{x}).$$

Notice that in this relation, the content of the registers of the left program and of the right program are *not* directly related. When the simulation reaches a program point decorated with assertions about the state of the product program, the validity of an assertion entails a relation between the contents of the registers of the initial programs. For instance, if the next instruction to execute in the left program is `return x`, then the well-formedness of the product function implies that the next instruction in the right program also has the shape `return y` (for some unspecified register `y`) and the validity of the assertion $\grave{x} = \acute{y}$ in the product implies that the returned values in the two programs are equal.

```
char G[3], H;
void init(char *p, int *q) {
    p += any_int() % 3;
    *p = 0;    // G: [0; 2]
    *q = 1; }  // 1.x: [0; 0]
```

```
int main(void) {
    int x;
    init(G, &x);
    return x;  // 0.x: [0; 0]
}
```

Figure 7. A simple program

IV. Defensive encoding of annotations

This section details the second main component on which our methodology relies, namely the generation of defensive programs from annotated programs. This component has two goals: first it must *precisely* encode the assertions, so that the defensive programs always fail when they detect an assertion violation; second, and this is more specific to this methodology, the two defensive programs must be sufficiently similar, to ensure the success of the relative-safety check. We first describe its implementation and then its formal verification.

A. Annotation syntax

We focus on *points-to* annotations: each instruction that accesses the memory (i.e., every load and store) is annotated with an optional set of symbolic pointers. Moreover, during compilation, local variables of functions are forgotten and simply allocated in a single stack frame at different offsets during the compilation from C#minor (i.e., before generating RTL code, on which our defensive transformation operates). Thus, we define a symbolic pointer as a symbolic block (either a global variable name or a depth in the call stack) together with a concrete range that denotes the pointer offset. Syntactically speaking, we use the annotation (`d.x: [l; h]`) to represent pointers to the variable `x` in the stack frame at relative depth `d` in the call stack and whose offsets are between `l` and `h`; and the annotation (`G: [l; h]`) to represent the pointers to the global variable `G` whose offsets are between `l` and `h`.

As an example, consider the program of Figure 7; it is shown using C syntax for easier reading but the annotation inference is done at the C#minor level. The three annotations that are automatically inferred by the Verasco static analyzer are shown as comments in the figure. There are three memory accesses in this program: the store through pointer `p`, the store through pointer `q`, and the load of `x` at the end of the main function. The first one writes global variable `G` at some offset between 0 and 2 (because of the `%3` computation); it can thus be annotated with (`G: [0; 2]`) in the `init` function. The second

one writes the local variable x of the main function; when this store is run, the main function is at relative depth 1 in the call stack; therefore this store is annotated with (1.x: [0; 0]). The third memory access loads the local variable x of the main function (*i.e.*, at relative depth 0 in the call stack); it is thus annotated with (0.x: [0; 0]).

B. Annotation semantics

Each annotation represents a set of concrete pointers. The program[2] allows us to statically compute the concrete addresses of global variables, but the addresses of the stack frames depend on the actual execution state when an annotated instruction is about to be executed. Therefore, to dynamically interpret an annotation, we extract the call stack from the current execution state.

C. Annotation encoding

The aim of this component is to produce a defensive program which dynamically checks the validity of all assertions. Therefore, for each memory access through a pointer p annotated with a set α of symbolic pointers, the defensive program checks that p is actually one of the pointers in the denotation of α.

There are two cases, depending whether the block of the pointer is definitely known or not. If the block is known, an annotation is encoded as two inequality comparisons with the range boundaries (or a disjunction of such tests, when there are several ranges). For instance, the annotation (G: [0; 2]) attached to the read through pointer p in Figure 7 is encoded as the assertion $\mathsf{G} \leqslant \mathsf{p} \wedge \mathsf{p} \leqslant \mathsf{G} + 2$.

If the block is unknown — since the inequality comparison is not well-defined for pointers of different blocks, but equality comparison is — the defensive program enumerates all the pointers in the denotation of α and compares each of them for equality to p. For example, if the set of annotations was the set { (G: [0; 2]); (H: [0; 0]) }, it would be encoded by the assertion $\mathsf{p} = \mathsf{G} \vee \mathsf{p} = \mathsf{G} + 1 \vee \mathsf{p} = \mathsf{G} + 2 \vee \mathsf{p} = \mathsf{H}$.

This second encoding might look very inefficient, but remember that the defensive program is not meant to be ever executed; it is only an intermediate artifact that witnesses the validity of the annotations.

D. Forging pointers: the shadow stack

In order to build defensive tests, we need to compute some concrete pointers that are symbolically given by the annotations. This is an issue when the annotation refers to a local variable of some suspended function. To forge such a pointer is generally not possible without any runtime support: there is *a priori* no direct way to forge a pointer to a stack frame of an arbitrary suspended function. Therefore, we make each function *leak* a pointer to its stack frame into a global variable (the so-called *shadow stack*).

The shadow stack is a global array that records a pointer to the stack frames of all currently running functions. The top of the shadow stack always holds a pointer to the stack frame of the current function. To maintain this stack, we add to each function a *prologue* whose execution pushes the current stack pointer atop the shadow stack and an *epilogue* whose execution pops a value from the shadow stack. Such an epilogue is actually inserted before every return instruction.

[2]A global environment, in CompCert parlance

E. Correctness theorem and proof

An execution state is said to be correctly annotated when either the next instruction to be executed is not an annotated memory access, or it is a memory access through a pointer p and it is annotated with a symbolic set of pointers α, such that pointer p belongs to the denotation of α.

The correctness theorem of the defensive encoding of a program ensures that the validity of the annotations is completely assessed by the safety of the defensive program.

Theorem IV.1 (Precision of the defensive form ↗). *Given a* safe *annotated RTL program p, if the defensive version of p is also safe, then every reachable state in the execution of p is correctly annotated.*

This theorem is only proven at the RTL level and not at the C#minor level as we do not need it for our methodology. Indeed, we only require the defensive program to be safe. In order to prove this theorem, we equip the original program p with a *blocking* semantics which refines the original RTL semantics to dynamically check, before every execution step that the current state is correctly annotated. Thus, proving that p is safe with regards to the blocking semantics entails that every reachable state of the program is correctly annotated.

The standard technique used throughout CompCert to prove that safety is preserved is to show a simulation between both programs. However, the corresponding compiler transformations need only to prove a forward simulation (i.e., that a safe original program results into a safe transformed program), while we need to prove the opposite direction. We thus have to directly show a backward simulation between the transformed program p' and the original program p. This cannot be obtained from a forward simulation as usually done in CompCert, as we would need to be able to match one step in the defensive program with steps in the original program, which is not possible for steps involved in the defensive checks. As always with such simulation proofs, the gist of our proof is to define the matching relation between execution states of both programs.

F. Compatibility of the two defensive programs

We produce two low-level defensive programs from a single high-level source: on the one hand we compile and then produce the defensive program; on the other hand, we first produce the defensive program and then compile it. These two operations (transformation into defensive form and compilation) do not necessarily commute. The equivalence checker can accommodate for some difference between the two compared programs, but the closer they are, the simpler is the work for the equivalence checker. Therefore we try to avoid unnecessary differences between the two programs. However, we have identified the following factors which contribute to such differences.

a) Wrong interleaving of the defensive checks: All critical operations must appear in the two programs in exactly the same order. Therefore, when producing the high-level defensive code corresponding to a single instruction with several annotations (e.g., ++*p;) the interleaving of the two defensive checks and the two memory accesses need to be guessed. In other words, the compilation of complex instructions needs to be correctly anticipated. Indeed, the low-level program has an annotated load and an annotated store as two distinct instructions. So, the (high-level) defensive check corresponding to the annotation of the store must appear *after* the code for the load.

b) Impact of the defensive transformation on the optimizations: Our defensive transformation affects the ability of the compiler to fully optimize the defensive program, which is thus possibly *less* optimized than the original program. It should be emphasized that the use of our methodology should not affect how the original program gets optimized: we should neither turn off the optimizations nor weaken them.

The major issue comes from the shadow stack: without it, some optimizations exploit the fact that the stack pointer does not escape, and therefore that calling a function cannot modify the local variables. Since the very purpose of the shadow stack is to make pointers to all stack frames available to all functions, there is no longer a way for the optimizer to prove that some stack pointer does not escape. Thus some optimizations that could be performed are hampered by the defensive transformation.

If the optimizer was aware of the shadow stack and of the fact that the leaked pointers are never used to access to the memory but only in comparisons, it could treat the leaks through the shadow stack as benign and proceed as if they were not there. This suggests that a proper implementation of shadow memory requires some support from the compiler. We have left the tackling of this issue as future work.

c) Optimizations of the defensive transformation: The defensive program which is produced at high-level undergoes the same optimizations as the original program. But the defensive program produced at low level (i.e., after the RTL optimizations passes) does not undergo these optimizations. This ends up with different defensive programs. To overcome this issue, we have investigated two different approaches.

On the one hand, we have restrained the optimizations that are applied to the high-level defensive programs, so that they do not target the added defensive code. For instance, the CSE optimization would remove many loads from the shadow stack and replace them by simple move instructions. Thus, to prove that the two defensive programs are equivalent, the equivalence checker would have to perform complex reasoning about the memory to justify that the loads on the left return the same values as the moves on the right. This reasoning is the same as the one that is performed during the CSE pass to justify the optimization; it is a reasoning on a single program rather than a relational property. Therefore, to avoid adding extra burden to the equivalence checker, we make the CSE pass aware of the shadow stack and prevent it, when optimizing the defensive transformation, to propagate the values read from the shadow stack.

On the other hand, when producing the low-level defensive program, we optimistically apply some optimizations to directly produce an "optimized" program. The selection pass, which occurs just before the generation of RTL code, replaces some instructions (e.g., bitwise or with the immediate value zero) with more specific ones (e.g., a simple move). We apply these transformations on the fly as we produce the low-level defensive code.

```
#include <verasco.h>

int t[10];
int main(void)
{
    unsigned bound = verasco_any_int() & 0xFFFF;
    for (signed i = 0; i < bound; ++i) {
        t[i % 4] = i * i;       /* (t: [0; 12]) */
    }
    return t[0] - 4;
}
```

Figure 8. Example: source code

V. A FULL EXAMPLE

The program shown on Figure 8 writes within a loop some cells of an array t. An analysis at high level can straightforwardly infer that the pointer offset of this memory access is between zero and twelve (as the array contains four-bytes integer values, and the high-level index, because of the modulo operation, is between zero and three). Thus, this access is annotated with (t: [0; 12]), as shown as a comment next to the write instruction. The defensive version of this program implements this annotation as a check that the pointer is between t + 0 and t + 12.

The Figure 9 shows the main loop of this defensive program, after compilation to RTL (and after all optimizations that are applied on this intermediate representation). Here the syntax uses named labels to ease readability. We can notice that the pointer computation — within the first highlighted block (reddish) — appears as a complex sequence of operations; analyzing it is much more difficult than at a higher level. The second highlighted block (blueish) corresponds to the defensive check: the pointer value is computed (in register x5) and compared to the two predicted bounds. If both comparisons succeed, the program execution goes through; otherwise, it jumps to some bogus code at label FAIL (not shown).

Part of the product program corresponding to this example is shown on Figure 10. The loop header is annotated with two invariants; they state that live variables are equal. Registers x4 and x5 correspond to the source variable i and registers x6 and x7 to the source variable bound. Since branching is a critical instruction, there is only one instance of it; but it is preceded by an assertion which ensures that the guards of the two programs are equivalent. The first highlighted sequence

```
LOOP:
if (x2 <u x3) goto BODY else goto END
BODY:
x28 = x2
x30 = x28 >>x 2
x29 = x30 * 4 + 0
x27 = x28 - x29
x5 = t + 0 + x27 * 4
x26 = t + 0
x23 = x26 <=u x5
x25 = t + 12
x24 = x5 <=u x25
x22 = x23 & x24
if (x22 !=u 0) goto OK else goto FAIL
OK:
x14 = x27
x15 = x2 * x2
int32[t + 0 + x14 * 4] = x15
x2 = x2 + 1
goto LOOP
END:
```

Offset computation

Defensive check

Figure 9. Example: high-level instrumentation, compiled to RTL (loop only)

```
LOOP: invariant x4 === x5 ∧ x6 === x7
assert ((x4 <u x6) !=u 0) === ((x5 <u x7) !=u 0)
if x4 <u x6 then BODY else END
BODY:
x16 = x4
x57 = x5
assert x16 === x57
x20 = x16 >>x 2
x61 = x20
x18 = x20 * 4 + 0
x12 = x16 - x18
x14 = x4 * x4
x34 = t + 0 + x12 * 4
x36 = t + 0
x38 = t + 12
x36 = x36 <=u x34
x38 = x34 <=u x38
x36 = x36 & x38
x34 = x36
x59 = x61 * 4 + 0
x55 = x57 - x59
x11 = t + 0 + x55 * 4
x53 = t + 0
x47 = x53 <=u x11
x51 = t + 12
x49 = x11 <=u x51
x45 = x47 & x49
assert (x34 !=u 0) === (x45 !=u 0)
if x34 !=u 0 then OK else FAIL
OK:
x29 = x55
x31 = x5 * x5
assert (t + 0 + x12 * 4) === (t + 0 + x29 * 4)
assert x14 === x31
x4 = x4 + 1
x5 = x5 + 1
goto LOOP
END:
```

Synch. loop header

Synchronous critical computation

Left code

Right code

Synchronous branch

Right code

Synch. memory store

Figure 10. Example: product program

(greenish) corresponds to the two first instructions of the loop body in Figure 9. Since the right shift instruction is a critical one, there is only a single copy of it, guarded by an assertion claiming that the two programs run this instruction on equal values. Then there is a long sequence of non-critical instructions: they correspond to a mix of the instructions computing the array offset, the written value, and the condition for the defensive check. The first instructions come from one program whereas the last ones come frow the other program.

Interestingly, the instructions of both programs are not exactly in the same order. For instance, the highlighted (reddish) lines which compute the written value are in one case before the defensive check and in the other case after this check. This is due to the fact that one program gets optimized before being put in defensive form, but the other program gets optimized after. The generator of verification conditions and the assorted solver are robust enough to handle such interleavings.

Finally, the critical memory access is modeled in the product as two assertions only: the first one ensures that the accessed address is the same; the second one ensures that the written value is the same.

VI. Experimental results

We have implemented in Coq the instantiation of the methodology that we have described in the previous sections; we have in particular proved the theorems shown in sections III and IV. We have also proved that the backend, (i.e., the compilation from RTL to Mach), preserves the validity of the annotations. Thanks to the extraction mechanism of Coq, we have extracted OCaml programs out of our development. The Coq development is about 6.6k lines of specification and over 10k lines of proof, excluding blanks and comments. The parts reused from CompCert and Verasco are not counted. The full development is available at the companion website http://www.irisa.fr/celtique/ext/csf17/.

This section describes some experiments that we have performed with these programs. First, we show that on a selection of diverse C programs we are able to infer, compile and validate the compilation of points-to invariants. The availability of points-to annotations at various levels of the compilation chain enables us to improve significantly other analyses. Thus, we show three use cases of our methodology: a client optimization at RTL level, an analysis of cache-based timing side-channels at pre-assembly level, and an analysis of stack resource usage at pre-assembly level.

A. Lowering points-to facts

Some measurements of execution time of test C programs (up to thousands of lines) are gathered in Figure 11. For each test program, we report its size in terms of number of C#minor instructions, the duration of inferring the annotations (first run of the Verasco analyzer), the duration of checking the safety of the high-level defensive program (second run of Verasco), and the duration of proving the equivalence of the two defensive programs. One cell in the "Check" column contains ∞. This means that the validation of the high-level defensive program

Program	Size	Infer (s)	Check (s)	Equiv (s)
blowfish	177	29.2	32.4	0.01
des	230	2.8	4.9	0.84
donna	1214	515	∞	310
RC4	94	4.6	5.1	0.02
salsa20	342	6.0	10.4	0.56
snow	871	2.7	8.2	0.12
tea	121	3.43	3.9	0.01
core (1)	166	0.05	0.29	0.03
core (2)	142	0.04	0.28	0.03
core (4)	198	0.06	0.35	0.04
aes	1147	38.3	119	137
almabench	266	6.2	24.7	3.5
fft	229	0.02	0.08	0.03
fftw	97	7.0	80.2	3.1
nbody	163	0.88	1.52	0.06
sha3	457	62.5	207	3.1
siphash24	321	0.68	2.1	0.21
random (1)	378	1.57	1.69	0.03
random (2)	1890	23.4	23.5	0.32
random (3)	2836	20.1	24.5	0.24
random (4)	746	10.1	11.2	0.05

Figure 11. Timings

was not possible due to current limitations of the Verasco analyzer.

The timing measurements have been performed on an otherwise idle Linux system on a x86_64 architecture clocked at 2.0 GHz, with 8 Gio of RAM. The figures are the average of ten measurements; variance was observed to be low at the displayed precision.

The first block of lines gathers test cases for the implementations of various cryptographic primitives within the PolarSSL library. The second block reports on test programs from the NaCl cryptography library [15]. The third block lists six programs derived from the CompCert benchmarks; they run numerical computations or cryptographic routines. The fourth block gathers C programs that were randomly generated by the Csmith tool [16].

This table shows that on various C programs, we are able to automatically infer, verify, propagate, and validate at low level, points-to information for every memory access. In most cases, the running times of this full process are affordable (from fractions of seconds up to a few seconds); unfortunately, in some cases, they are rather high (tens or hundreds of seconds). This occurs when there is a lot of aliasing, so that the defensive encoding must enumerate all pointers in the annotation ranges, and also when there are many 64-bit operations. This suggests some optimizations of our implementation.

For instance, the product construction could be aware of the compiler intrinsics; as an example, 64-bit integer addition is represented in RTL as an opaque builtin call and could be considered as an innocuous local operation rather than as an unknown (and maybe critical) system call.

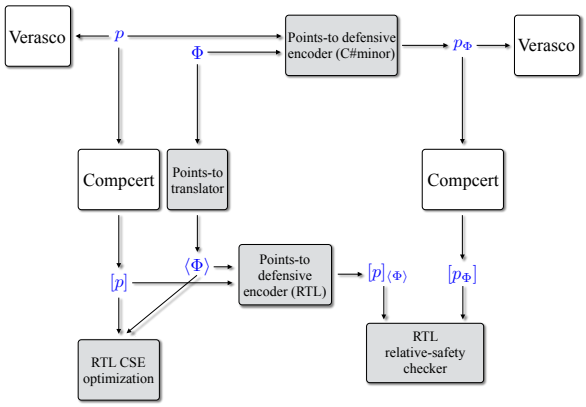

Figure 12. First use case: improving CSE

B. First use case: improving CSE

CSE is an optimization pass that is implemented in particular in the CompCert compiler. It operates on (extended) basic blocks and relies on an intra-procedural value analysis. This optimization attempts to remove redundant computations and memory loads. It is safe to remove a load when, in the same extended basic-block, the same load (i.e., same address, same chunk) occurs before, and no overlapping store happens in-between. Therefore, the efficiency of this optimization (with respect to eliminated redundant loads[3]) lies in the ability to prove that two memory accesses cannot overlap.

The value analysis of CompCert computes some invariants on the RTL program that allows us to prove such facts. However, this analysis has some precision limitations [4]: in particular, guards of conditional branches are ignored. Indeed, it is intra-procedural and cannot infer relational invariants. A more precise analysis can help to improve the disjointedness check in the CSE pass. Indeed, the Verasco analyzer can infer more fine-grained points-to invariants. Thanks to our methodology, we can propagate these invariants to the RTL language, further in the compilation chain, and prove that they are still correct, even after the other optimization passes that are performed before CSE[5].

Figure 12 shows the instantiation of our methodology with CompCert and Verasco, the results can then be used to implement a new CSE pass. Figure 13 illustrates how the precise points-to annotations help significantly to recognize redundant loads in some programs. For each function, the "Size" column lists the static number of loads before the CSE pass, the "CSE" column tells how many of these loads are removed

[3] We are not concerned by the performance of the compiled program, but rather by the fact that the propagated annotations yield more precise information than what can be inferred at low level.

[4] These limitations are understandably justified by other requirements of the compiler such as efficiency and separate compilation.

[5] Namely, tail-call introduction, inlining, renumbering and constant propagation.

Function	Size	CSE	CSE+annot
matmult	29	4	15
snow_keystream_fast	240	32	47
SHA256_Transform	1742	1223	1664
dfft	26	8	10
keccakf	131	25	85
fcontract	74	18	60

Figure 13. The CSE benchmark (numbers of loads)

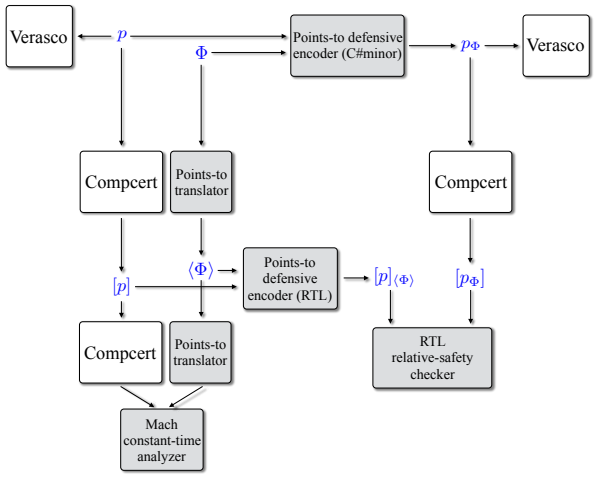

Figure 14. Second use case: cryptographic constant-time

by CompCert's CSE, and the "CSE+annot" column tells how many of these loads are removed with our variant of CSE that relies on annotations rather than on the RTL value-analysis. The *matmult* function performs matrix multiplication; its inner loop has been fully unrolled, so that there are many redundant loads in a single basic block. These loads are interleaved with stores (to write the result), so the intra-procedural value-analysis of CompCert cannot prove that loads and stores are disjoint. On the contrary, the annotations of our methodology can. The other functions feature similar interleavings of loads and stores, and the annotations help to prove that they do not overlap. Considering these selected examples as a whole, the original CSE removes about 60 % of the loads whereas our enhanced CSE removes nearly 85 % of them. A manual inspection of the code shows that this result is close to optimal: hardly any additional load could be eliminated by such an optimization.

C. Second use case: cryptographic constant-time

The second use case is a security analysis that aims at proving that a program is "cryptographically constant-time", a programming discipline used by practitioners to minimize the risks of cache-based timing attacks against cryptographic libraries. Informally, a program achieves cryptographic constant-time if its control flow and sequence of memory accesses are independent of some of its inputs, which users tag as confidential. Previous work by [12] develops an information-flow analysis for cryptographic constant-time, focusing on the Mach intermediate language of CompCert. However, the information-flow analysis is based on a weak points-to analysis, requiring that off-the-shelf implementations from standard cryptographic libraries such as PolarSSL undergo manual rewriting before being analyzed. In addition, the analysis requires that programs are fully inlined, and as a consequence some programs like donna cannot be analyzed with this approach.

We developed an information-flow type system for verifying cryptographic constant-time. Our type system operates on Mach pre-assembly programs[6], and is very similar to [12]: each register and each memory location are given a (flow-sensitive) security level: low (public) or high (secret). Then, the type system performs a data-flow analysis to check that the targets of conditional jumps and memory accesses do not depend on high values. In order to keep track of the security levels of values in memory, we use the points-to information derived from analyzing the C#minor programs using the Verasco analyzer.

Lowering of points-to annotations is justified in two steps: the lowering to RTL is justified by translation validation using the methodology presented in this paper, whereas the lowering from RTL to Mach is justified by a direct proof as illustrated in Figure 14. The results obtained with the methodology of this paper improve on [12] in two directions; first, there is no need to perform code rewriting before analyzing programs. Second, programs that were previously out of reach can be proved to verify cryptographic constant-time. More specifically, we have analyzed the cryptographic programs that appear in the first block of the table in Figure 11, as well as the AES and SHA-3 implementations from CompCert benchmarks. We are able to automatically prove that they verify the constant-time policy, or the stealth constant-time policy, a variant also considered in [12] and inspired by stealth memory [17], [18]. All programs are analyzed in a few milliseconds, except donna whose analysis requires a few seconds.

D. Third use case: resource analysis

Memory usage is another potential source of information leakage. As a consequence, it is important to make sure that memory usage does not depend dramatically on secrets. As a first step in this direction, we have instantiated our methodology to a resource analysis based on stack space usage [13] illustrated in Figure 15. This yields an alternative method to recover the results of [13] (which uses a certification, rather than validation; see below for a longer comparison) and shows that our methodology is not only limited to points-to analysis.

Each function in the analyzed program is annotated with the size of the stack space it uses at assembly level. Given a memory usage bound N that the program is not allowed to exceed, we add a global variable that is initialized to N in the defensive version of the program. Each annotation is then transformed as follows:

[6]Mach is the last intermediate language of CompCert, where programs are represented by lists of low-level instructions operating over machine registers and memory locations.

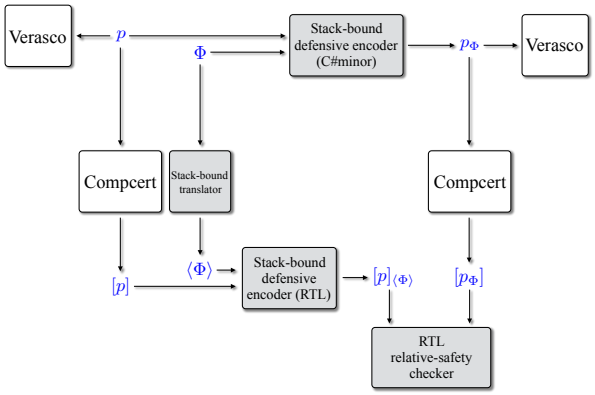

Figure 15. Third use case: resource analysis

File Name	Size	Time (s)	Verified Stack Bound (B)
nbody.c	105	7.81	148
mandelbrot.c	53	0.11	28
mibench/auto/bitcnt_1.c	24	0.03	40
mibench/auto/bitcnt_2.c	20	0.03	40
mibench/auto/bitcnt_3.c	31	0.28	56
mibench/auto/bitstrng.c	37	0.20	136
custom/fact_sq.c	14	0.03	472
custom/sum.c	19	0.03	340

Figure 16. Verified stack-bounds

- We add code at the beginning of each function to test if the corresponding stack space usage is lower than the remaining memory space modeled by the global counter, the counter is then decreased if it is true and the program crashes otherwise.
- We add code at the end of each function to increase back the counter by the appropriate value.

We prove that if the defensive program at high level is validated and if the equivalence checker agrees, then the compiled program will not stack overflow if the available memory space is larger than the given bound. Moreover, we have carried an experimental evaluation of our approach on a subset of examples from [13] and from the MiBench embedded benchmark suite [19], and obtained comparable results.

We briefly compare the two approaches: by using validation rather than verification, we avoid the need to make extensive changes to CompCert and rework the proof of every compilation pass. On the other hand, we note that our bounds are not automatically inferred yet. However, it would be possible to instrument Verasco to obtain the call-stack traces of programs and thus easily infer their stack bounds. Moreover, our method can only verify constant bounds and does not handle symbolic ones yet such as those manually verified in [13]. However, we should be able to accommodate this extension with minimal changes.

Some of our results are reported in Figure 16. For each program, we report its size in terms of number of C#minor instructions and the duration needed for verifying the given bound.

VII. Related work

A. Verified and verifying compilation

Verified compilation is an active area of research whose goal is to build a (machine-checked) proof of compiler correctness. However, most verified compilers, including CompCert [20] and CakeML [21] prove preservation of safety, and cannot be used directly for lowering analysis results. Verified compilers that go beyond preservation of safety exist for specific properties, such as resource consumption [13]. Our approach yields a promising alternative to [13], as mentioned previously.

Verifying compilation enforces properties of target programs through a full static analyzer at target level [22], [23]. In this way, the results of the source-level static analysis are not even used, and so there is no need to develop sound methods for carrying evidence from source to target level. Certifying compilation is typically restricted to safety properties.

Certificate translation [24], [25] is an alternative approach inspired from proof-carrying code [26], in which the compiler comes equipped with an automated translator which maps correctness proofs of the source program into correctness proofs of the target program. However, this approach is hard to implement for realistic languages.

B. Verified static analysis

Early works on verified static analysis consider interval analyses for toy languages [27]–[29]. More recent works target intermediate representations of CompCert. Blazy *et al.* propose a value analysis at RTL level [30], with a naive approach for memory abstraction. [31] verifies a points-to analysis at the same RTL level. Like the current work, they consider points-to information but without numerical information about offsets. [11] operate over source level (C#minor) with a complex memory abstraction [32]. They do not provide translation mechanism, but our current work is based on their static analysis. The CompCert compiler itself [20] has been recently extended with a memory-aware value analysis at RTL level. This analysis is used several times in the CompCert backend for code optimization. However, CompCert is not able to propagate the inferred information. Instead, the analysis is relaunched several times, after each program transformation. This analysis is less precise than Verasco, as shown in section VI-B.

Outside CompCert, several verified static analyses have been proposed for the Java bytecode language. Klein and Nipkow verify a Java bytecode verifier [33]. Cachera *et al.* verify an inter-procedural class analysis [34].

Our work allows us to generate sound points-to information at assembly level. [35] proposes a direct verified static analysis at binary level but for a toy imperative language and memory model. This paper focuses on the hard problem of disassembling self-modifying programs, that we do not consider here.

C. Differential verification by product programs

Approaches built on product program have been considered for different purposes, notably translation validation of compiler optimizations and information-flow security (see for instance [14], [36]–[39]). This approach is also featured in the Symdiff tool [8], [40] which applies a very simple product construction inspired from self-composition, and in the CTverif tool [41] which applies a product construction inspired from cross-products. Handling loops often requires using "mostly synchronous" product constructions (with inference of loop invariants). Approaches built on formalisms that inherently support relational verification include a spate of specialized formalisms tailored to properties such as information flow, continuity, and reliability; see for instance [42]–[44].

Our approach is more closely related to works that focus on regression verification [45], [46] or equivalence checking [47], [48]. None of this work is formally verified.

VIII. Conclusion and perspectives

We have proposed a method that combines in an original way defensive programs and relative safety to validate the results of source-level static analyzes on low-level programs, and instantiated this method with the CompCert compiler and the Verasco C static analyzer. The outcome is a formally verified translation to lower intermediate representations of points-to assertions computed by Verasco.

There are multiple directions for further work. One possible direction is to implement other analyses. As a concrete example, we could use our methodology to bound the number of iteration of each loop. Such an analysis is especially useful for worst case execution time analysis which is necessary for critical systems where programs need to provide an answer in a limited time. Such an analysis can be implemented by annotating each loop by a bound on the number of iterations it must not exceed, and adding code before a loop to initialize a counter to this value and add code at the beginning of a loop to decrement this counter and test that it is still positive. Another direction is to build more general relative-safety checkers and equivalence checkers for RTL or assembly programs. This would have multiple benefits, both for the approach considered in this paper, and in general for a posteriori validation of compiler optimizations, and verification of relational properties of compiled programs.

References

[1] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[2] G. Balakrishnan and T. W. Reps, "WYSINWYX: what you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, 2010. [Online]. Available: http://doi.acm.org/10.1145/1749608.1749612

[3] V. D'Silva, M. Payer, and D. X. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE Computer Society, 2015, pp. 73–87. [Online]. Available: http://dx.doi.org/10.1109/SPW.2015.33

[4] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000. [Online]. Available: http://doi.acm.org/10.1145/353323.353382

[5] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens, "Provably correct inline monitoring for multithreaded java-like programs," *Journal of Computer Security*, vol. 18, no. 1, pp. 37–59, 2010. [Online]. Available: http://dx.doi.org/10.3233/JCS-2010-0365

[6] S. He, S. K. Lahiri, and Z. Rakamarić, "Verifying relative safety, accuracy, and termination for program approximations," in *Proceedings of the 8th NASA Formal Methods Symposium (NFM)*, ser. Lecture Notes in Computer Science, S. Rayadurgam and O. Tkachuk, Eds., vol. 9690. Springer, 2016, pp. 237–254.

[7] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth, "Will you still compile me tomorrow? static cross-version compiler validation," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 191–201.

[8] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *Computer Aided Verification*. Springer, 2012, pp. 712–717.

[9] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: towards usable verification," in *Proceedings of 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, p. 32.

[10] X. Leroy, "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant," in *33rd symposium Principles of Programming Languages*. ACM Press, 2006, pp. 42–54.

[11] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, "A formally-verified C static analyzer," in *Proc. of the 42th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*. ACM, 2015.

[12] G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *ACM SIGSAC Conference on Computer and Communications Security, CCS'14*. ACM, 2014.

[13] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao, "End-to-end verification of stack-space bounds for C programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, p. 30.

[14] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *Formal Methods*, ser. Lecture Notes in Computer Science, M. J. Butler and W. Schulte, Eds., vol. 6664. Springer, 2011, pp. 200–214.

[15] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, 2012, pp. 159–176.

[16] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 283–294.

[17] U. Erlingsson and M. Abadi, "Operating system protection against side-channel attacks that exploit memory latency," Microsoft Research, Tech. Rep. MSR-TR-2007-117, 2007.

[18] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: system-level protection against cache-based side channel attacks in the cloud," in *USENIX Security 2012*. Berkeley, CA, USA: USENIX Association, 2012, pp. 11–11.

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15

[20] X. Leroy, "A formally verified compiler back-end," *J. Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[21] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "Cakeml: a verified implementation of ML," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14,*

San Diego, CA, USA, January 20-21, 2014, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 179–192. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535841

[22] G. C. Necula and P. Lee, "The design and implementation of a certifying compiler," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, J. W. Davidson, K. D. Cooper, and A. M. Berman, Eds. ACM, 1998, pp. 333–344.

[23] G. C. Necula, "Translation validation for an optimizing compiler," in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, M. S. Lam, Ed. ACM, 2000, pp. 83–94.

[24] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk, "Certificate translation for optimizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 5, 2009. [Online]. Available: http://doi.acm.org/10.1145/1538917.1538919

[25] G. Barthe and C. Kunz, "An abstract model of certificate translation," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 4, p. 13, 2011. [Online]. Available: http://doi.acm.org/10.1145/1985342.1985344

[26] G. C. Necula, "Proof-carrying code," in *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, P. Lee, F. Henglein, and N. D. Jones, Eds. ACM Press, 1997, pp. 106–119.

[27] D. Pichardie, "Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés," Ph.D. dissertation, U. Rennes 1, 2005.

[28] Y. Bertot, "Structural abstract interpretation: A formal study using Coq," in *Language Engineering and Rigorous Software Development, LerNet Summer School*. Springer, 2008, pp. 153–194.

[29] T. Nipkow, "Abstract interpretation of annotated commands," in *ITP*, ser. LNCS, vol. 7406. Springer, 2012, pp. 116–132.

[30] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie, "Formal verification of a C value analysis based on abstract interpretation," in *SAS*, ser. LNCS, vol. 7935. Springer, 2013, pp. 324–344.

[31] X. Leroy and V. Robert, "A formally-verified alias analysis," in *Proc. of CPP 2012*, ser. LNCS, vol. 7679. Springer, 2012, pp. 11–26.

[32] S. Blazy, V. Laporte, and D. Pichardie, "An abstract memory functor for verified c static analyzers," in *Proc. of the 21st ACM SIGPLAN Int. Conference on Functional Programming (ICFP 2016)*. ACM, 2016.

[33] G. Klein and T. Nipkow, "A machine-checked model for a Java-like language, virtual machine, and compiler," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 4, pp. 619–695, 2006.

[34] D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu, "Extracting a data flow analyser in constructive logic," *Theor. Comput. Sci.*, vol. 342, no. 1, pp. 56–78, 2005.

[35] S. Blazy, V. Laporte, and D. Pichardie, "Verified abstract interpretation techniques for disassembling low-level self-modifying code," *Journal of Automated Reasoning*, vol. 56, no. 3, pp. 283–308, 2016.

[36] G. Barthe, P. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *Proceedings of 17th IEEE Computer Security Foundations Workshop, CSFW'04*, R. Foccardi, Ed., 2004, pp. 100–114.

[37] A. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *Security in Pervasive Computing*, D. Hutter and M. Ullmann, Eds., vol. 3450. Springer, 2005, pp. 193–209, preliminary version in the informal proceedings of WITS'03.

[38] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Static Analysis Symposium*, C. Hankin and I. Siveroni, Eds., vol. 3672. Springer, 2005, pp. 352–367.

[39] A. Zaks and A. Pnueli, "CoVaC: Compiler Validation by Program Analysis of the Cross-Product," in *Formal Methods*, ser. Lecture Notes in Computer Science, J. Cuéllar, T. S. E. Maibaum, and K. Sere, Eds., vol. 5014. Springer, 2008, pp. 35–51.

[40] S. K. Lahiri, R. Sinha, and C. Hawblitzel, "Automatic rootcausing for program equivalence failures in binaries," in *Computer Aided Verification*. Springer, 2015, pp. 362–379.

[41] J. C. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[42] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity analysis of programs," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: ACM, 2010, pp. 57–70.

[43] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: reliability- and accuracy-aware optimization of approximate computational kernels," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, A. P. Black and T. D. Millstein, Eds. ACM, 2014, pp. 309–328.

[44] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, pp. 165–179.

[45] B. Godlin and O. Strichman, "Regression verification: proving the equivalence of similar programs," *Softw. Test., Verif. Reliab.*, vol. 23, no. 3, pp. 241–258, 2013.

[46] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. ACM, 2014, pp. 349–360. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642987

[47] S. Kundu, Z. Tatlock, and S. Lerner, "Proving optimizations correct using parameterized program equivalence," vol. 44, no. 6, pp. 327–337, 2009.

[48] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven equivalence checking," in *OOPSLA*. ACM Press, 2013, pp. 391–406.