



# Towards Effective Preservation of Robust Safety Properties

Matteo Busi  
Computer Science Dept., Pisa Univ.  
Pisa, Italy  
matteo.busi@di.unipi.it

Pierpaolo Degano  
IMT School for Advanced Studies &  
Computer Science Dept., Pisa Univ.  
Pisa, Italy  
pierpaolo.degano@unipi.it

Letterio Galletta  
IMT School for Advanced Studies  
Lucca, Italy  
letterio.galletta@imtlucca.it

## ABSTRACT

Secure compilation investigates when compilation chains preserve security properties. Over the years, different formal criteria and proof techniques have been put forward for proving a compiler secure. However, these proposals require a lot of manual effort by compiler designers. This paper introduces a formal approach to mechanically support these efforts. We focus on the specific class of robust safety properties and we propose a translation validation approach that leverages program analysis techniques to check that a compilation run preserves security.

## 1 INTRODUCTION

Secure compilation puts together techniques from different fields, e.g., security, programming languages, formal verification and hardware architectures, to devise compilation chains that protect various aspects of software and eliminate security vulnerabilities [16]. Besides translating *source programs* into *target* ones at lower level, a secure compiler should also provide mitigation against possible attacks and offer mechanisms to enforce secure interoperability between code written in safe and unsafe languages. An important feature of such compilers is to ensure that the security properties at the source level are fully preserved at the target level or, dually, that all the attacks that can be carried out at the target have counterparts at the source level. This property enables us to establish programs secure at the source level without caring much about what happens at low level, where the overwhelming number of details make reasoning far harder.

Over the years different *formal criteria* and *proof techniques* have been proposed to prove a compiler secure with respect to a class of security properties. The classical criterion of *fully abstract compilation* [1] defines a compiler secure when it preserves and reflects the equivalence of behaviors between the original and compiled programs under any *untrusted* context of execution. The proof technique of *constant-time simulation* [7, 8, 32] has been used to prove the preservation of secret independent timing as required for “constant-time” cryptographic implementations. More recently, different notions of *robust property preservation*, i.e., the preservation of security properties like trace properties or hyperproperties in

arbitrary adversarial contexts, have been offered as secure compilation principles [3], the proofs of which resort to existing proof techniques, e.g., backtranslation [26, 28].

Although these approaches are viable for proving that a compilation chain preserves a given class of security properties, their proofs are still hard challenges that require demanding proof engineering and manual efforts by compiler designers. This paper lays the basis for *mechanically* checking the security preservation in compilers. In particular, we focus on *robust safety properties* [3], we generalize *translation validation* [29] and couple it with program analysis to check their preservation after compilation.

Translation validation is a standard technique to automatically check the correctness of the compilation of a *single* program, rather than proving the compiler correct for them *all*. Roughly, it tries to build a suitable simulation between the given source program **P** and its compiled version **P**: if such a simulation exists and can be effectively computed, then the compilation of **P** is correct (for readability we use *blue, sans-serif* for the elements of the source language **S**, *red, bold* for those of the target language **T**, and black for the common ones). However, when it comes to security properties it has been proved that building the required simulation is decidable only if **P** is finite-state [14].

To overcome this limitation, we leverage program analysis techniques, and propose *secure translation validation (STV)* as a formal framework to derive a (approximated) procedure to check robust preservation of safety properties. The idea is to detect if the compiled version **P** violates the safety properties of its source counterpart **P** when it is about to run in a *given* environment (modeled as a *target context*). More precisely, *STV* detects if a compilation of **P** preserves all the safety properties under a given target context at *link time*, i.e., when the compiled program is plugged into its execution context. We argue that this is the right time, as one typically wants some security guarantees on a module, e.g., a library, *before* launching the used program, so it is *not too early*. Neither it is *too late*, since linking the same program to different contexts may give different security guarantees.

Intuitively, our *STV* framework works as follows. First we take two program analyses, one for the source language **S** and one for the target language **T**. The analysis for **S** has to *under-approximate* the behavior of the source program under analysis **P**, while that of **T** has to *over-approximate* the behavior of its target counterpart **P**. Under-approximation at the source and over-approximation at the target are crucial to guarantee the absence of false negatives, i.e., target programs classified as secure when they are not. However, we cannot guarantee the absence of false positives (safe programs signaled as insecure), which is the typical price to pay when using static analysis – yet we firmly stay in the safe side. At link time we

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SAC '22, April 25–29, 2022, Virtual Event

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-8713-2/22/04...\$15.00

<https://doi.org/10.1145/3477314.3507084>


plug the compiled program  $P$  into the target context  $C$  to obtain  $C[P]$ ; we over-approximate its behavior using the static analysis for  $T$ ; and we verify that the compilation process broke none of the properties of interest.

Our verification procedure exploits the backtranslation proof technique as a verification tool. Indeed, starting from  $C$  we build a source context  $C$  witnessing the presence of all the behaviors of  $C[P]$  at the source level, and we then check that the under-approximation associated with  $C[P]$  includes the over-approximation of  $C[P]$ . Needless to say, the precision and efficacy of our proposal heavily depends on the precision of the program analyses used to approximate program behavior, as well as on the analysis domains that should permit to efficiently test inclusion of behaviors.

To show the viability of our proposal, we apply our *STV* framework to a use case inspired by secure compartmentalization: we consider a source language equipped with compartments that ensure isolation, and we compile it to an assembler-like language that provides no isolation. We execute the operational semantics of programs to compute the required under-approximation at the source level and history expressions [9, 10] (aka trace effects [33]) to over-approximate the behavior at the target. Then, we show how *STV* exploits the results of these analyses and of the backtranslation-based procedure to check that the resulting target program preserves isolation when plugged into an adversarial context.

In summary, the main contributions of this paper are:

- we introduce *STV* as a mean for enabling the mechanical check of the preservation of robust safety properties; it generalizes translation validation and exploits program analysis techniques to guarantee decidability;
- we provide a theoretical framework for *STV* that is general and consists of two equivalent characterizations (Sections 4 and 5). The first is more abstract and suitable to prove correspondence with the adopted secure compilation criterion of [3]; the second one is more algorithmic and suitable for verification;
- we define an algorithm for *STV* by deriving it from the second characterization above; also we show how backtranslation can be used not only as a manual proof technique to establish that a compiler is secure, but also to support an automatic verification procedure (Section 5);
- finally, we apply *STV* to a use case inspired by that of [17] to check the preservation of isolation at the target level (Section 6). As an aside result we provide, to the best of our knowledge, the first procedure to extract history expressions from an assembler-like language.

Most of our formal development has been mechanized using the Coq proof assistant (marked ) and is available online at <https://github.com/matteobusi/stv/tree/sac2022> as supplementary material, together with the rest of our pencil and paper proofs.

## 2 BACKGROUND

*Safety properties.* The *security* of a system, either hardware or software, is defined in a variety of ways, depending on the level of detail one is interested in and on the *security policies* to consider. If not stated otherwise, we assume  $W \in \text{Whole}$  to be a *whole program*, i.e., a complete and executable one.

When defining security policies, it is common to consider the execution traces of programs, namely their *behavior*. The behavior of a program  $W$  is a set of finite and infinite traces over a set of *observables* (or *events*)  $\Sigma$ . Intuitively, an observable encodes what an external observer learns about  $W$  by looking at a single execution step. Each step is specified by a labeled transition function  $\rightarrow$ , and traces through its reflexive and transitive closure  $\rightarrow^*$ . Typical examples of observables include I/O operations, run-time errors, termination, or divergence [19]. Formally, the behavior of  $W$  is  $\text{beh}(W) \subseteq \Sigma^* \cup \Sigma^\omega$ .

Building on this notion of behavior one easily defines security policies as *trace properties* [6, 18]. Intuitively, trace properties encode the set of behaviors *allowed* for a given program. Below we focus on a class of trace properties called *safety properties*. A safety property  $\pi_S$  requires that something *bad and finitely observable* ( $m$ ) never happens [12]:

$$\pi_S \in \text{Safety} \Leftrightarrow (\forall t \in \Sigma^\omega. t \notin \pi_S \Rightarrow \exists m \in \Sigma^*. m \leq t \wedge (\forall t' \in \Sigma^\omega. m \leq t' \Rightarrow t' \notin \pi_S))$$

where  $m \leq t$  means that  $m$  is a prefix of  $t$ . Attackers are implicitly considered in the definition above. Usually they are modeled as *contexts*, i.e., programs with a hole, in symbols  $C$  to be filled with a victim program  $P$ , so giving rise to whole programs  $C[P]$  to be analyzed. Also, say that a source program *robustly* satisfies a (family of) safety properties if it does satisfy them against any attacker. For further details we refer the interested reader to [6, 12, 18].

*Secure compilation.* It is common to define compilers as follows:

**Definition 2.1.** A compiler  $\llbracket \cdot \rrbracket_T^S$  is a function mapping programs in a *source language*  $S$  to programs in a *target language*  $T$ .

We now recall relevant notions of *secure compilation*. Roughly, a secure compiler *preserves* the security of programs it compiles, i.e., if a program satisfies a security policy *before* compilation, then it must satisfy it *after* compilation. Here, we focus on the *robust preservation of safety properties* [3]: a compiler  $\llbracket \cdot \rrbracket_T^S$  is such if it compiles a source program  $P$ , which robustly satisfies a (family of) safety properties, to a program  $\llbracket P \rrbracket_T^S$  that still robustly satisfies the same family of safeties. Formally:

**Definition 2.2** (*Robust safety property preservation (RSP)* [3]). The compiler  $\llbracket \cdot \rrbracket_T^S$  *robustly preserves safety properties* (written  $\llbracket \cdot \rrbracket_T^S \in \text{RSP}$ ) iff  $\forall P, \pi \in \text{Safety}. (\forall C. C[P] \models \pi) \Rightarrow (\forall C. C[\llbracket P \rrbracket_T^S] \models \pi)$ .

Showing that a compiler adheres to RSP may not be at all trivial. To simplify proofs, Abate et al. [3] proposed a *property-less* characterization of secure compilers that is equivalent to specialized versions of Definition 2.2, without mention of safety properties they refer to. Formally:

**Definition 2.3** (*Robustly safe compiler (RSC)* [3]). A compiler  $\llbracket \cdot \rrbracket_T^S$  is *robustly safe* (written  $\llbracket \cdot \rrbracket_T^S \in \text{RSC}$ ) iff

$$\forall P, C, m. (C[\llbracket P \rrbracket_T^S] \rightsquigarrow m) \Rightarrow (\exists C. C[P] \rightsquigarrow m),$$

where  $C[P] \rightsquigarrow m \triangleq \exists t \geq m. t \in \text{beh}(C[P])$ .

We refer the reader to [2, 3, 27] for further details and for the proof of the correspondence between the notions of RSP and RSC.

*Translation validation.* Pnueli et al. [29] introduced translation validation to automatically verify the correctness of compilers, without proving correctness again from scratch, or adapting existing proofs after changes in the compiler code base. Since it has been proposed, this technique has been widely employed [22, 23, 31].

At a very high-level, the approach works as follows. A program  $P$  is compiled to a target program  $\llbracket P \rrbracket_T^S$ : if the *analyzer* proves that  $\llbracket P \rrbracket_T^S$  implements (i.e., refines) the original program  $P$ , then it generates a proof script and exits successfully; otherwise it produces an input on which  $P$  and  $\llbracket P \rrbracket_T^S$  behave differently. Remarkably, translation validation performs a correctness verification after *each* run of the compiler, so it builds *no a priori* proof of correctness.

The process is automatized in two steps. First a homogeneous semantics to both  $S$  and  $T$  is given through transition systems; then an appropriate *syntax-based simulation* is defined between the two transition systems that is amenable to be built mechanically.

*Program Analysis.* Analysing the behavior of programs has been a main concern ever since, as well as the quest for formal methods to ensure their correctness. A major, unsolvable problem is that any analysis has to face the halting problem.

Some proposals simply ignore this issue and check *some* of the runs of a program, so *under-approximating* its behavior: errors can be found, but you can neither claim there are none left out nor that you can stop analysing the behaviour after some point. A typical example is *program testing*.

Other techniques, namely *static analysis*, instead get rid of the halting problem by soundly *over-approximating* program behaviors, i.e., by *finitely* building a set containing *all* its runs and also others that will never occur at run time. Clearly, if this over-approximation enjoys a given property, so does the program it abstracts. However, the price to pay are false alarms. Among this class we find many largely used techniques, all firmly based on formal language semantics. They include type checking, data- and control-flow analysis, and abstract interpretation; a comprehensive survey is in [24].

For applying our proposal, we use a *type and effect system* that leverages inference to extract types and finitely represent the changes on the program state caused by execution steps. In our case, the effects are a sort of basic process algebra, called *history expressions* [9], that finitely represent a set of runs. History expressions have been used to verify if a program satisfies a given safety property [10]. In particular, they enable using the well known automata-based model checking framework [34]: verification consists of checking the intersection between (the language of) a finite state automaton (the property) and a pushdown automaton (the model) [10].

### 3 AN OVERVIEW OF STV

We now intuitively describe our approach via a running example. Consider our source language to be  $\mu C$ , an imperative language inspired by the one of [17] with buffers and *compartments*. Each compartment must implement an *interface*, which defines its public functions. The interfaces serve as an encapsulation mechanism because only their public functions are callable by other compartments. The target language is  $\mu A$ , an assembly language endowed with modules. Each module defines a list of functions and carries

the name of the interface it implements but, differently from the source, the target does not enforce encapsulation. Each function is composed by a list of labeled basic blocks, and we assume that each function has an entry point  $\ell_{\text{entry}}$  that labels its initial basic block.

Consider now the following program  $P$ , defined as a *Main* compartment where, as usual, the function *main* is its entry point (assume  $\text{IMain} = \{\text{main}\}$ ):

```

1 | comp Main : IMain {
2 |   fun main {
3 |     if Main.ok (arg[0])
4 |     then { Main.store(arg[0]) }
5 |     else { Logger.err(arg[0]) }
6 |   }
7 |
8 |   fun ok { /* Omitted, checks if a value is valid */ }
9 |
10 |  fun store { storage[0] := arg[0]; }
11 | }
```

Intuitively, the above program first checks if the parameter value  $v$  (stored in the  $0^{\text{th}}$  position of the buffer *arg*) passed to *main* is valid (Line 3). The value  $v$  is stored in the *storage* buffer (Line 4) if it is valid; otherwise, a call to the *attacker-provided* compartment *Logger* logs the error. The compiled version of the program above is  $P = \llbracket P \rrbracket$  (we comment the omitted details):

```

1 | module Main : IMain {
2 |   define main {
3 |      $\ell_{\text{entry}}$ :
4 |       // - store  $r_{\text{com}}$  in arg[0] position
5 |       // -  $r_{\text{aux}}$  the current argument on the stack
6 |       // - store the old stack pointer
7 |       // - clear all the registers except  $r_{\text{com}}$ 
8 |       call Main.ok  $\ell_{\text{okret}}$ 
9 |      $\ell_{\text{okret}}$ :
10 |      // upon return:
11 |      // - restore the old stackpointer
12 |      // - pop the old argument from the stack
13 |      bnz  $r_{\text{com}}$   $\ell_1$   $\ell_2$ 
14 |      $\ell_1$ :
15 |      // setup the call
16 |      call Main.store  $\ell_{\text{end}}$ 
17 |      $\ell_2$ :
18 |      // setup the call
19 |      call Logger.err  $\ell_{\text{end}}$ 
20 |      $\ell_{\text{end}}$ :
21 |      nop
22 |      jmp  $\ell_{\text{trailer}}$ 
23 |      $\ell_{\text{trailer}}$ :
24 |      // - store current stack pointer value
25 |      // - clear all the registers except  $r_{\text{com}}$ 
26 |      ret
27 |   }
28 |
29 |   define ok { ... }
30 |
31 |   define store { ... }
32 | }
```

where the register  $r_{\text{com}}$  is used for passing and returning parameters.

Suppose now to link  $P$  with the following implementation of *Logger*, provided by a malicious party:

```

1 | module Logger : ILogger {
2 |   define err {
3 |      $\ell_{\text{entry}}$ :
4 |       // call Main.store directly
5 |       // i.e. pass the value in  $r_{\text{com}}$  to Main.store
```

```

6 |      call Main.store  $\ell_{trailer}$ 
7 |       $\ell_{trailer}:$ 
8 |      ret
9 |    }
10| }

```

Intuitively, the above code invokes **Main.store** skipping the sanity check provided by the function **Main.ok**, and enables data corruption. This attack is possible because of the lack of encapsulation of the target language.

Of course, this attack is not possible at the source and can be detected by our *STV* framework. To do that, we first extract from the target program text a history expression that abstractly represents the behavior of **P** when plugged in a context **C** containing the malicious logger above. This abstract behavior also records the call to the function **Main.store**. Secondly, from the target context **C** we derive a source counterpart **C** via backtranslation. Finally, we check whether the behavior of **P** plugged in **C** is included in that of **P** plugged in **C**. This is not the case because the encapsulation property of  $\mu\mathbf{C}$  is violated by any source-level context that calls **Main.store**, which is private to the compartment **Main**. Therefore, it is *not secure* to execute the compiled program **P** with the malicious logger. In Section 6 we detail the steps sketched above.

## 4 GENERAL FRAMEWORK

In this section we lay the basis for our *secure translation validation* (*STV*) approach, formalizing the intuitions discussed above.

We assume as given a compiled program which is about to be linked (e.g., by a static linker or a program loader) with a given context (e.g., its execution environment). We want to mechanically prove that it is safe to execute this program, by showing that any attack in the target can also be carried out in the source.

For our purposes, a (source or target) programming language is defined as follows:

**Definition 4.1** (*Programming language* 🐼). A programming language  $L$  is a 5-tuple  $(Whole, Partial, Ctx, \cdot[\cdot], \cdot \xrightarrow{\cdot})$  where:

- (1) *Whole* is the set of whole programs;
- (2) *Partial* is the set of partial programs;
- (3) *Ctx* is the set of contexts;
- (4)  $\cdot[\cdot] : Ctx \rightarrow Partial \rightarrow Whole$  is the linking operator;
- (5)  $\cdot \xrightarrow{\cdot} : Whole \rightarrow \Sigma \rightarrow Whole$  is a labeled transition *function*, where  $\Sigma$  is a (finite) set of observables.

Intuitively, the linking operator completes a partial program with the needed information from the context and produces a whole program, while the transition function describes the observable execution of whole programs. We specialize the notion of behavior of a whole program  $W$  as the set of all the prefixes of its traces, i.e.,  $beh(W) \triangleq \{t \in \Sigma^* \mid \exists W'. W \xrightarrow{t}^* W'\}$ , where  $*$  denotes the reflexive and transitive closure.

Recall that we assume a compiler  $\llbracket \cdot \rrbracket$  from a source language **S** to a target language **T**. Below, we formalize our notion of *STV* in the case of safety properties. The notion of  $STV_{RSP}$  states when a target program can be executed securely. Intuitively, in the definition below we write  $P \vdash_P^C STV_{RSP}$  whenever the target program **P** satisfies a safety property  $\pi$  under the *given* target context **C**, provided that the source program **P** does in *any* source context. Perhaps

surprisingly, **P** is *not* required to be the compiled version of **P**, but imposing  $P = \llbracket P \rrbracket$  for a given  $\llbracket \cdot \rrbracket$  would allow using information about  $\llbracket \cdot \rrbracket$  to prove  $STV_{RSP}$ . We do not want this, because it would make the compiler part of the *trusted computing base*. Formally:

**Definition 4.2** (*STV robust safety property preservation* 🐼). A target program **P** can be executed securely in the target context **C** w.r.t. a source program **P** (written  $P \vdash_P^C STV_{RSP}$ ) iff

$$\forall \pi \in \text{Safety}. (\forall C. C[P] \models \pi) \Rightarrow C[P] \models \pi.$$

Similarly to the principles of [3], we have the following *criterion* that does not explicitly mention the class of all the safety properties:

**Definition 4.3** (*STV robustly safe compiler* 🐼). A program **P** satisfies the *STV criterion for safety properties* when linked with the context **C** w.r.t. **P** (written  $P \vdash_P^C STV_{RSC}$ ) iff

$$\forall t. t \in beh(C[P]) \Rightarrow \forall m \leq t. (\exists C, t'. t' \in beh(C[P]) \wedge m \leq t').$$

The criterion above is equivalent to Definition 4.2:

**THEOREM 4.4** ( $STV_{RSP} \Leftrightarrow STV_{RSC}$  🐼). For any target program **P**, target context **C** and source program **P**:

$$P \vdash_P^C STV_{RSP} \Leftrightarrow P \vdash_P^C STV_{RSC}.$$

Finally, the theorem below assess that our definitions above actually correspond to those by Abate et al. [3]. More precisely, the first item shows that our definition of  $STV_{RSP}$  is equivalent to that of *RSP*. Similarly, the second one highlights the correspondence between  $STV_{RSC}$  and *RSC* (Definition 2.3):

**THEOREM 4.5.** (🐼) Let  $\llbracket \cdot \rrbracket$  be a compiler from **S** to **T**.

- $STV_{RSP} \Leftrightarrow RSP : (\forall C, P. \llbracket P \rrbracket \vdash_P^C STV_{RSP}) \Leftrightarrow \llbracket \cdot \rrbracket \in RSP;$
- $STV_{RSC} \Leftrightarrow RSC : (\forall C, P. \llbracket P \rrbracket \vdash_P^C STV_{RSC}) \Leftrightarrow \llbracket \cdot \rrbracket \in RSC.$

## 5 AN EFFECTIVE FRAMEWORK

The formalization of the previous section is comprehensive, but it is hard to derive from it an algorithm for checking whether a compiler is  $STV_{RSC}$  since the set of all traces of a program is in general undecidable. Program analysis comes to our rescue. To fit in it, we introduce two variants of the notion of *abstract STV* and put forward a few conditions for their applicability.

Roughly, an *analysis* is a mapping between a language and an *abstract* version of it:

**Definition 5.1** (*Analysis* 🐼). An *analysis*  $(\llbracket \cdot \rrbracket)_{\mathbb{L}}^L$  from the (concrete) language  $L$  to the (abstract) language  $\mathbb{L}$  consists of three maps  $(\alpha_{Partial}, \alpha_{Ctx}, \alpha_{Whole})$ , where

- $\alpha_{Partial}$  maps partial programs of  $L$  to those of  $\mathbb{L}$ ;
- $\alpha_{Ctx}$  maps contexts of  $L$  to those of  $\mathbb{L}$ ; and
- $\alpha_{Whole}$  maps whole programs of  $L$  to those of  $\mathbb{L}$ .

From now onward we will abuse the notation and feel free to use  $(\llbracket \cdot \rrbracket)_{\mathbb{L}}^L$  in place of the relevant map. For instance, if  $C$  is a context,  $(\llbracket C \rrbracket)_{\mathbb{L}}^L$  corresponds to  $\alpha_{Ctx}(C)$ . Furthermore, assume as given two concrete languages **S** and **T**, their abstract counterparts  $\mathbb{S}$  and  $\mathbb{T}$ , and the analyses  $(\llbracket \cdot \rrbracket)_{\mathbb{S}}^S$  and  $(\llbracket \cdot \rrbracket)_{\mathbb{T}}^T$ .



We now characterize the analyses that we feel sensible here: those that enjoy (a non-empty combination of) *soundness*, *completeness*, and *linearity*. Intuitively, an analysis is *sound* if the behavior of a given program  $W$  is *included in* that of its abstract version  $\langle\langle W \rangle\rangle_{\mathbb{L}}^L$ . Dually, *complete analyses* are such that the behavior of  $W$  *includes* that of  $\langle\langle W \rangle\rangle_{\mathbb{L}}^L$ . Finally, an analysis is *linear* if the behavior of  $\langle\langle C[P] \rangle\rangle_{\mathbb{L}}^L$  equals the behavior of  $\langle\langle C \rangle\rangle_{\mathbb{L}}^L$  and  $\langle\langle P \rangle\rangle_{\mathbb{L}}^L$  linked together. Formally:

**Definition 5.2** (Sound 🐦, complete 🐦, and linear 🐦 analyses). An analysis  $\langle\langle \cdot \rangle\rangle_{\mathbb{L}}^L$  is

- *Sound* iff  $\forall C, P. \text{beh}(C[P]) \subseteq \text{beh}(\langle\langle C[P] \rangle\rangle_{\mathbb{L}}^L)$ ;
- *Complete* iff  $\forall C, P. \text{beh}(C[P]) \supseteq \text{beh}(\langle\langle C[P] \rangle\rangle_{\mathbb{L}}^L)$ ;
- *Linear* iff  $\forall C, P. \text{beh}(\langle\langle C[P] \rangle\rangle_{\mathbb{L}}^L) = \text{beh}(\langle\langle C \rangle\rangle_{\mathbb{L}}^L \langle\langle P \rangle\rangle_{\mathbb{L}}^L)$ .

Exploiting the above notions, we define  $aSTV_{RSC}$ , an abstract version that implies the criterion  $STV_{RSC}$  as follows: we first compute the source context through backtranslation; then we link the result with the source program; and finally we perform the source analysis on the resulting whole program. Note that we assume the source program to be available at *link time*, as required in our setting.

**Definition 5.3** ( $aSTV_{RSC}$  🐦). A target program  $P$  satisfies the *abstract STV criterion for safety properties* when linked with  $C$  w.r.t.  $P$  (written  $P \vdash_P^C aSTV_{RSC}$ ) iff

$$\forall t. t \in \text{beh}(\langle\langle C \rangle\rangle_{\mathbb{T}}^T \langle\langle P \rangle\rangle_{\mathbb{T}}^T) \Rightarrow (\exists \langle\langle \cdot \rangle\rangle_{\mathbb{S}}^S. t \in \text{beh}(\langle\langle \langle\langle C \rangle\rangle_{\mathbb{T}}^T \langle\langle P \rangle\rangle_{\mathbb{T}}^T \rangle\rangle_{\mathbb{S}}^S)).$$

Note that one could do the same for the target analysis. However, in our setting it is highly unlikely that the target context and the target program can be analysed together *before* linking time.

We can easily prove that this abstract notion of STV implies the “behavioral” ones of Definitions 4.2 and 4.3:

**THEOREM 5.4** ( $aSTV_{RSC} \Rightarrow STV_{RSC}$  🐦). If  $\langle\langle \cdot \rangle\rangle_{\mathbb{S}}^S$  is complete,  $\langle\langle \cdot \rangle\rangle_{\mathbb{T}}^T$  is sound and linear, then  $\forall C, P, P. P \vdash_P^C aSTV_{RSC} \Rightarrow P \vdash_P^C STV_{RSC}$ .

**COROLLARY 5.5** ( $aSTV_{RSC} \Rightarrow STV_{RSP}$  🐦). If  $\langle\langle \cdot \rangle\rangle_{\mathbb{S}}^S$  is complete,  $\langle\langle \cdot \rangle\rangle_{\mathbb{T}}^T$  is sound and linear, then  $\forall C, P, P. P \vdash_P^C aSTV_{RSC} \Rightarrow P \vdash_P^C STV_{RSP}$ .

However, the definition of  $aSTV_{RSC}$  may be too demanding since backtranslation depends on the information about a specific trace of the abstracted target program to build the wanted source context. If such a dependency is not acceptable (e.g., for performance reasons) we can change the definition above as follows, where  $\langle\langle \cdot \rangle\rangle_{\mathbb{S}}^S$  now maps contexts of  $\mathbb{T}$  into contexts of  $\mathbb{S}$ :

**Definition 5.6** ( $aSTV_{TI-RSC}$  🐦). A target program  $P$  satisfies the *abstract trace-independent STV criterion for safety properties* when linked with  $C$  w.r.t.  $P$  (written  $P \vdash_P^C aSTV_{TI-RSC}$ ) iff

$$\exists \langle\langle \cdot \rangle\rangle_{\mathbb{S}}^S. \forall t. t \in \text{beh}(\langle\langle C \rangle\rangle_{\mathbb{T}}^T \langle\langle P \rangle\rangle_{\mathbb{T}}^T) \Rightarrow t \in \text{beh}(\langle\langle \langle\langle C \rangle\rangle_{\mathbb{T}}^T \langle\langle P \rangle\rangle_{\mathbb{T}}^T \rangle\rangle_{\mathbb{S}}^S).$$

Crucially, this new notion implies the old one:

**THEOREM 5.7** ( $aSTV_{TI-RSC} \Rightarrow aSTV_{RSC}$  🐦). For any  $C, P, P$ :

$$P \vdash_P^C aSTV_{TI-RSC} \Rightarrow P \vdash_P^C aSTV_{RSC}.$$

Moreover, note that if  $\langle\langle \cdot \rangle\rangle_{\mathbb{S}}^S$  is complete, and  $\langle\langle \cdot \rangle\rangle_{\mathbb{T}}^T$  is linear and sound, then the abstract trace-independent STV implies the behavioral STV principle of Definition 4.2.

---

### Algorithm 1 Pseudocode for the STV algorithm 🐦.

---

**function** SAFETORUN( $C \in \mathbb{T}_{Ctx}, P \in \mathbb{S}_{Par}, P \in \mathbb{T}_{Par}$ )

**Requires:**

- (1)  $\subseteq$  on abstract behaviors to be computable
- (2)  $\langle\langle \cdot \rangle\rangle_{\mathbb{S}}^S$  to be an analysis from  $\mathbb{S}$  to  $\mathbb{S}$
- (3)  $\langle\langle \cdot \rangle\rangle_{\mathbb{T}}^T$  to be an analysis from  $\mathbb{T}$  to  $\mathbb{T}$
- (4)  $b: \mathbb{T}_{Ctx} \rightarrow \mathbb{S}_{Ctx}$  to be a partial backtranslation

**if**  $b(C)$  is undefined **then**

**return** (MAYBE\_UNSAFE,  $\perp$ )

**else**

**if**  $\text{beh}(\langle\langle C \rangle\rangle_{\mathbb{T}}^T \langle\langle P \rangle\rangle_{\mathbb{T}}^T) \subseteq \text{beh}(\langle\langle b(C) \rangle\rangle_{\mathbb{S}}^S \langle\langle P \rangle\rangle_{\mathbb{S}}^S)$  **then**

**return** (SAFE,  $\perp$ )

**else**

**return** (MAYBE\_UNSAFE,  $b(C)$ )

**end if**

**end if**

**end function**

---

*An algorithm for STV.* We now convert Definition 5.6 into an effective procedure, described in Algorithm 1. Assume inclusion between behaviors of programs in  $\mathbb{S}$  and  $\mathbb{T}$  to be computable, and let  $b: \mathbb{T}_{Ctx} \rightarrow \mathbb{S}_{Ctx}$  be a *partial* function mapping target to source contexts (intuitively, a partial backtranslation). The parameters of the algorithm are a target context  $C$ , a source program  $P$ , and target program  $P$ , and it works as follows. If  $b(C)$  is defined, the result is (SAFE,  $\perp$ ) if the conditions of Definition 5.6 hold. Otherwise, the algorithm returns the pair (MAYBE\_UNSAFE,  $b(C)$ ) meaning that it *may be unsafe* to link the target program  $P$  with  $C$ , being  $b(C)$  a possible source-level attacker of  $P$ . If instead  $b(C)$  is undefined the result is (MAYBE\_UNSAFE,  $\perp$ ), again meaning that it *may be unsafe* to link  $P$  with  $C$ , but no counterexample is exhibited. If Algorithm 1 returns (SAFE,  $\perp$ ), then it is safe to run the program according to the definition of  $aSTV_{TI-RSC}$ :

**THEOREM 5.8** ((SAFE,  $\perp$ )  $\Rightarrow aSTV_{TI-RSC}$  🐦). Let  $P$  be a partial target program,  $C$  be a target context and  $P$  be a source program. If  $\text{SAFETORUN}(C, P, P) = (\text{SAFE}, \perp)$  then  $P \vdash_P^C aSTV_{TI-RSC}$ .

Let  $P$  be a source program and suppose to be about to link its compiled version  $P = \llbracket P \rrbracket_{\mathbb{T}}^T$  with a target context  $C$ . Resorting to program analyses, we make effective the usage of the above security notions and we automatically assess if it is secure to run  $P$  in  $C$ . Indeed, our main result is given by the following corollary: Algorithm 1 can be used to mechanically check whether the current run of the compiler  $\llbracket \cdot \rrbracket_{\mathbb{T}}^T$  on  $P$  robustly preserves all the safety properties.

**COROLLARY 5.9** (ALGORITHM 1 IS CORRECT 🐦). Let  $\langle\langle \cdot \rangle\rangle_{\mathbb{S}}^S$  be a complete, and  $\langle\langle \cdot \rangle\rangle_{\mathbb{T}}^T$  be sound and linear.

If  $\text{SAFETORUN}(C, P, P) = (\text{SAFE}, \perp)$  then  $P \vdash_P^C STV_{RSP}$ .

## 6 STV AT WORK: A CASE STUDY

This section illustrates a use case for our STV framework, inspired by the secure compartmentalization [17]. We consider  $\mu C$ , a source language equipped with a software isolation mechanism: a system is made of a set of mutually distrustful *compartments* that run with minimal privileges and interact only through well-defined

*interfaces*. An interface of a compartment defines a set of public functions that can be invoked by others. Compartments may also have private functions not accessible from outside. Our target is  $\mu A$ , an assembler-like language with modules and functions. Differently from  $\mu C$  compartments, modules provide no isolation and do not distinguish public from private functions, therefore, any module can invoke any functions from others.

We define a compiler that separately compiles  $\mu C$  compartments into  $\mu A$  modules, that are then linked to form a whole program. We want to ensure security of a specific compartment **Main** against the others, namely the context, considered as the attacker – the parts of the operating system not expressed as compartments are considered trusted. We apply our framework to check that a run of the compiler preserves isolation.

## 6.1 Source language and analysis


Assume to have four sets of names: for interfaces ( $i \in \text{IntName}$ ), compartments ( $u \in \text{CompName}$ ) and functions ( $f \in \text{FunName}$ ) and buffers ( $b \in \text{BufferName}$ ). An interface associates interface names with sets of function names. Finally, we have a set of values ( $v \in \text{Val}$ ) restricted for simplicity to integers.

The syntax of our version of  $\mu C$  is:

$$\begin{aligned} \text{Partial}_S \ni P &::= \text{comp Main} : \text{IMain} \{ d^* \} \\ d &::= \text{fun } f \{ e \} \\ e &::= v \mid e_1 \otimes e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid b[e_1] := e_2 \\ &\quad \mid b[e] \mid u.f(e) \mid \text{halt} \\ \text{Ctx}_S \ni C &::= \cdot \mid \text{comp } u : i \{ d^* \} C \end{aligned}$$

where  $\otimes$  represents a generic operator, typically  $\times, +, -, \cdot, \leq$ . A partial program  $P$  consists of the only compartment **Main** with interface **IMain** that always contains the function name **main**. Each compartment  $u$  is defined as a (possibly empty) list of function declarations (possibly mutually recursive), and has a set of private buffers to model its internal state (we assume that each compartment has a unique name). The expressions are as usual except for the invocation of function  $f$  of the compartment  $u$ , and for the **halt** that stops the execution. Our functions have a single parameter, passed through the special buffer **arg**. The contexts are a list of compartments with a hole at the end to be filled with a partial program. Linking a partial program to a context is as expected and results in a whole program  $W$ . Hereafter, we consider only well-typed programs that call only functions of existing compartments.

The observable behavior of a whole program  $W$  is the sequence of function calls and returns performed during the execution. Formally, a trace  $t$  is a sequence of elements from the set  $\Sigma = \text{FunName} \cup \{\text{ret}\}$ . Also, let  $\epsilon$  be empty sequence, that is the neutral element of the sequence concatenation operator  $\cdot$  (i.e.,  $\epsilon \cdot t = t \cdot \epsilon = t$ ).

The semantics of  $\mu C$  is a standard CEK machine [15], quite similar to the one of Juglaret et al. [17]. The semantics of  $\mu C$  is mechanized in the Coq proof assistant . Here, we only describe the transitions between configurations. Given a whole program, we derive a map  $\Delta$  associating each compartment with its functions and with the list of its buffers together with their length. The transitions of  $\mu C$  have the form:  $\Delta \vdash \gamma \xrightarrow{o} \gamma'$ , where  $\gamma = (u, s, \sigma, K, e)$ :  $u$  is the compartment name; the store  $s$ :  $\text{IntName} \rightarrow \text{BufferName} \rightarrow N \rightarrow \text{Val}$  associates a value with each buffer position;  $\sigma$ :  $(\text{CompName} \times \text{Val} \times$

$\text{Cont})^*$  represents the run-time stack that stores the compartment name, the old value of the parameter stored in **arg** (to allow recursion), and the continuation;  $e$  is the expression being executed; similarly for  $\gamma'$ . The label  $o$  in  $\Sigma$  is the observable of the transition: it is an element of  $\text{FunName}$  ( $\text{ret}$  resp.) when  $e$  results in calling (resp. returning from) a function; it is  $\epsilon$  otherwise.

The initial configuration  $\gamma_0(s_{\text{init}})$  is  $(\text{Main}, s_{\text{init}}, \epsilon, \epsilon, e_{\text{main}})$ , where the initial store  $s_{\text{init}}$  initializes buffers, and takes care of the parameter of the **main** function; the stack and the continuation are empty; and the expression  $e_{\text{main}}$  is the body of the **main** function of the **Main** compartment. A configuration gets stuck when accessing either undefined buffers or outside their range. As usual graceful termination happens when the continuation is empty and the current expression is a value, or when reaching a **halt** expression.

Recall from Section 4 that the behavior of a whole program  $W$ , i.e.,  $\text{beh}(W)$ , is the set of the prefixes of its traces. The analysis we consider here is program testing at source level and we assume as given a set of test cases  $\mathcal{T}$ . Consequently, we define the abstract counterpart  $\mu C_k$  of  $\mu C$  as the set of traces resulting from computations shorter than  $k$  steps (we are abusing Definition 4.1). Formally, given  $W = C[P]$ :

$$(\mathcal{W})_{\mu C_k}^{\mu C} = \{ t \mid \forall s_{\text{init}} \in \mathcal{T}. \exists \gamma', n. \Delta \vdash \gamma_0(s_{\text{init}}) \xrightarrow{\mu C}^t \gamma' \wedge n \leq k \}.$$

Since the test cases in the chosen set identify the traces considered by the analysis, their choice is therefore crucial to ensure a good coverage and effectiveness of the analysis. The required completeness of our analysis easily follows:

THEOREM 6.1.  $(\cdot)_{\mu C_k}^{\mu C}$  is a complete analysis.

## 6.2 Target language and analysis

**6.2.1 The concrete target language.** The target language  $\mu A$  shares with  $\mu C$  the sets of names for interfaces, functions, as well as the set of values and operators. Also assume a finite set of registers **Reg** ranged over by  $r$  possibly indexed; a finite set of basic block labels **BBLLabel** ranged over by  $\ell$ ; and let **Label** be **BBLLabel**  $\cup$   $\text{FunName}$ . The syntax of  $\mu A$  is:

$$\begin{aligned} P &::= \text{Main} : \text{IMain} \{ d^* \} \\ d &::= \text{define } f \{ b^* \} \quad b ::= \ell : c^* j \\ c &::= \text{nop} \mid \text{const } v \mid \text{mov } r_s r_d \mid \text{ld } r_s r_d \mid \text{st } r_s r_d \mid \text{op } r_1 r_2 \\ j &::= \text{halt} \mid \text{bnz } r \ell_1 \ell_2 \mid \text{jmp } \ell \mid \text{call } u.f \ell_r \mid \text{ret} \\ C &::= \cdot \mid u : i \{ d^* \} C \end{aligned}$$

A partial program  $P$  only consists of the module **Main** with interface **IMain** containing the function **main** only. A function  $f$  consists of a list of basic blocks  $b$  that in turn are made by a label  $\ell$ , a sequence of instructions  $c$  ending with a jump instruction  $j$  leaving the block. The instructions include the standard ones for moving, storing and loading constants and memory content into/from registers, and the standard arithmetic/logic operations. Besides halting and function returns, the jump instructions include conditional and unconditional branches and function calls. Note that the targets of the jumps are labels, in particular,  $\ell_r$  represents the label of the basic block to run upon a return from  $u.f$ . As usual, parameters are passed back and forth through a special register  $r_{\text{com}}$ . A context  $C$  is a list of modules with a hole at the end to be filled with a partial program  $P$  to give raise to a whole program  $W = C[P]$ .

We take the same observables of  $\mu\mathbf{C}$ , thus, as before the behavior of a whole program is the sequence of functions calls and returns. Hereafter, we consider only well-formed programs, i.e., they use only existing labels and registers.

To define the semantics it is convenient to assume a memory layout  $\mathcal{L}$  that specifies the portion of memory where the text of the program is stored, and that is read-only as usual. We use natural numbers to model memory locations (when needed we denote with the label of basic block its location in memory), so a memory **mem** is a mapping from naturals to values. A register file **reg** is a mapping from registers to values.

The semantics of the language  $\mu\mathbf{C}$  is given in terms of a labeled transition system with transitions of the form:

$$(\mathbf{u}, \sigma, \mathbf{mem}, \mathbf{reg}, \mathbf{pc}) \xrightarrow{o} (\mathbf{u}', \sigma', \mathbf{mem}', \mathbf{reg}', \mathbf{pc}')$$

where  $\mathbf{u}$  is the current module;  $\sigma \in (\text{IntName}, \text{Label})^*$  represents the run-time stack that stores the name of the calling module and the return address; **mem** and **reg** are the current memory and register file, respectively; **pc** is a location in  $\mathcal{L}$  representing the program counter (the semantics ensures this invariant during execution).

The initial configuration is  $(\text{Main}, \epsilon, \mathbf{mem}, \mathbf{reg}_i, \ell_e)$  where **mem** is the initial memory that contains the program code in  $\mathcal{L}$ ;  $\mathbf{reg}_i$  is the initial register file including the register **rcom** carrying the parameter of the function **Main.main**;  $\ell_e$  represents the entry basic block of the function **Main.main**.

A computation gracefully terminates when it reaches the **halt** instruction; whereas it gets stuck when it tries to write a memory location in  $\mathcal{L}$ . Also here the behavior of a whole program **W**, i.e.,  $\text{beh}(\mathbf{W})$ , is the set of the prefixes of its traces.

**6.2.2 The abstract target language.** Our abstract language is a sort of basic process algebra, called *history expressions*  $\mathbb{H}$ . They are extracted by statically analyzing a program through a *type and effect system* and are a sound over-approximation of a program behavior, i.e., function calls and returns in our case. We first describe their syntax and their semantics, and then we present how they are extracted from a  $\mu\mathbf{A}$  program.

The syntax of our history expression follows:

$$\begin{aligned} \Pi &::= \text{Main} : \text{IMain} \{ \delta^* \} \quad \delta ::= (\mathbf{f}, (\ell : \beta)^*) \\ \beta &::= \epsilon \mid o \mid \beta_1; \beta_2 \mid \beta_1 + \beta_2 \mid \mathbf{h} \\ \Gamma &::= \cdot \mid \mathbf{u} : \mathbf{i} \{ \delta^* \} \Gamma \end{aligned}$$

A partial history expression  $\Pi$  abstracts a  $\mu\mathbf{A}$  partial program **P** where the abstract function definition  $\delta$  associates a function name with the abstraction of its basic blocks. The history expression  $\beta \in \mathcal{B}$  abstracts a sequence of assembly instructions with  $\epsilon$  as the empty sequence;  $o$  records an observable event from  $\Sigma$ ;  $\beta_1; \beta_2$  is for sequencing; the non-deterministic choice  $\beta_1 + \beta_2$  abstracts conditionals; finally,  $\mathbf{h}$  are *history variables* associated with a bijection  $\Lambda$  to functions and labels of basic blocks, which represent their abstract behavior (*latent effect*). Abstract contexts  $\Gamma$  are a list of abstract modules ending with a hole to be filled with a partial history expression  $\Pi$  to get the whole history expression  $\mathcal{W} = \Gamma[\Pi]$ .

The semantics of a whole history expression is a set of (possibly infinite) histories, i.e., a trace of labels, obtained by a layered transaction system describing abstract executions. Actually, labels are distinct from history expressions, but we overload them for

$$\begin{array}{c} \frac{o \in \Sigma}{\rho \vdash o \xrightarrow{o} \epsilon} \quad \frac{o \notin \Sigma}{\rho \vdash o \xrightarrow{\epsilon} \epsilon} \quad \rho \vdash \epsilon; \beta \xrightarrow{\epsilon} \beta \\[10pt] \frac{\rho \vdash \beta_1 \xrightarrow{\lambda} \beta'_1}{\rho \vdash \beta_1; \beta_2 \xrightarrow{\lambda} \beta'_1; \beta_2} \quad \frac{}{\rho \vdash \mathbf{h} \xrightarrow{\epsilon} \rho(\mathbf{h})} \\[10pt] \rho \vdash \beta_1 + \beta_2 \xrightarrow{\epsilon} \beta_1 \end{array}$$

Figure 1: Semantics for abstract assembly instructions

$$\begin{array}{c} \frac{\Lambda(\mathbf{u}'.\mathbf{f}') = \mathbf{h}' \quad \Lambda(\mathbf{u}.\mathbf{f}.\ell_r) = \mathbf{h}}{\mathbf{P}, \mathbf{u}, \mathbf{f} \vdash \text{call } \mathbf{u}' \mathbf{f}' \ell_r \triangleright \mathbf{u}'.\mathbf{f}'; \mathbf{h}'; \mathbf{h}} \\[10pt] \frac{\Lambda(\mathbf{u}.\mathbf{f}.\ell_1) = \mathbf{h}_1 \quad \Lambda(\mathbf{u}.\mathbf{f}.\ell_2) = \mathbf{h}_2}{\mathbf{P}, \mathbf{u}, \mathbf{f} \vdash \text{bnz } r \ell_1 \ell_2 \triangleright \mathbf{h}_1 + \mathbf{h}_2} \end{array}$$

Figure 2: The history expression for calls and branching

simplicity: a label  $\lambda$  is the empty history  $\epsilon$ , the abstract observable  $o$ . The full definition of the layered transition systems is available as additional material (📄), here we consider only the layer for abstract instructions  $\beta$ . The rules in Figure 1 describe the abstract execution steps for function calls and returns, sequencing, jumps and conditionals. The environment  $\rho$  maps the history variable  $h$  to the expression representing the behavior of the function and basic block associated with  $\mathbf{h}$ . (The rule for variables implements the copy rule.) As in [10], in the rules we assume  $(\mathcal{B}, ;)$  to be a monoid,  $(\mathcal{B}, +)$  to be a commutative monoid and the following axiom:  $o; (\beta_1 + \beta_2) = o; \beta_1 + o; \beta_2$ . The above semantics induces  $\text{beh}(\mathcal{W})$  as the set of prefixes the history generates.

We now describe how to compute the history expressions from a  $\mu\mathbf{A}$  program, and we neglect its types since of no interest here. We only consider the analysis of partial programs. In particular, the cases for function calls and conditional branches are in Figure 2, where **P**, **u**, **f** are the partial program, the module and the function under consideration, respectively. Let  $\Lambda$  be the bijection mentioned above from qualified function names and block labels to history variables. The effect of a function call consists of the sequence of the observable  $\mathbf{u}'.\mathbf{f}'$  representing the call, its latent effect  $\mathbf{h}'$  and the latent effect  $\mathbf{h}$  associated with the return address  $\ell_r$ . The effect of a conditional jump is the non-deterministic choice made of the latent effects of the branches. Now, following Section 4 the target analysis is  $\llbracket \mathbf{W} \rrbracket_{\mathcal{H}}^{\mu\mathbf{A}} = \mathcal{W}$ , where  $\mathcal{W}$  is extracted from the whole program **W**; similarly for partial programs **P** and context  $\Gamma$ .

The required linearity and soundness of the analysis follow:

**THEOREM 6.2 (LINEARITY).** *Let  $\mathbf{C}$  be a context and **P** be a partial program. If  $\llbracket \mathbf{C} \rrbracket_{\mathcal{H}}^{\mu\mathbf{A}} = \Gamma$ ,  $\llbracket \mathbf{P} \rrbracket_{\mathcal{H}}^{\mu\mathbf{A}} = \Pi$ , and  $\llbracket \mathbf{C}[\mathbf{P}] \rrbracket_{\mathcal{H}}^{\mu\mathbf{A}} = \mathcal{W}$  then  $\text{beh}(\mathcal{W}) = \text{beh}(\Gamma[\Pi])$ .*

**THEOREM 6.3 (SOUNDNESS).** *Let **W** be a whole program, if  $\llbracket \mathbf{W} \rrbracket_{\mathcal{H}}^{\mu\mathbf{A}} = \mathcal{W}$  then  $\text{beh}(\mathbf{W}) \subseteq \text{beh}(\mathcal{W})$ .*

### 6.3 From $\mu C$ to $\mu A$ and back

**6.3.1 The compiler.** The compiler from  $\mu C$  to  $\mu A$  is essentially the one of [17]: it translates a compartment into a module with the same name, interface and functions. The full definition of the compiler is in the additional material. A  $\mu C$  function `fun f { e }` generates the  $\mu A$  function `f`, where `e` is compiled to a set of basic blocks. Expressions are translated straightforwardly, except for calls and returns, also because we do not distinguish between inter- and intra-module calls, unlike [17]. Roughly, a call `u.f(e)` becomes a push of its actual argument (and some additional information) on the stack, followed by a cleanup of registers and then by `call u.f  $\ell_r$` . As expected, upon return the environment of the caller is restored and  $\ell_r$  is used as return address.

**6.3.2 Backtranslation.** Our backtranslation maps contexts of  $\mu A$  to those of  $\mu C$ . Differently from classical ones, ours does not work as a proof technique only, but it is an *algorithmic* tool to assess the security of the compiled program and context w.r.t. the source.

Our backtranslation *bt* is inductively defined on the syntax of target contexts (its full definition is in the additional material). The cases for contexts are simple inductive steps rendering the syntax of target modules in that of source components. The cases for function definitions behave similarly, but special code for copying actual arguments from `arg[0]` to `rcom` (and vice versa) is added at the beginning and at the end of functions body, respectively. Intuitively, for basic blocks we need the following auxiliary structures at the source: (i) the buffer `regu` with the register contents; (ii) the global buffer `mem` storing the target memory, initialized with the initial target memory `memi`<sup>1</sup> and (iii) the function  `$\ell 2b$`  mapping function names and labels to their corresponding code.

Roughly,  $\ell : c^*j$  is backtranslated to a sequence of source expressions simulating the behavior of the target code. We just illustrate the cases for the `mov` instruction, the conditional jump and the call (recall that `rcom` is for parameter passing, and note that `bnz` is backtranslated as an *if-then-else* whose branches are the backtranslation of the basic blocks identified by the labels).

```
bt(mov rs rd) = regu[rd] := regu[rs]
bt(bnz r  $\ell_1$   $\ell_2$ ) = if reg[r] then bt( $\ell_1 : \ell 2b(\ell_1)$ ) else bt( $\ell_2 : \ell 2b(\ell_2)$ )
bt(call u.f  $\ell_r$ ) = u.f(reg[com])
```

**6.3.3 Completing the example.** We now show how all the ingredients of our STV fit together by detailing the example of Section 3. As expected, our observables are function calls and returns, while we assume the context to only contain the malicious logger.

Following the intuition of Section 6.2, we first extract the history expressions for the partial target program **P** and for the target context **C** containing the malicious logger. For the partial program **P** we obtain the history expression  **$\Pi$**  (we omit the parts for `Main.ok` and `Main.store`):

```
1 | Main : IMain {
2 |   (main,
3 |      $\ell_{entry} : \text{Main.ok}; h_{\text{Main.ok}}; h_{\text{Main.main.}\ell_{okret}}$ ,
4 |      $\ell_{okret} : h_{\text{Main.main.}\ell_1} + h_{\text{Main.main.}\ell_2}$ ,
5 |      $\ell_1 : \text{Main.store}; h_{\text{Main.store}}; h_{\text{Main.main.}\ell_{end}}$ ,
```

<sup>1</sup>Note that  $\mu C$  has no explicit global buffers, but one can implement them using a dedicated compartment having a local memory buffer accessible to others through read and write functions.

```
6 |      $\ell_2 : \text{Logger.err}; h_{\text{Logger.err}}; h_{\text{Main.main.}\ell_{end}}$ ,
7 |      $\ell_{end} : h_{\text{Main.main.}\ell_{trailer}}$ ,
8 |      $\ell_{trailer} : \text{ret}$ )
9 | }
```

Similarly, for the malicious context  $\Gamma$  we have:

```
1 | Logger : ILogger {
2 |   (err,
3 |      $\ell_{entry} : \text{Main.store}; h_{\text{Main.store}}; h_{\text{Main.main.}\ell_{trailer}}$ ,
4 |      $\ell_{trailer} : \text{ret}$ )
5 | }
```

We apply backtranslation to the code of the malicious logger of Section 3, and obtain the following source context:

```
1 | comp Logger : ILogger {
2 |   fun err {
3 |     reg[com] := arg[0]; // initialize registers
4 |     Main.store (reg[com]);
5 |     arg[0] := reg[com];
6 |   }
7 | } [-]
```

Next we link the source program **P** with the source context above obtaining the whole program **W**, and test it, i.e., we apply our source analysis. To this aim, assume that maximal execution length is  $k = 5$ ; the test cases in  **$\mathcal{T}$**  assign 2 and 42 as parameter of `Main.main`; the function `Main.ok` returns 0 whenever its parameter is equal to 42. Thus, the result of the analysis is (up to  $\epsilon$  moves):

$$(\mathbb{W})_{\mu C_k}^{\mu C} = \{t \mid t \text{ prefix of } \text{Main.ok}; \text{ret}; \text{Main.store}; \text{ret} \vee \\ t \text{ prefix of } \text{Main.ok}; \text{ret}; \text{Logger.err}\}$$

Note that the second maximal trace terminates abruptly because the call to `Main.store` performed by `Logger.err` is prevented by the semantics and thus leads to a stuck configuration.

We then compare the above set of traces with those generated by  $\Gamma[\Pi]$ . Since the set  $(\mathbb{W})_{\mu C_k}^{\mu C}$  is finite (as always is when the analysis is based on program testing) it is easy to see that it does not include  $\text{beh}(\Gamma[\Pi])$  (also finite in this example) because the trace `Main.ok; ret; Logger.err; Main.store; ret; ret` belongs to  $\text{beh}(\Gamma[\Pi])$ , but not to  $(\mathbb{W})_{\mu C_k}^{\mu C}$ . So, Algorithm 1 answers that it *may not be secure* to execute our compiled program in the given target context, as expected.

Note that when  $\text{beh}(\Gamma[\Pi])$  is not finite, we can resort to standard results from automata theory to check inclusion of behavior. Indeed, since  $(\mathbb{W})_{\mu C_k}^{\mu C}$  is finite, it can trivially be represented as a regular language, whereas  $\text{beh}(\Gamma[\Pi])$  as a context-free language [10]. Thus, checking the required inclusion is decidable through well known algorithms, e.g., using automata-based model checking [34].

Take instead a *non-malicious* implementation of `Logger.err`, e.g., one that calls no function. The traces of the whole program obtained by linking **P** and the backtranslated source context include now those of the target: Algorithm 1 establishes the preservation of the security properties of interest.

## 7 DISCUSSION

Here we detail some assumptions our STV relies on and discuss the trade-offs to be considered when we want to use it in practice.



A first assumption is that the partial program  $P$  to be compiled is trusted, and that the context in which we plug it consists of the modules that  $P$  *explicitly* refers to. Also we require the compiled code of these modules to be available at link time just before  $P$  is about to run, so as to extract the abstraction of their behavior. Since these modules can in turn depend on other modules (even system code like system calls), the transitive resolution of dependency may require the analysis of many modules. From a theoretical perspective, it is enough to have the code of all the modules our program  $P$  depends on to perform *STV*, but this could be infeasible in practice, e.g., if the time spent in the analysis is too high. The scenario we just sketched is quite general because no assumptions on security policies we want to preserve are made (thus on the kind of observable behavior we want to approximate). However, in a setting where the code of the modules does not change over time and where the class of security policies to be preserved is fixed, we could extract the abstraction of the behavior from the module code just once, and reuse it on need. Another possibility could consist of reducing the set of modules to check by certifying them through other mechanisms, e.g., remote attestation [13], so including them into the trusted computing base, e.g., this could be done for the software of the operating system. Alternatively, one may limit the extent of *STV* only to the modules which  $P$  directly depends on, and may resort to other dynamic defense mechanisms to complete the protection of the program. For example, a possible implementation could couple *STV* together with enclaved execution [11, 20, 25]. More precisely, *STV* may check that a partial program  $P$  is not linked and loaded in the same enclave with possibly malicious code, leaving to the enclave mechanism the protection from other malicious (system) software. Studying a possible integration between *STV* and enclaves is future work. In conclusion, an actual implementation of our framework should trade security guarantees off for performance, and should also cooperate with existing countermeasures.

A second assumption is that the threat model we consider is *static*: we can beforehand classify a context as malicious or not by inspecting its code. Moreover, we do not consider the case in which a harmless context becomes malicious at run-time because compromised, e.g., due to undefined behavior. Again, a more general threat model could be faced by integrating *STV* with other defense mechanisms, e.g., software sandboxing [30], software fault isolation [35], robustly safe compartmentalizing compilation [5]. An interesting future work may consist of relaxing the assumption on the separation between the trusted partial program and the (malicious) context. We may consider a set of mutually distrustful modules that can be compromised at run-time and become attack vectors [5].

Finally, the precision and efficacy of our proposal heavily depend on those of the analyses used to approximate program behavior. Indeed, inaccurate analyses may lead to a high number of false positives, i.e., safe programs that are signed as insecure. Accurate analyses better approximate the behavior of programs and reduce the number of false positives, but they can be computationally expensive. This is the price we have to pay if we want a *sound* push-button verification procedure, and constitutes the major challenge to scale to more complex, real-world languages. However, we

believe that this issue can be mitigated by leveraging the results from the (very vast) literature on program analysis and verification. In the future, we plan to investigate which classes of static analyses may be of interest for *STV*.

## 8 RELATED WORK

In the last two decades, secure compilation has been mainly defined in terms of a full abstraction property of compilers [1]. Patrignani and Garg [27] highlighted some potential shortcomings of this approach. Patrignani and Garg [28] and Abate et al. [2, 3] then proposed new notions of secure compilation, just preserving specific families of (hyper)properties. Abate et al. [4] compared the two approaches and gave a unifying criterion in a categorical framework. Here, we considered *robust safety preservation* [3] and proposed a way to automatically check it.

Since the first work by Pnueli et al. [29] translation validation has found a vast number of applications for checking the *correctness* of compilation runs. Necula [23] applied it to the GCC compiler, and Sewell et al. [31] used a similar approach for the verification of the seL4 micro-kernel. Variants of this technique were applied to prove the correctness of compiler optimizations by Namjoshi and Zuck [22]. Their approach is based on a *stuttering simulation* ensuring after each step of optimization that the resulting program is a refinement of the input of the transformation, if a simulation exists. Differently from these approaches, our proposal does not target correctness but the preservation of robust safety properties.

To the best of our knowledge, the only paper about a fully automatic approach to security preservation is by Namjoshi and Tabajara [21], who proposed a translation validation schema based on refinements. More precisely, they develop two automaton-based refinement schemata to handle hyperproperty preservation in the case of passive attackers. Roughly, the idea of both schemata is the following: if a *witness* refinement relation exists between the (suitably encoded) source and target systems, then the hyperproperty of choice has been preserved. Their work differs from ours in at least two aspects. The first concerns the underlying threat model: we focus on partial programs which are plugged in possibly malicious contexts, i.e., attackers that actively interact with the victim. The second difference is that our proposal is *not* tied to a specific analysis or verification technique.

Our use case of Section 6 is inspired by the work of Juglaret et al. [17]. Our source language is essentially theirs, but the target one slightly extends their with the notion of module. The main differences with their work are that (i) we consider the compiler as a black box; and (ii) we establish that the compilation of a single program preserves security by running an algorithm based on translation validation and static analysis at link time. On the contrary, they proved the compiler secure by establishing a generalization of full abstraction using the Coq proof assistant.

Our idea of abstracting through a type and effect system the behavior of programs using history expressions to prove their properties dates back to Bartoletti et al. [9, 10], who used them to statically prove access control on a functional language. Also Skalka et al. [33] use them under the name of trace effects to enforce access control in a functional language. Differently from that strain of work, here we extract history expressions from an assembly-like language.

## 9 CONCLUSION

In this paper we performed a first step towards the problem of automatically checking security preservation under compilation. We proposed *STV* as the basis for an automatic tool to check the preservation of robust safety properties, a recently introduced class of security policies. The distinguishing feature of our approach is that it leverages program analysis to devise a mechanical (and approximated) procedure, and detects if all the safety properties are preserved under a given target context at linking time.

We introduced a general theory for *STV*, mechanized it in the Coq proof assistant, and provided sufficient conditions for its correctness. This ensures that no insecure code is ever executed, even if it can possibly signal as insecure harmless programs, because of the approximation introduced by program analysis techniques. This is the trade-off to pay for a fully automatic verification procedure.

Finally, we illustrated our idea on a use case that considers the preservation of security properties on compartmentalized languages. More precisely, we considered a source language equipped with *compartments* to ensure isolation, and we compiled it to an assembler-like language that provides no isolation. We exploited our *STV* to check that the resulting target program preserves isolation when plugged into an adversarial context. To do that, we just compared an over-approximation of the behavior of the compiled program, given in terms of history expressions, in a given environment with an under-approximation of the corresponding source program, given in terms finite execution traces (program testing at the source level). An aside result of our use case is that, to the best of our knowledge, we provided the first procedure to extract history expressions from an assembler-like language.

As a future work, besides what already discussed in Section 7, we plan to extend our framework to deal with a larger class of security policies, as subset-closed hypersafety and to allow different observables in source and target programs following the line of Abate et al. [2]. Finally, we plan to assess the practical effectiveness of our framework through a proof-of-concept implementation.

## ACKNOWLEDGMENTS

This work was partially supported by the MIUR project PRIN 2017FTXR7S *IT MATTERS*; Matteo Busi was also partially supported by the research grant on Formal Methods and Techniques for Secure Compilation of the Computer Science Dept., University of Pisa.

## REFERENCES

- [1] Martin Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects (LNCS)*, J. Vitek and C.D. Jensen (Eds.), Vol. 1603. Springer, 19–34.
- [2] Carmine Abate, Roberto Blanco, Ștefan Ciobăcă, et al. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *29th European Symposium on Programming*, 1–28.
- [3] Carmine Abate, Roberto Blanco, Deepak Garg, et al. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium*, 2019, 256–271.
- [4] Carmine Abate, Matteo Busi, and Stelios Tsampas. 2021. Fully Abstract and Robust Compilation. In *Asian Symposium on Programming Languages and Systems*. Springer, 83–101.
- [5] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, et al. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proc 2018 ACM SIGSAC Conference on Computer and Communications Security*, 1351–1368.
- [6] Bowen Alpern and Fred B. Schneider. 1985. Defining Liveness. *Inf. Process. Lett.* 21, 4 (1985), 181–185.
- [7] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, et al. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. <https://doi.org/10.1145/3371075>
- [8] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic “Constant-Time”. In *31st IEEE Computer Security Foundations Symposium*, 2018, 328–343.
- [9] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. 2005. Enforcing Secure Service Composition. In *18th IEEE Computer Security Foundations W/S*, 211–223.
- [10] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, et al. 2009. Local policies for resource usage analysis. *ACM Trans. Program. Lang. Syst.* 31, 6 (2009), 23:1–23:43. <https://doi.org/10.1145/1552309.1552313>
- [11] Matteo Busi, Job Noorman, Jo Van Bulck, et al. 2021. Securing Interruptible Enclaved Execution on Small Microprocessors. *ACM Trans. Program. Lang. Syst.* 43, 3 (2021).
- [12] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.
- [13] George Coker, Joshua D. Guttman, Peter Loscocco, et al. 2011. Principles of remote attestation. *Int. J. Inf. Sec.* 10, 2 (2011), 63–81.
- [14] Chaoyang Deng and Kedar S. Namjoshi. 2016. Securing a Compiler Transformation. In *Proc Static Analysis - 23rd International Symposium, 2016 (LNCS)*, X. Rival (Ed.), Vol. 9837. Springer, 170–188.
- [15] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.
- [16] Catalin Hritcu, David Chisnall, Deepak Garg, et al. 2019. Secure Compilation. <https://blog.sigplan.org/2019/07/01/secure-compilation/>. Last access Feb 2021.
- [17] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, et al. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *IEEE 29th Computer Security Foundations Symposium*, 45–60.
- [18] Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.* 3, 2 (1977), 125–143.
- [19] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, 42–54.
- [20] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, et al. 2013. Innovative instructions and software model for isolated execution. In *2nd W/S on H/W and Architectural Support for Security and Privacy*, R.B. Lee and W. Shi (Eds.). ACM.
- [21] Kedar S. Namjoshi and Lucas M. Tabajara. 2020. Witnessing Secure Compilation. In *Proc Verification, Model Checking, and Abstract Interpretation - 21st International Conference, 2020*, 1–22.
- [22] Kedar S. Namjoshi and Lenore D. Zuck. 2013. Witnessing Program Transformations. In *Proc Static Analysis - 20th International Symposium*, 2013, 304–323.
- [23] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proc 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, M.S. Lam (Ed.), ACM, 83–94.
- [24] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer. <https://doi.org/10.1007/978-3-662-03811-6>
- [25] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, et al. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.* 20, 3, Article 7 (July 2017), 33 pages. <https://doi.org/10.1145/3079763>
- [26] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A survey of fully abstract compilation and related work. *Comput. Surveys* (2019).
- [27] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperty Preservation. In *30th IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 392–404.
- [28] Marco Patrignani and Deepak Garg. 2019. Robustly Safe Compilation. In *Proc 28th European Symposium on Programming*, 469–498.
- [29] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proc 4th Tools and Algorithms for Construction and Analysis of Systems, International Conference (LNCS)*, B. Steffen (Ed.), Vol. 1384. Springer, 151–166.
- [30] Charles Reis and Steven D. Gribble. 2009. Isolating Web Programs in Modern Browser Architectures. In *Proc 4th ACM European Conference on Computer Systems*. ACM, 219–232. <https://doi.org/10.1145/1519065.1519090>
- [31] Thomas A.L. Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 471–482.
- [32] Robert Sison and Toby Murray. 2019. Verifying That a Compiler Preserves Concurrent Value-Dependent Information-Flow Security. In *10th International Conference on Interactive Theorem Proving*, J. Harrison, J. O’Leary, and A. Tolmach (Eds.), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:19.
- [33] Christian Skalka, David Darais, Trent Jaeger, et al. 2020. Types and Abstract Interpretation for Authorization Hook Advice. In *CSF. IEEE*, 139–152.
- [34] Moshe Y. Vardi. 2007. Automata-Theoretic Model Checking Revisited. In *VMCAI (LNCS)*, Vol. 4349. Springer, 137–150.
- [35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, et al. 1993. Efficient Software-Based Fault Isolation. *SIGOPS Oper. Syst. Rev.* 27, 5 (Dec. 1993), 203–216.