# Witnessing Secure Compilation

Kedar S. Namjoshi<sup>1</sup> and Lucas M. Tabajara<sup>2</sup>

Bell Labs, Nokia, Murray Hill, NJ, USA kedar.namjoshi@nokia-bell-labs.com
Rice University, Houston, TX, USA lucasmt@rice.edu

Abstract. Compiler optimizations are designed to improve run-time performance while preserving input-output behavior. Correctness in this sense does not necessarily preserve security: it is known that standard optimizations may break or weaken security properties that hold of the source program. This work develops a translation validation method for secure compilation. Security (hyper-)properties are expressed using automata operating over a bundle of program traces. A flexible, automaton-based refinement scheme, generalizing existing refinement methods, guarantees that the associated security property is preserved by a program transformation. In practice, the refinement relations ("security witnesses") can be generated during compilation and validated independently with a refinement checker. This process is illustrated for common optimizations. Crucially, it is not necessary to verify the compiler implementation itself, which is infeasible in practice for production compilers.

### 1 Introduction

Optimizing compilers such as GCC and LLVM are used to improve run-time performance of software. Programmers expect the optimizing transformations to preserve program behavior. A number of approaches, ranging from automated testing (cf. [27,13]) to translation validation (cf. [24,21,30]) to full mathematical proof (cf. [14]) have been utilized to check this property.

Programmers also implicitly expect that optimizations do not alter the security properties of the source code (except, possibly, to strengthen them). It is surprising, then, to realize that even correctly implemented optimizations, such as the one illustrated in Figure 1, may weaken security guarantees (cf. [12,10]).

Fig. 1. Dead Store Elimination: Introducing Information Leaks

This common optimization removes stores that have no effect on the program's input-output behavior. Assuming that x is not referenced in rest\_of\_program,

the transformation correctly replaces  $\mathbf{x}:=0$  in the source program on the left with  $\mathtt{skip}$ . The original assignment, however, was carefully placed to clear the memory holding the secret key. By removing it, the secret key becomes accessible in the rest of the program and vulnerable to attack. While this leak can be blocked using compiler directives [28], such fixes are unsatisfactory, as they are not portable and might not block other vulnerabilities.

The ideal is a mathematical guarantee that security is preserved across compiler optimizations. A full mathematical proof establishing this property over all source programs is technically challenging but feasible for a compact, formally developed compiler such as CompCert [14,3]. It is, however, infeasible for compilers such as GCC or LLVM with millions of lines of code written in hard-to-formalize languages such as C and C++.

The alternative method of *Translation Validation* [24] settles for the less ambitious goal of formally checking the result of every run of a compiler. In the form considered here, a compiler is *designed* to generate additional information (called a "certificate" or a "witness") to simplify this check [25,20]. To establish correctness preservation, a refinement (i.e., simulation) relation is the natural choice for a witness. Moreover, such a relation is validated by checking inductiveness over a pair of single-step transitions (in the source and target programs), a condition that is easily encoded as an SMT query.

For many optimizations, the associated refinement relations are logically simple<sup>3</sup>, being formed of equalities between variables at corresponding source and target program points [25,16,20]. Crucially, neither the compiler nor the witness generation machinery have to be proved correct: an invalid witness points to either a flaw in compilation, or a misunderstanding by the compiler writer of the correctness argument; both outcomes are worth further investigation.

In this work, we investigate translation validation for preservation of *security* properties. Two key questions are: What is a useful witness format for security properties? and How easy is it to generate and check witnesses for validity? Refinement proof systems are known only for two important security properties, non-interference [7,8,17] and constant-time execution [3].

In this context, a major contribution of this work is the development of a uniform, automaton-based refinement scheme for a large class of security properties. We use a logic akin to HyperLTL [5] to describe hyperproperties [26,6] (which are sets of sets of sequences). A security property  $\varphi$  is represented by a formula  $Q_1\pi_1,\ldots,Q_k\pi_k:\kappa(\pi_1,\ldots,\pi_k)$ , where the  $\pi_i$ 's represent traces over an observation alphabet, the  $Q_i$ 's stand for either existential or universal quantifiers, and  $\kappa$  is a set of bundles of k program traces, represented by a non-deterministic Büchi automaton, A, whose language is the  $complement^4$  of the set  $\kappa$ .

<sup>&</sup>lt;sup>3</sup> Examples include dead-store elimination, constant propagation, loop unrolling and peeling, loop-invariant code motion, static single assignment (SSA) conversion.

<sup>&</sup>lt;sup>4</sup> In this, we follow the standard practice in model checking of using automata for the negation of the desired property, rather than for the property itself.

A transformation from program S to program T preserves a security property  $\varphi$  if every violation of  $\varphi$  by T has a matching violation of  $\varphi$  by S. Intuitively, matching violations have the same inputs and the same cause.

The first refinement scheme that we propose applies to purely universal properties, i.e., those of the form  $\forall \pi_1 \dots \forall \pi_k : \kappa(\pi_1, \dots, \pi_k)$ . The witness is a relation R between the product transition systems  $A \times T^k$  and  $A \times S^k$  that meets certain inductiveness conditions. If R can be defined, the transformation preserves  $\varphi$ .

The second refinement scheme applies to properties with arbitrary quantification (the  $\forall \exists$  alternation is needed to express limits on an attacker's knowledge). Here, the witness is a pair of relations: one being a refinement relation (as before) between the product transition systems  $A \times T^k$  and  $A \times S^k$ ; the other component is an input-preserving bisimulation relation between T and S. We show that if such a pair of relations can be defined, the transformation from S to T preserves  $\varphi$  in the sense above.

We give examples of program transformations to illustrate the definition of security witnesses. As is the case for correctness, these witness relations also have a simple logical form, which can be analyzed by SMT solvers. The information needed to define these relations is present during compilation, so the relations may be easily generated by a compiler.

Finally, we show that the refinement proof rules derived by this scheme from the automata-theoretic formulations of non-interference and constant-time are closely related to the known proof rules for these properties. Proofs carried out with the known notions are also valid for the newly derived ones.

The key contribution of this work is in the flexible refinement scheme, which makes it possible to construct refinement proof rules for a wide range of security properties, including subtle variations on standard properties such as non-interference. The synthesized refinement rules may also be used for deductive proofs of security preservation. The primary inspiration for the automaton-based refinement notion comes from a beautiful paper by Manna and Pnueli [15] demonstrating how to synthesize a deductive verification system for a temporal property from its associated automaton.

# 2 Example

To illustrate the approach, consider the following source program, S.

```
L1: int x := read_secret_input();
L2: int y := 42;
L3: int z := y - 41;
L4: x := x * (z - 1);
```

In this program, x stores the value of a secret input. As will be described in Section 3.1, this program can be modeled as a transition system. The states of the system can be considered to be pairs  $(\alpha, \ell)$ . The first component  $\alpha : \mathcal{V} \to \text{Int}$  is a partial assignment mapping variables in  $\mathcal{V} = \{x, y, z\}$  to values in Int, the set of values that a variable of type int can contain. The second component

 $\ell \in \text{Loc} = \{\text{L1}, \text{L2}, \text{L3}, \text{L4}\}$  is a location in the program, indicating the next instruction to be executed. In the initial state,  $\alpha$  is empty and  $\ell$  points to location L1. Transitions of the system update  $\alpha$  according to the variable assignment instructions, and  $\ell$  according to the control flow of the program.

To specify a notion of security for this program, two elements are necessary: an attack model describing what an attacker is assumed to be capable of observing (Section 3.2) and a security property over a set of program executions (Section 4). Suppose that an attacker can see the state of the memory at the end of the program, represented by the final value of  $\alpha$ , and the security property expresses that for every two possible executions of the program, the final state of the memory must be the same, regardless of the secret input, thus guaranteeing that the secret does not leak. Unlike correctness properties, this is a two-trace property, which can be written as a formula of the shape  $\forall \pi_1, \pi_2 : \kappa(\pi_1, \pi_2)$ , where  $\kappa(\pi_1, \pi_2)$  expresses that the memory at the end of the program is the same for traces  $\pi_1$  and  $\pi_2$  (cf. Section 4). The negation of  $\kappa$  can then be translated into an automaton A that detects violations of this property.

It is not hard to see that the program satisfies the security property, since y and z have constant values and at the end of the program x is 0. However, it is important to make sure that this property is preserved after the compiler performs optimizations that modify the source code. This can be done if the compiler can provide a witness in the form of a refinement relation (Section 5). Consider, for example, a compiler which performs constant folding, which simplifies expressions that can be inferred to be constant at compile time. The optimized program T would be:

```
L1: int x := read_secret_input();
L2: int y := 42;
L3: int z := 1;
L4: x := 0;
```

By taking the product of the automaton A with two copies of S or T (one for each trace  $\pi_i$  considered by  $\kappa$ ), we obtain automata  $A \times S^2$  and  $A \times T^2$  whose language is the set of pairs of traces in each program that violates the property. Since this set is empty for S, it should be empty for T as well, a fact which can be certified by providing a refinement relation R between the state spaces of  $A \times T^2$  and  $A \times S^2$ .

As the transformation considered here is very simple, the refinement relation is simple as well: it relates configurations  $(q,t_0,t_1)$  and  $(p,s_0,s_1)$  of the two spaces if the automaton states p,q are identical, corresponding program states  $t_0,s_0$  and  $t_1,s_1$  are also identical (including program location), and the variables in  $s_0$  and  $s_1$  have the constant values derived at their location (see Section 6 for details). The inductiveness of this relation over transitions of  $A \times T^2$  and  $A \times S^2$  can be easily checked by an SMT solver by representing the states symbolically.

# 3 Background

We propose an abstract program and attack model defined in terms of labeled transition systems. We also define Büchi automata over bundles of program traces, which will be used in the encoding of security properties, and describe a product operation between programs and automata that will assist in the verification of program transformations.

Notation Let  $\Sigma$  be an alphabet, i.e., a set of symbols, and let  $\Gamma$  be a subset of  $\Sigma$ . An infinite sequence  $u = u(0), u(1), \ldots$ , where  $u(i) \in \Sigma$  for all i, is said to be a "sequence over  $\Sigma$ ". For variables x, y denoting elements of  $\Sigma$ , the notation  $x =_{\Gamma} y$  (read as "x and y agree on  $\Gamma$ ") denotes the predicate where either x and y are both not in  $\Gamma$ , or x and y are both in  $\Gamma$  and x = y. For a sequence u over  $\Sigma$ , the notation  $u|_{\Gamma}$  (read as "u projected to  $\Gamma$ ") denotes the sub-sequence of u formed by elements in  $\Gamma$ . The operator compress(v) =  $v|_{\Sigma}$ , applied to a sequence v over  $\Sigma \cup \{\varepsilon\}$ , removes all  $\varepsilon$  symbols in v to form a sequence over  $\Sigma$ . For a bundle of traces  $w = (w_1, \ldots, w_k)$  where each trace is an infinite sequence of  $\Sigma$ , the operator zip(w) defines an infinite sequence over  $\Sigma^k$  obtained by choosing successive elements from each trace. In other words, u = zip(w) is defined by  $u(i) = (w_1(i), \ldots, w_k(i))$ , for all i. The operator unzip is its inverse.

#### 3.1 Programs as Transition Systems

A program is represented as a transition system  $S = (C, \Sigma, \iota, \rightarrow)$ :

- -C is a set of program states, or configurations;
- $-\Sigma$  is a set of observables, partitioned into input, I, and output, O;
- $-\iota \in C$  is the initial configuration;
- $-(\rightarrow) \subseteq C \times (\Sigma \cup \{\varepsilon\}) \times C$  is the transition relation.

Transitions labeled by input symbols in I represent instructions in the program that read input values, while transitions labeled by output symbols in O represent instructions that produce observable outputs. Transitions labeled by  $\varepsilon$  represent internal transitions where the state of the program changes without any observable effect.

An execution is an infinite sequence of transitions  $(c_0, w_0, c_1)(c_1, w_1, c_2) \dots \in (\to)^{\omega}$  such that  $c_0 = \iota$  and adjacent transitions are connected as shown. (We may write this as the alternating sequence  $c_0, w_0, c_1, w_1, c_2, \dots$ ) To ensure that every execution is infinite, we assume that  $(\to)$  is left-total. To model programs with finite executions, we assume that the alphabet has a special termination symbol  $\bot$ , and add a transition  $(c, \bot, c)$  for every final state c. We also assume that there is no infinite execution where the transition labels are always  $\varepsilon$  from some point on.

An execution  $x=(c_0,w_0,c_1)(c_1,w_1,c_2)\dots$  has an associated trace, denoted trace(x), given by the sequence  $w_0,w_1,\dots$  over  $\Sigma \cup \{\varepsilon\}$ . The compressed trace of execution x, compress(trace(x)), is denoted ctrace(x). The final assumption above ensures that the compressed trace of an infinite execution is also infinite. The sequence of states on an execution x is denoted by states(x).

#### Attack Models as Extended Transition Systems 3.2

The choice of how to model a program as a transition system depends on the properties one would like to verify. For correctness, it is enough to use the standard input-output semantics of the program. To represent security properties, however, it is usually necessary to extend this base semantics to bring out interesting features. Such an extension typically adds auxiliary state and new observations needed to model an attack. For example, if an attack is based on program location, that is added as an auxiliary state component in the extended program semantics. Other examples of such structures are modeling a program stack as an array with a stack pointer, explicitly tracking the addresses of memory reads and writes, and distinguishing between cache and main memory accesses. These extended semantics are roughly analogous to the leakage models of [3]. The base transition system is extended to one with a new state space, denoted  $C_e$ ; new observations, denoted  $O_e$ ; and a new alphabet,  $\Sigma_e$ , which is the union of  $\Sigma$  with  $O_e$ . The extensions do not alter input-output behavior; formally, the original and extended systems are bisimular with respect to  $\Sigma$ .

#### 3.3 Büchi Automata over trace bundles

A Büchi automaton over a bundle of k infinite traces over  $\Sigma_e$  is specified as  $A = (Q, \Sigma_e^k, \iota, \Delta, F)$ , where:

- Q is the state space of the automaton;  $\Sigma_e^k$  is the alphabet of the automaton, each element is a k-vector over  $\Sigma_e$ ;
- $\begin{array}{l} -\iota \stackrel{e}{\in} Q \text{ is the initial state;} \\ -\Delta \subseteq Q \times \varSigma_e^k \times Q \text{ is the transition relation;} \\ -F \subseteq Q \text{ is the set of accepting states.} \end{array}$

A run of A over a bundle of traces  $t = (t_1, \ldots, t_k) \in (\Sigma^{\omega})^k$  is an alternating sequence of states and symbols, of the form  $(q_0 = \iota), a_0, q_1, a_1, q_2, \ldots$  where for each  $i, a_i = (t_1(i), \dots, t_k(i))$  — that is,  $a_0, a_1, \dots$  equals zip(t) — and  $(q_i, a_i, q_{i+1})$ is in the transition relation  $\Delta$ . The run is accepting if a state in F occurs infinitely often along it. The language accepted by A, denoted by  $\mathcal{L}(A)$ , is the set of all k-trace bundles that are accepted by A.

Automaton-Program Product In verification, the set of traces of a program that violate a property can be represented by an automaton that is the product of the program with an automaton for the negation of that property. Security properties may require analyzing multiple traces of a program; therefore, we define the analogous automaton as a product between an automaton A for the negation of the security property and the k-fold composition  $P^k$  of a program P. For simplicity, assume for now that the program P contains no  $\varepsilon$ -transitions. Programs with  $\varepsilon$ -transitions can be handled by converting A over  $\Sigma_e^k$  into a new

automaton  $\hat{A}$  over  $(\Sigma_e \cup \{\varepsilon\})^k$  (see Appendix for details). Let  $A = (Q^A, \Sigma_e^k, \Delta^A, \iota^A, F^A)$  be a Büchi automaton (over a k-trace bundle) and  $P = (C, \Sigma_e, \iota, \to)$  be a program. The product of A and  $P^k$ , written  $A \times P^k$ , is a Büchi automaton  $B = (Q^B, \Sigma_e^k, \Delta^B, \iota^B, F^B)$ , where:

```
\begin{array}{l} -\ Q^B = Q^A \times C^k; \\ -\ \iota^B = (\iota^A, (\iota, \ldots, \iota)); \\ -\ ((q,s), u, (q',s')) \text{ is in } \Delta^B \text{ if, and only if, } (q,u,q') \text{ is in } \Delta^A, \text{ and } (s_i, u_i, s_i') \text{ is } \\ \text{in } (\rightarrow) \text{ for all } i; \\ -\ (q,s) \text{ is in } F^B \text{ iff } q \text{ is in } F^A. \end{array}
```

**Lemma 1.** Trace  $zip(t_1,...,t_k)$  is in  $\mathcal{L}(A \times P^k)$  if, and only if,  $zip(t_1,...,t_k)$  is in  $\mathcal{L}(A)$  and, for all  $i, t_i = trace(x_i)$  for some execution  $x_i$  of P.

**Bisimulations** For programs  $S = (C^S, \Sigma_e, \iota^S, \to^S)$  and  $T = (C^T, \Sigma_e, \iota^T, \to^T)$ , and a subset I of  $\Sigma_e$ , a relation  $B \subseteq C^T \times C^S$  is a bisimulation for I if:

- 1.  $(\iota^T, \iota^S) \in B$ ;
- 2. For every (t, s) in B and (t, v, t') in  $(\to^T)$  there is u and s' such that (s, u, s') is in  $(\to^S)$  and  $(t', s') \in B$  and u = v.
- 3. For every (t, s) in B and (s, u, s') in  $(\rightarrow^S)$  there is v and t' such that (t, v, t') is in  $(\rightarrow^T)$  and  $(t', s') \in B$  and u = v.

# 4 Formulating Security Preservation

A temporal correctness property  $\varphi$  is expressed as a set of infinite traces. Many security properties can only be described as properties of pairs or tuples of traces. A standard example is that of noninterference, which models potential leakage of secret inputs: if two program traces differ only in secret inputs, they should be indistinguishable to an observer that can only view non-secret inputs and outputs. The general notion is that of a hyperproperty [26,6], which is a set containing sets of infinite traces; a program satisfies a hyperproperty H if the set of all compressed traces of the program is an element of H. Linear Temporal Logic (LTL) is commonly used to express correctness properties. Our formulation of security properties is an extension of the logic HyperLTL, which can express common security properties including several variants of noninterference [5].

A security property  $\varphi$  has the form  $(Q_1\pi_1,\ldots,Q_n\pi_k:\kappa(\pi_1,\ldots,\pi_k))$ , where the  $Q_i$ 's are first-order quantifiers over trace variables, and  $\kappa$  is set of k-trace bundles, described by a Büchi automaton whose language is the *complement* of  $\kappa$ . This formulation borrows the crucial notion of trace quantification from HyperLTL, while generalizing it, as automata are more expressive than LTL, and atomic propositions may hold of k-vectors rather than on a single trace.

The satisfaction of property  $\varphi$  by a program P is defined in terms of the following finite two-player game, denoted  $\mathcal{G}(P,\varphi)$ . The protagonist, Alice, chooses an execution of P for each existential quantifier position, while the antagonist, Bob, chooses an execution of P at each universal quantifier position. The choices are made in sequence, from the outermost to the innermost quantifier. A play of this game is a maximal sequence of choices. The outcome of a play is thus a "bundle" of program executions, say  $\sigma = (\sigma_1, \ldots, \sigma_k)$ . This induces a corresponding bundle of compressed traces,  $t = (t_1, \ldots, t_k)$ , where  $t_i = \mathsf{ctrace}(\sigma_i)$  for each i. This play is a win for Alice if t satisfies  $\kappa$  and a win for Bob otherwise.

A strategy for Bob is a function, say  $\xi$ , that defines a non-empty set of executions for positions i where  $Q_i$  is a universal quantifier, in terms of the earlier choices  $\sigma_1, \ldots, \sigma_{i-1}$ ; the choice of  $\sigma_i$  is from this set. A strategy for Alice is defined symmetrically. A strategy is winning for player X if every play following the strategy is a win for X. This game is determined, in that for any program P one of the players has a winning strategy. Satisfaction of a security property is defined by the following.

**Definition 1.** Program P satisfies a security property  $\varphi$ , written  $\models_P \varphi$ , if the protagonist has a winning strategy in the game  $\mathcal{G}(P,\varphi)$ .

#### 4.1 Secure Program Transformation

Let  $S = (C^S, \Sigma_e, \iota^S, \to^S)$  be the transition system representing the original source program and let  $T = (C^T, \Sigma_e, \iota^T, \to^T)$  be the transition system for the transformed target program. Any notion of secure transformation must imply the preservation property that if S satisfies  $\varphi$  and the transformation from S to T is secure for  $\varphi$  then T also satisfies  $\varphi$ . This property in itself is, however, too weak to serve as a definition of secure transformation.

Consider the transformation shown in Figure 1, with use(x) defined so that it terminates execution if the secret key x is invalid. As the source program violates non-interference by leaking the validity of the key, the transformation would be trivially secure if the preservation property is taken as the definition of secure transformation. But that conclusion is wrong: the leak introduced in the target program is clearly different and of a more serious nature, as the entire secret key is now vulnerable to attack.

This analysis prompts the formulation of a stronger principle for secure transformation. (Similar principles have been discussed in the literature, e.g., [11].) The intuition is that every instance and type of violation in T should have a matching instance and type of violation in S. To represent different types of violations, we suppose that the negated property is represented by a collection of automata, each checking for a specific type of violation.

**Definition 2.** A strategy  $\xi^S$  for the antagonist in  $\mathcal{G}(S,\varphi)$  (representing a violation in S) matches a strategy  $\xi^T$  for the antagonist in game  $\mathcal{G}(T,\varphi)$  (representing a violation in T) if for every maximal play  $u=u_1,\ldots,u_k$  following  $\xi^T$ , there is a maximal play  $v=v_1,\ldots,v_k$  following  $\xi^S$  such that (1) the two plays are input-equivalent, i.e.,  $u_i|_I=v_i|_I$  for all i, and (2) if u is accepted by the m-th automaton for the negated property, then v is accepted by the same automaton.

**Definition 3.** A transformation from S to T preserves security property  $\varphi$  if for every winning strategy for the antagonist in the game  $\mathcal{G}(T,\varphi)$ , there is a matching winning strategy for the antagonist in the game  $\mathcal{G}(S,\varphi)$ .

As an immediate consequence, we have the preservation property.

**Theorem 1.** If a transformation from S to T preserves security property  $\varphi$  and if S satisfies  $\varphi$ , then T satisfies  $\varphi$ .

In the important case where the security property is purely universal, of the form  $\forall \pi_1, \ldots, \forall \pi_k : \kappa(\pi_1, \ldots, \pi_k)$ , a winning strategy for the antagonist is simply a bundle of k traces, representing an assignment to  $\pi_1, \ldots, \pi_k$  that falsifies  $\kappa$ .

# 5 Refinement for Preservation of Universal Properties

We define an automaton-based refinement scheme that is sound for purelyuniversal properties  $\varphi$ , of the form  $(\forall \pi_1, \ldots, \forall \pi_k : \kappa(\pi_1, \ldots, \pi_k))$ . Section 8 generalizes this to properties with arbitrary quantifier prefixes. We assume for simplicity that programs S and T have no  $\varepsilon$ -transitions; we discuss how to remove this assumption at the end of the section. An automaton-based refinement scheme for preservation of  $\varphi$  is defined below.

**Definition 4.** Let S,T be programs over the same alphabet,  $\Sigma_e$ , and A be a Büchi automaton over  $\Sigma_e^k$ . Let I be a subset of  $\Sigma_e$ . A relation  $R \subseteq (Q^A \times (C^T)^k) \times (Q^A \times (C^S)^k)$  is a refinement relation from  $A \times T^k$  to  $A \times S^k$  for I if

- 1. Initial configurations are related, i.e.,  $((\iota^A, \iota^{T^k}), (\iota^A, \iota^{S^k}))$  is in R, and
- 2. Related states have matching transitions. That is, if  $((q,t),(p,s)) \in R$  and  $((q,t),v,(q',t')) \in \Delta^{A \times T^k}$ , there are u,p', and s' such that the following hold:
  - (a) ((p,s), u, (p',s')) is a transition in  $\Delta^{A \times S^k}$ ;
  - (b) u and v agree on I, that is,  $u_i =_I v_i$  for all i;
  - (c) the successor configurations are related, i.e.,  $((q',t'),(p',s')) \in R$ ; and
  - (d) acceptance is preserved, i.e., if  $q' \in F$  then  $p' \in F$ .

**Lemma 2.** If there exists a refinement from  $A \times T^k$  to  $A \times S^k$  then, for every sequence v in  $\mathcal{L}(A \times T^k)$ , there is a sequence u in  $\mathcal{L}(A \times S^k)$  such that u and v are input-equivalent.

Theorem 2 (Universal Refinement). Let  $\varphi = (\forall \pi_1, \dots, \pi_k : \kappa(\pi_1, \dots, \pi_k))$  be a universal security property; S and T be programs over a common alphabet  $\Sigma_e = \Sigma \cup O_e$ ;  $A = (Q, \Sigma_e^k, \iota, \Delta, F)$  be an automaton for the negation of  $\kappa$ ; and  $R \subseteq (Q \times (C^T)^k) \times (Q \times (C^S)^k)$  be a refinement relation from  $A \times T^k$  to  $A \times S^k$  for I. Then, the transformation from S to T preserves  $\varphi$ .

*Proof.* A violation of  $\varphi$  by T is given by a bundle of executions of T that violates  $\kappa$ . We show that there is an input-equivalent bundle of executions of S that also violates  $\kappa$ . Let  $x=(x_1,\ldots,x_k)$  be a bundle of executions of T that does not satisfy  $\kappa$ . By Lemma 1,  $v=\operatorname{zip}(\operatorname{trace}(x_1),\ldots,\operatorname{trace}(x_k))$  is accepted by  $A\times T^k$ . By Lemma 2, there is a sequence u accepted by  $A\times S^k$  that is input-equivalent to v. Again by Lemma 1, there is a bundle of executions  $y=(y_1,\ldots,y_k)$  of S such that  $u=\operatorname{zip}(\operatorname{trace}(y_1),\ldots,\operatorname{trace}(y_k))$  and y violates  $\kappa$ . As u and v are input equivalent,  $\operatorname{trace}(x_i)$  and  $\operatorname{trace}(y_i)$  are input-equivalent for all i, as required.  $\square$ 

The refinement proof rule for universal properties is implicit: a witness is a relation R from  $A \times T^k$  to  $A \times S^k$ ; this is valid if it satisfies the conditions set out in Definition 4. The theorem establishes the soundness of this proof

rule. Examples of witnesses for specific compiler transformations are given in Section 6, which also discusses SMT-based checking of the proof requirements.

To handle programs that include  $\varepsilon$ -transitions, we can convert the automaton A over  $\Sigma_e^k$  into a buffering automaton  $\hat{A}$  over  $(\Sigma_e \cup \{\varepsilon\})^k$ , such that  $\hat{A}$  accepts  $\mathsf{zip}(v_1,\ldots,v_k)$  iff A accepts  $\mathsf{zip}(\mathsf{compress}(v_1),\ldots,\mathsf{compress}(v_k))$ . The refinement is then defined over  $\hat{A} \times S^k$  and  $\hat{A} \times T^k$ . Details can be found in the Appendix. Another useful extension is the addition of stuttering, which can be necessary for example when a transformation removes instructions. Stuttering relaxes Definition 4 to allow multiple transitions on the source to match a single transition on the target, or vice-versa. This is a standard technique for verification [4] and one-step formulations suitable for SMT solvers are known (cf. [18,14]).

# 6 Checking Transformation Security

This section first describes how to construct the SMT formula that checks the correctness of a given refinement relation. Next, it demonstrates through concrete examples how to express a refinement relation for specific program transformations.

#### 6.1 Refinement Check

Assume that the refinement relation R, the transition relations  $\Delta$ ,  $(\rightarrow_T)$  and  $(\rightarrow_S)$  and the set of accepting states F are described by SMT formulas over variables ranging over states and alphabet symbols.

To verify that the formula R is indeed a refinement, we perform an inductive check following the definition of refinement given in Definition 4. To prove the base case, which says that the initial states of  $A \times T^k$  and  $A \times S^k$  are related by R, we simply evaluate the formula on the initial states.

Proving the inductive step again follows from Definition 4, which states that the transition relations of the automata must preserve membership in R. The correctness of the inductive step would be expressed by an SMT query of the shape  $(\forall q^T, q^S, p^T, t, s, t', \sigma^T : (\exists \sigma^S, p^S, s' : \varphi_1 \to \varphi_2))$ , where:

$$\varphi_1 \equiv R((q^T, t), (q^S, s)) \wedge \Delta(q^T, \sigma^T, p^T) \wedge \bigwedge_{i=1}^k (t_i \xrightarrow{\sigma_i^T}_T t_i')$$

$$\varphi_2 \equiv \Delta(q^S, \sigma^S, p^S) \wedge \bigwedge_{i=1}^k (s_i \xrightarrow{\sigma_i^S}_S s_i') \wedge \bigwedge_{i=1}^k (\sigma_i^T =_I \sigma_i^S)$$

$$\wedge R((p^T, t'), (p^S, s')) \wedge (F(p^T) \to F(p^S))$$

Note that this formula has a quantifier alternation, which is hard for SMT solvers to handle. However, the formula can be reduced to a validity check by providing Skolem functions from the universal to the existential variables. We expect the compiler to provide these functions. As we will see in the examples below, in many cases the compiler can choose simple-enough Skolem functions

that the validity of the formula can be verified using only equality reasoning, making it unnecessary to even expand  $\Delta$  and F to their definitions. More generally, a compiler writer must have a proof in mind for each optimization and should therefore be able to provide the necessary Skolem functions to match the refinement relation.

# 6.2 Refinement Relations for Compiler Optimizations

We consider three common optimizations below. In addition, further examples for dead-branch elimination, expression flattening, loop peeling and register spilling can be found in the Appendix. All transformations were based on the examples in [3].

**Example 1: Constant Folding** Section 2 presented an example of a program transformation by constant folding. We now proceed to show how a refinement relation can be defined to serve as a witness for the security of this transformation, so its validity can be checked using an SMT solver as described above.

Recall that states of S and T are of the form  $(\alpha, \ell)$ , where  $\alpha : \mathcal{V} \to \text{Int}$  and  $\ell \in \text{Loc}$ . Then, R can be expressed by the following formula over states  $q^T, q^S$  of the automaton A and states t of  $T^k$  and s of  $S^k$ , where  $t_i = (\alpha_i^T, \ell_i^T)$ :

$$\begin{split} (q^T = q^S) \wedge (t = s) \wedge \bigwedge_{i=1}^k (\ell_i^T = \texttt{L3} \to \alpha_i^T(\texttt{y}) = 42) \\ \wedge \bigwedge_{i=1}^k (\ell_i^T = \texttt{L4} \to \alpha_i^T(\texttt{z}) = 1) \wedge \bigwedge_{i=1}^k (\ell_i^T = \texttt{L5} \to \alpha_i^T(\texttt{x}) = 0) \end{split}$$

Since this is a simple transformation, equality between states is all that is needed to establish a refinement. However, to allow the refinement to be verified automatically, the relation also has to carry information about the constant values at specific points in the program. In general, if the transformation relies on the fact that at location  $\ell$  variable v has constant value c, the constraint  $\bigwedge_{i=1}^k (\ell_i^T = \ell \to \alpha_i^T(v) = c)$  is added to R.

 $\bar{R}$  can be checked using the SMT query described in Section 6.1. Note that for this particular transformation the compiler can choose Skolem functions that assign  $\sigma^S = \sigma^T$  and  $p^S = p^T$ . In this case, from  $(q^T = q^S)$  (given by the refinement relation) and  $\Delta(q^T, \sigma^T, p^T)$  the solver can automatically infer  $\Delta(q^S, \sigma^S, p^S)$ ,  $(\sigma_i^T =_I \sigma_i^S)$  and  $F(p^T) \to F(p^S)$  using only equality reasoning. Therefore, the refinement check is independent of the security property in this case. This applies to many other transformations that occur in practice as well, since their reasons for preserving security are usually simple.

**Example 2: Common-Branch Factorization** Common-branch factorization is a program optimization applied to conditional blocks where the instructions at the beginning of the *then* and *else* blocks are the same. If the condition does

not depend on a variable modified by the common instruction, this instruction can be moved outside of the conditional. Consider for example:

```
// Source program S
                                    // Target program T
                                   L1: a := arr[0];
L1: if (j < arr_size) {
        a := arr[0];
                                   L2: if (j < arr_size) {
L2:
                                            b := arr[j];
L3:
        b := arr[j];
                                   L3:
L4: } else {
                                   L4: } else {
L5:
        a := arr[0];
                                   L5:
L6:
        b := arr[arr_size - 1];
                                            b := arr[arr_size - 1];
                                   L6:
L7: }
                                   L7: }
```

Suppose that the attack model allows the attacker to observe memory accesses, represented by the index j of every array access  $\mathtt{arr[j]}$ . We assume that other variables are stored in registers rather than memory (see Appendix for a discussion on register spilling). Under this attack model the compressed traces produced by T are identical to the ones of S, therefore the transformation will be secure regardless of the security property  $\varphi$ . However, because the order of instructions is different, a more complex refinement relation R will be needed, compared to constant folding:

$$\begin{split} ((t=s) \land (q^T=q^S)) \lor \bigwedge_{i=1}^k ((\ell_i^T = \texttt{L2}) \\ & \land ((\alpha_i^S(\texttt{i}) < \alpha_i^S(\texttt{arr\_size})) ? (\ell_i^S = \texttt{L2}) : (\ell_i^S = \texttt{L5})) \\ & \land (\alpha_i^T = \alpha_i^S[\texttt{a} := \texttt{arr}[\texttt{O}]]) \land \varDelta(q^S, (0, \dots, 0), q^T) \end{split}$$

The refinement relation above expresses that the states of the programs and the automata are identical except when T has executed the factored-out instruction but S hasn't yet. At that point, T is at location L2 and S is either at location L2 or L5, depending on how the guard was evaluated. Note that it is necessary for R to know that the location of S depends on the evaluation of the guard, so that it can verify that at the next step T will follow the same branch. The states of  $\hat{A} \times S^k$  and  $\hat{A} \times T^k$  are then related by saying that after updating  $\mathbf{a} := \text{arr}[\mathbf{0}]$  on every track of S the two states will be the same. Note that since this instruction produces an observation representing the index of the array access, the states of the automata are related by  $\Delta(q^S, (0, \ldots, 0), q^T)$ , indicating that the access has been observed by  $\hat{A} \times T^k$  but not yet by  $\hat{A} \times S^k$ .

**Example 3: Switching Instructions** This optimization switches two sequential instructions if the compiler can guarantee that the program's behavior will not change. For example, consider the following source and target programs:

```
// Source program S
                                      // Target program T
L1: int a[10], b[10];
                                     L1: int a[10], b[10];
                                     L2: a[0] := secret_input();
L2: a[0] := secret_input();
L3: b[0] := secret_input();
                                     L3: b[0] := secret_input();
L4: for(int j:=1; j<10; j++){
                                     L4: for(int j:=1; j<10; j++){
L5:
      a[j] := b[j-1];
                                      L5:
                                            b[j] := a[j-1];
L6:
      b[j] := a[j-1];
                                      L6:
                                            a[j] := b[j-1];
L7:
      public_output(j);
                                      L7:
                                            public_output(j);
L8: }
                                      L8: }
```

Note that the traces produced by T and S are identical. Therefore, a refinement relation for this pair of programs can be given by the following formula, regardless of the security property under verification:

$$\begin{split} (q^S = q^T) \wedge \bigwedge_{i=1}^k (\ell_i^S = \ell_i^T) \wedge (\ell_i^S \neq \texttt{L6} \rightarrow \alpha_i^S = \alpha_i^T) \\ \wedge (\ell_i^S = \texttt{L6} \rightarrow \alpha_i^S [\texttt{b[j]} := \texttt{a[j-1]}] = \alpha_i^T [\texttt{a[j]} := \texttt{b[j-1]}]) \end{split}$$

The formula expresses that the state of the source and target programs is the same except between executing the two switched instructions. At that point, the state of the two programs is related by saying that after executing the second instruction in each of the programs they will again have the same state.

More generally, a similar refinement relation can be used for any source-target pair that satisfies the assumptions that (a) neither of the switched instructions produces an observable output, and (b) after both switched instructions are executed, the state of the two programs is always the same. All that is necessary in this case is to replace L6 by the appropriate location  $\ell^S_{switch}$  where the switch happens and  $\alpha^S_i[b[j] := a[j-1]] = \alpha^T_i[a[j] := b[j-1]]$  by an appropriate formula  $\delta(\alpha^S_i, \alpha^T_i)$  describing the relationship between the states of the two programs at that location.

If the instructions being switched do produce observations, setting up the refinement relation becomes harder. This is due to the fact that the relationship  $(q^S = q^T)$  might not hold in location  $\ell^S_{switch}$ , but expressing the true relationship between  $q^S$  and  $q^T$  is complex and might require knowledge of the state of all copies of S and T at once. It becomes simpler for some special cases, for example if the different copies of S and T are guaranteed to be synchronized (i.e., it is always the case that  $\ell^S_i = \ell^S_j$  and  $\ell^T_i = \ell^T_j$ , for all i and j), or if only one of the instructions produces an observation. Details can be found in the Appendix.

# 7 Connections to Existing Proof Rules

We establish connections to known proof rules for preservation of the non-interference [7,8,17] and constant-time [3] properties. We show that under the assumptions of those rules, there is a simple and direct definition of a relation

that meets the automaton-based refinement conditions for automata representing these properties. The automaton-based refinement method is thus general enough to serve as a uniform replacement for the specific proof methods.

#### 7.1 Constant Time

We first consider the lockstep CT-simulation proof rule introduced in [3] to show preservation of the constant-time property. For lack of space, we refer the reader to the original paper for the precise definitions of observational non-interference (Definition 1), constant-time as observational non-interference (Definition 4), lockstep simulation (Definition 5, denoted  $\approx$ ), and lockstep CT-simulation (Definition 6, denoted ( $\equiv_S, \equiv_C$ )).

We do make two minor adjustments to better fit the automaton notion, which is based on trace rather than state properties. First, we add a dummy initial source state  $\hat{S}(i)$  with a transition with input label i to the actual initial state S(i); and similarly for the target program, C. Secondly, we assume that a final state has a self-loop with a special transition label,  $\bot$ . Then the condition  $(b \in S_f \leftrightarrow b' \in S_f)$  from Definition 1 in [3] is covered by the (existing) label equality t = t'. With these changes, the observational non-interference property can be represented in negated form by the automaton shown in Figure 2, which simply looks for a sequence starting with an initial pair of input values satisfying  $\phi$  and ending in unequal transition labels. The states are I (initial), S (sink), M (mid), and F (fail), which is also the accepting state.

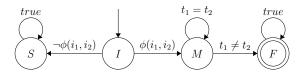


Fig. 2. A Büchi automaton for the negation of the constant-time property.

We now define the automaton-based relation, using the notation in Theorem 1 of [3]. Define relation R by  $(q, \alpha, \alpha')R(p, a, a')$  if  $a \approx \alpha, a' \approx \alpha'$ , and

- 1. p = F, i.e., p is the fail state, or
- 2. p = q = S, or
- 3. p=q=I, and  $\alpha=\hat{C}(i)$ ,  $\alpha'=\hat{C}(i')$ ,  $\alpha=\hat{S}(i)$ ,  $\alpha=\hat{S}(i')$ , for some i,i', or
- 4. p = q = M, and  $\alpha \equiv_C \alpha'$ , and  $a \equiv_S a'$ .

**Theorem 3.** If  $(\equiv_S, \equiv_C)$  is a lockstep CT-simulation with respect to the lockstep simulation  $\approx$ , the relation R is a valid refinement relation.

*Proof.* Every initial state of  $\mathcal{A} \times C^2$  has a related initial state in  $\mathcal{A} \times S^2$ . As related configurations are pairwise connected by  $\approx$ , which is a simulation, it follows

that any pairwise transition from a C-configuration is matched by a pairwise transition from the related S-configuration, producing states b, b' and  $\beta, \beta'$  that are pointwise related by  $\approx$ . These transitions have identical input labels, as the only transitions with input labels are those from the dummy initial states.

The remaining question is whether the successor configurations are connected by R. We reason by cases.

First, if p = F, then p' is also F. Hence, the successor configurations are related. This is also true of the second condition, where p = q = S, as the successor states are p' = q' = S.

If p=q=I the successor states are  $\beta=C(i), \beta'=C(i')$  and b=S(i), b'=S(i'), and the successor automaton state is either p'=q'=S, if  $\phi(i,i')$  does not hold, or p'=q'=M, if it does. In the first possibility, the successor configurations are related by the second condition; in the second, they are related by the final condition, as  $C(i) \equiv_C C(i')$  and  $S(i) \equiv_S S(i')$  hold if  $\phi(i,i')$  does [Definition 6 of [3]].

Finally, consider the interesting case where p=q=M. Let  $\tau,\tau'$  be the transition labels on the pairwise transition in C, and let t,t' be the labels on the corresponding pairwise transition in S. We consider two cases:

- (1) Suppose  $t \neq t'$ . Then p' = F and the successor configurations are related, regardless of p'.
- (2) Otherwise, t = t' and p' = M. By CT-simulation [Definition 6 of of [3]:  $a \equiv_S a'$  and  $\alpha \equiv_C \alpha'$  by the relation R], it follows that  $b \equiv_S b'$  and  $\beta \equiv_C \beta'$  hold, and  $\tau = \tau'$ . Thus, the successor automaton state on the C-side is q' = M and the successor configurations are related by the final condition.

This completes the case analysis. Finally, the definition of R implies that if q=F then p=F, as required.

#### 7.2 Non-Interference

Refinement-based proof rules for preservation of non-interference have been introduced in [7,8,17]. The rules are not identical but are substantially similar in nature, providing conditions under which an ordinary simulation relation,  $\prec$ , between programs C and S implies preservation of non-interference. We choose the rule from [8], which requires, in addition to the requirement that  $\prec$  is a simulation preserving input and output events, that (a) A final state of C is related by  $\prec$  only to a final state of S (precisely, both are final or both non-final), and (b) If  $t_0 \prec s_0$  and  $t_1 \prec s_1$  hold, and all states are either initial or final, then the low variables of  $t_0$  and  $t_1$  are equal iff the low variables of  $s_0$  and  $s_1$  are equal.

We make two minor adjustments to better fit the automaton notion, which is based on trace rather than state properties. First, we add a dummy initial source state  $\hat{S}(i)$  with a transition that exposes the value of local variables and moves to the actual initial state S(i) (i is the secret input); and similarly for the target program, C. Secondly, we assume that a final state has a self-loop with a special transition label that exposes the value of local variables on termination. With these changes, the negated non-interference can be represented by the

automaton shown in Figure 3. It accepts an pair of execution traces if, and only if, initially the low-variables on the two traces have identical values, and either the corresponding outputs differ at some point, or final values of the low-variables are different. (The transition conditions are written as Boolean predicates which is a readable notation for describing a set of pairs of events; e.g., the  $Low_1 \neq Low_2$  transition from state I represents the set of pairs (a,b) where a is the init(Low = i) event, b is the init(Low = j) event, and  $i \neq j$ .)

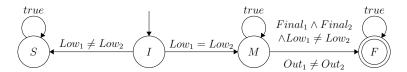


Fig. 3. A Büchi automaton for the negation of the non-interference property.

Define the automaton-based relation R by  $(q, t_0, t_1)R(p, s_0, s_1)$  if p = q and  $t_0 \prec s_0$  and  $t_1 \prec s_1$ . We have the following theorem.

**Theorem 4.** If the simulation relation  $\prec$  between C and S satisfies the additional properties needed to preserve non-interference, then R is a refinement.

*Proof.* Consider  $(q, t_0, t_1)R(p, s_0, s_1)$ . As  $\prec$  is a simulation, for any joint transition from  $(t_0, t_1)$  to  $(t'_0, t'_1)$ , there is a joint transition from  $(s_0, s_1)$  to  $(s'_0, s'_1)$  such that  $t'_0 \prec s'_0$  and  $t'_1 \prec s'_1$  holds. This transition preserves input and output values, as  $\prec$  is an input-output preserving simulation.

We have only to establish that the automaton transitions also match up. If the automaton state is either F or S, the resulting state is the same, so by the refinement relation, we have p' = p = q = q'.

Consider q = I. If q' = S then the values of the low variables in  $t_0, t_1$  differ; in which case, by condition (b), those values differ in  $s_0, s_1$  as well, so p' is also S. Similarly, if q' = M, then p' = M.

Consider q = M. If q' = F then either (1)  $t_0, t_1$  are both final states and the values of the low variables differ; in which case, by condition (b), those values differ in  $s_0, s_1$  as well, so p' is also F, or (2) the outputs of the transitions from  $t_0, t_1$  to  $t'_0, t'_1$  differ; in which case, as  $\prec$  preserves outputs, this is true also of the transition from  $s_0, s_1$  to  $s'_0, s'_1$ , so p' is also F. If q' = M then the outputs are identical and one of  $t_0, t_1$  is non-final; in which case, by condition (a), that is true also for the pair  $s_0, s_1$ , so p' is also M.

Finally, by the relation R, if q = F, the accepting state, then p = F as well. This completes the case analysis and the proof.

# 8 Witnessing General Security Properties

The notion of refinement presented in Section 5 suffices for universal hyperproperties, as in that case a violation corresponds to a bundle of traces rejected by the

automaton. Although many important hyperproperties are universal in nature, there are cases that require quantifier alternation. One example is generalized noninterference, as formalized in [5], which says that for every two traces of a program, there is a third trace that has the same high inputs as the first but is indistinguishable from the second to a low-clearance individual. A violation for such hyperproperties, as defined in Section 4, is not simply a bundle of traces, but rather a winning strategy for the antagonist in the corresponding game. A refinement relation does not suffice to match winning strategies. Therefore, we also introduce an input-equivalent bisimulation B from T to S, which is used in a back-and-forth manner to construct a matching winning strategy for the antagonist in  $\mathcal{G}(S,\varphi)$  from any winning strategy for the antagonist in  $\mathcal{G}(T,\varphi)$ .

A bisimulation B ensures, by induction, that any infinite execution in T has an input-equivalent execution in S, and vice-versa. For an execution x of T, we use B(x) to denote the set of input-equivalent executions in S induced by B, which is non-empty. The symmetric notion,  $B^{-1}(y)$ , refers to input-equivalent executions in T induced by B for an execution y of S.

**Definition 5.** Let  $\xi^T$  be a strategy for the antagonist in  $\mathcal{G}(T,\varphi)$  and B be a bisimulation between T and S. Then, the strategy  $\xi^S = \mathcal{S}(\xi^T, B)$  for the antagonist in  $\mathcal{G}(S,\varphi)$  proceeds in the following way to produce a play  $(y_1,\ldots,y_k)$ :

- For every i such that  $\pi_i$  is existentially quantified, let  $y_i$  be chosen by the protagonist in  $\mathcal{G}(S,\varphi)$ . Choose an input-equivalent execution  $x_i$  from  $B^{-1}(y_i)$ ;
- For every i such that  $\pi_i$  is universally quantified, choose  $x_i$  in  $\xi^T(x_1, \dots, x_{i-1})$  and choose  $y_i$  from  $B(x_i)$ .

Thus, the bisimulation helps define a strategy  $\xi^S$  to match a winning antagonist stategy  $\xi^T$  in T. We can establish that this stategy is winning for the antagonist in S in two different ways. First, we do so under the assumption that S and T are input-deterministic, i.e., any two executions of the program with the same input sequence have the same observation sequence. This is a reasonable assumption, covering sequential programs with purely deterministic actions.

**Theorem 5.** Let S and T be input-deterministic programs over the same input alphabet I. Let  $\varphi$  be a general security property with automaton A representing the negation of its kernel  $\kappa$ . If there exists (1) a bisimulation B from T to S, and (2) a refinement relation R from  $A \times T^k$  to  $A \times S^k$  for I, then T securely refines S for  $\varphi$ .

*Proof.* We have to show, from Definition 3, that for any winning strategy  $\xi^T$  for the antagonist in  $\mathcal{G}(T,\varphi)$ , there is a matching winning stategy  $\xi^S$  in  $\mathcal{G}(S,\varphi)$ . Let  $\xi^S = \mathcal{S}(\xi^T,B)$ . Let  $y = (y_1,\ldots,y_k)$  be the bundle of executions resulting from a play following the strategy  $\xi^S$ , and  $x = (x_1,\ldots,x_k)$  the corresponding bundle resulting from  $\xi^T$ . By construction, y and x are input-equivalent.

Since  $\xi^T$  is a winning strategy, the trace of x is accepted by  $A \times T^k$ . Then, from the refinement R and Lemma 2, there is a bundle  $z = (z_1, \ldots, z_k)$  accepted by  $A \times S^k$  that is input-equivalent to x. Therefore, z is a win for the antagonist. Since

z is input-equivalent to x, it is also input-equivalent to y. Input-determinism requires that z and y are identical, so y is also a win for the antagonist. Thus,  $\xi^S$  is a winning strategy for the antagonist in  $\mathcal{G}(S,\varphi)$ . 

If S and T are not input-deterministic, a new notion of refinement is defined that intertwines the automaton-based relation, R, with the bisimulation, B. A relation  $R \subseteq (Q^A \times (C^T)^k) \times (Q^A \times (C^S)^k)$  is a refinement relation from  $A \times T^k$  to  $A \times S^k$  for I relative to  $B \subseteq C^T \times C^S$ , if

- 1.  $((\iota^A, \iota^{T^k}), (\iota^A, \iota^{S^k}))$  is in R and  $(\iota_i^{T^k}, \iota_i^{S^k}) \in B$  for all i; and 2. If ((q, t), (p, s)) is in R,  $(t_i, s_i)$  is in B for all i, ((q, t), v, (q', t')) is in  $\Delta^{A \times T^k}$ . (s, u, s') is in  $(\rightarrow^{S^k})$ , u and v agree on I, and  $(t'_i, s'_i) \in B$ , there is p' such that all of the following hold:
  - (a)  $((p, s), u, (p', s')) \in \Delta^{A \times S^k}$ ;
  - (b)  $((q', t'), (p', s')) \in R$ ;
  - (c) if  $q' \in F$  then  $p' \in F$ .

Typically, a refinement relation implies, as in Lemma 2, that a run in  $A \times T^k$ is matched by some run in  $A \times S^k$ . The unusual refinement notion above instead considers already matching executions of T and S, and formulates an inductive condition under which a run of A on the T-execution is matched by a run on the S-execution. The result is the following theorem, establishing the new refinement rule, where the witness is the pair (R, B).

**Theorem 6.** Let S and T be programs over the same input alphabet I. Let  $\varphi$  be a general security property with automaton A representing its kernel  $\kappa$ . If there exists (1) a bisimulation B from T to S, and (2) a relation R from  $A \times T^k$  to  $A \times S^k$  that is a refinement relative to B, then T securely refines S for  $\varphi$ .

#### 8.1 Checking General Refinement Relations

The main difference when checking security preservation of general hyperproperties, compared to the purely-universal properties handled in Section 5, is the necessity of the compiler to provide also the bisimulation B as part of the witness. The verifier must then check also that B is a bisimulation, which can be performed inductively using SMT queries in a similar way to the refinement itself. In the case that the language semantics guarantee input-determinism, as described above, then Theorem 5 holds and checking B and R separately is sufficient. Otherwise, the check for R described in Section 6.1 has to be modified to follow Theorem 6 by checking if R is a refinement relative to B.

Note that appropriate formulas B(t,s) describing bisimulations can be extracted from the examples of refinement relations given in Section 6:

- 1. Constant Folding:  $(t = s) \wedge (\ell^T = L3 \rightarrow \alpha^T(y) = 42) \wedge (\ell^T = L4 \rightarrow 2) \wedge (\ell^T = L4) \wedge (\ell^T =$  $\alpha^T(\mathbf{z}) = 1) \land (\ell^T = \mathbf{L5} \rightarrow \alpha^{T}(\mathbf{x}) = 0)$
- 2. Common-Branch Factorization:  $(t=s) \lor ((\ell^T=\texttt{L2}) \land ((\alpha^S(\texttt{i}) < \alpha^S(\texttt{arr\_size}))? (\ell^S=\texttt{L2}) : (\ell^S=\texttt{L5})) \land (\alpha^T=\alpha^S[\texttt{a}:=\texttt{arr}[\texttt{0}]]))$

3. Switching Instructions: 
$$(\ell^S = \ell^T) \wedge (\ell^S \neq L6 \rightarrow \alpha^S = \alpha^T) \wedge (\ell^S = L6 \rightarrow \alpha^S[b[j] := a[j-1]] = \alpha^T[a[j] := b[j-1]])$$

Note the similarities between the bisimulations above and the refinement relations from Section 6. When the transformation does not alter the observable behavior of a program, it is often the case that the refinement relation between  $\hat{A} \times T^k$  and  $\hat{A} \times S^k$  is essentially formed by the k-product of a bisimulation between T and S across the several tracks.

### 9 Discussion and Related Work

This work tackles the important problem of ensuring that optimizations carried out by a compiler do not break vital security properties of the source program. We propose a methodology based on property-specific refinement rules, with the refinement relations (witnesses) being generated at compile time and validated independently by a generic refinement checker. This structure ensures that neither the code of the compiler nor the witness generator have to be formally verified in order to obtain a formally verifiable conclusion. It is thus eminently suited to production compilers, which are large and complex, and are written in hard-to-formalize languages such as C or C++.

The refinement rules are synthesized from an automaton-theoretic definition of a security property. This construction applies to a broad range of security properties, including those specifiable in the HyperLTL logic [5]. When applied to automaton-based formulations of the non-interference and constant-time properties, the resulting proof rules are essentially identical to those developed in the literature in [7,8,17] for non-interference and in [3] for constant-time. Manna and Pnueli show in a beautiful paper [15] how to derive custom proof rules for deductive verification of a LTL property from an equivalent Büchi automaton; our constructions are inspired by this work.

Refinement witnesses are in a form that is composable: i.e., for a security property  $\varphi$ , if R is a refinement relation establishing a secure transformation from A to B, while R' witnesses a secure transformation from B to C, then the relational composition R; R' witnesses a secure transformation from A to C. Thus, by composing witnesses for each compiler optimization, one obtains an end-to-end witness for the entire optimization pipeline.

Other approaches to secure compilation include full abstraction, proposed in [1] (cf. [22]), and trace-preserving compilation [23]. These are elegant formulations but difficult to check fully automatically, and hence not suitable for translation validation. The theory of hyperproperties [6] includes a definition of refinement in terms of language inclusion (i.e., T refines S if the language of T is a subset of the language of S), which implies that any subset-closed hyperproperty is preserved by refinement in that sense. Language inclusion is also not directly checkable and thus cannot be used for translation validation. The refinement theorem in this paper for universal properties (which are subset-closed) uses a tighter step-wise inductive check that is suitable for automated validation.

Translation validation through compiler-generated refinement relations arises from work on "Credible Compilation" by [25,16] and "Witnessing" by [20]. As the compiler and the witness generator do not require formal verification, the size of the trusted code base shrinks substantially. Witnessing also requires much less effort than a full mathematical proof: as observed in [19], a mathematical correctness proof of SSA (Static Single Assignment) conversion in Coq is about 10,000 lines [29], while refinement checking can be implemented in around 1,500 lines of code, most of which comprises a witness validator which can be reused across different transformations. Our work shows how to extend this concept, originally developed for correctness checking, to the preservation of a large class of security properties, with the following important distinction. The refinement relations used for correctness are strong in that (via well-known results) refinement preserves all linear-time properties defined over atomic propositions common to both programs. Strong preservation is needed as the desired correctness properties may not be fully known or their specifications may not be available in practice. On the other hand, security properties are limited in nature and are likely to be well known in advance (e.g., "do not leak secret keys"). This motivates our construction of property-specific refinement relations. Being more focused, they are also easier to establish than refinements that preserve all properties in a class.

The refinement rules defined here implicitly require that a security specification apply equally well to the target and source programs. Thus, they are most applicable when the target and source languages and attack models are identical. This is the case, for instance, in the optimization phase of a compiler, where a number of transformations are applied to code that remains within the same intermediate representation. To complete the picture, it is necessary to look more generally at transformations that go from a higher-level language (say LLVM bytecode) to a lower-level one (say x86 machine code). The so-called "attack surfaces" are different for these two levels, so it would be necessary to also incorporate a back-translation of failures [9] in the refinement proof rules. How best to do so is an intriguing topic for future work.

Another question that we leave to future work is the completeness of the refinement rules. We have shown that a variety of common compiler transformations can be proved secure through logically simple refinement relations. The completeness question is whether *every* secure transformation has an associated stepwise refinement relation. In the case of correctness, this is a well-known theorem by Abadi and Lamport [2]. To the best of our knowledge, a corresponding theorem is not known for security hyperproperties.

There are a number of practical concerns that must be addressed to implement this methodology in a real compiler. One of these is a convenient notation for specifying desired security properties at the source level, for example as annotations to the source program. It is also necessary to define precisely how a security property is transformed by a program optimization. For instance, if a transformation introduces fresh variables, there needs to be a reasonable way to

determine whether those should be assigned a high or low security level for a non-interference property.

# References

- 1. Martín Abadi. Protection in programming-language translations. In Jan Vitek and Christian Damsgaard Jensen, editors, Secure Internet Programming, Security Issues for Mobile and Distributed Objects, volume 1603 of Lecture Notes in Computer Science, pages 19–34. Springer, 1999.
- Martín Abadi and Leslie Lamport. The existence of refinement mappings. In Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988, pages 165–175. IEEE Computer Society, 1988.
- 3. Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018, pages 328–343. IEEE Computer Society, 2018.
- 4. Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59:115–131, 1988.
- 5. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, Principles of Security and Trust Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, volume 8414 of Lecture Notes in Computer Science, pages 265–284. Springer, 2014.
- Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008, pages 51-65. IEEE Computer Society, 2008.
- 7. Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In Suresh Jagannathan and Peter Sewell, editors, The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014, pages 165–178. ACM, 2014.
- 8. Chaoqiang Deng and Kedar S. Namjoshi. Securing a compiler transformation. In Xavier Rival, editor, Static Analysis 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, volume 9837 of Lecture Notes in Computer Science, pages 170–188. Springer, 2016.
- 9. Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In Rastislav Bodík and Rupak Majumdar, editors, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 22, 2016, pages 164–177. ACM, 2016.
- Vijay D'Silva, Mathias Payer, and Dawn Xiaodong 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, pages 73-87. IEEE Computer Society, 2015.

- 11. Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009, pages 432-441. ACM, 2009.
- 12. Michael Howard. When scrubbing secrets in memory doesn't work, 2002. http://archive.cert.uni-stuttgart.de/bugtraq/2002/11/msg00046.html. Also https://cwe.mitre.org/data/definitions/14.html.
- Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, ACM SIG-PLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, pages 216-226. ACM, 2014.
- 14. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006, pages 42-54. ACM, 2006.
- 15. Zohar Manna and Amir Pnueli. Specification and verification of concurrent programs by forall-automata. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, Temporal Logic in Specification, Altrincham, UK, April 8-10, 1987, Proceedings, volume 398 of Lecture Notes in Computer Science, pages 124–164. Springer, 1987.
- Darko Marinov. Credible compilation. PhD thesis, Massachusetts Institute of Technology, 2000.
- 17. Toby C. Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018, pages 16–30. IEEE, 2018.
- 18. Kedar S. Namjoshi. A simple characterization of stuttering bisimulation. In S. Ramesh and G. Sivakumar, editors, Foundations of Software Technology and Theoretical Computer Science, 17th Conference, Kharagpur, India, December 18-20, 1997, Proceedings, volume 1346 of Lecture Notes in Computer Science, pages 284–296. Springer, 1997.
- 19. Kedar S Namjoshi. Witnessing an SSA transformation. In  $VeriSure\ Workshop$ , CAV, 2014.
- 20. Kedar S. Namjoshi and Lenore D. Zuck. Witnessing program transformations. In Francesco Logozzo and Manuel Fähndrich, editors, Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, volume 7935 of Lecture Notes in Computer Science, pages 304–323. Springer, 2013.
- 21. G. Necula. Translation validation of an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000*, pages 83–95, 2000.
- 22. Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation. *ACM Computing Surveys*, 2019. (to appear).
- Marco Patrignani and Deepak Garg. Secure compilation and hyperproperty preservation. In 30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017, pages 392-404. IEEE Computer Society, 2017.

- 24. A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (CVT)- automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.
- 25. Martin Rinard. Credible compilation. Technical report, In Proceedings of CC 2001: International Conference on Compiler Construction, 1999.
- 26. Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings, volume 3672 of Lecture Notes in Computer Science, pages 352–367. Springer, 2005.
- 27. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, pages 283–294. ACM, 2011.
- 28. Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead store elimination (still) considered harmful. In Engin Kirda and Thomas Ristenpart, editors, 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017., pages 1025–1040. USENIX Association, 2017.
- Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of ssa-based optimizations for LLVM. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, pages 175–186, 2013.
- 30. Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3):223–247, 2003.

# A Appendix

#### A.1 Proofs

**Lemma 2** Let S, T be programs over the same alphabet  $\Sigma_e$  and A be a Büchi automaton over  $\Sigma_e^k$ . Let I be a subset of  $\Sigma_e$ , and let R be a refinement relation from  $A \times T^k$  to  $A \times S^k$  for I. For every sequence v in  $\mathcal{L}(A \times T^k)$ , there is a sequence v in  $\mathcal{L}(A \times S^k)$  such that v and v are input-equivalent.

Proof. Let  $v = v_0, v_1 \dots$  be in  $\mathcal{L}(A \times T^k)$ . Let  $\rho^T = (q_0 = \iota^A, t_0 = \iota^{T^k}), v_0, (q_1, t_1), v_1, \dots$  be an accepting run on this sequence. By a simple induction from the refinement conditions, there is a sequence  $u = u_0, u_1, \dots$  and a run  $\rho^S = (p_0 = \iota^A, s_0 = \iota^S^k), u_0, (p_1, s_1), u_1, \dots$  of  $A \times S^k$  on u such that for each i, all of the following hold.

- 1.  $((q_i, t_i), (p_i, s_i)) \in R$ ;
- 2.  $u_i$  and  $v_i$  agree on I;
- 3. if  $q_i \in F$  then  $p_i \in F$ .

As the run  $\rho^T$  is accepting for  $A \times T^k$ , there are infinitely many points on the run satisfying F. By the third condition above, F holds infinitely often along  $\rho^S$  as well, so that run is accepting for  $A \times S^k$ . By the second condition,  $u_i$  and  $v_i$  agree on I for all i, so u and v are input-equivalent, as required.

**Theorem 6** Let S and T be programs over the same input alphabet I. Let  $\varphi$  be a general security property with automaton A representing its kernel  $\kappa$ . If there exists (1) a bisimulation B from T to S, and (2) a relation R from  $A \times T^k$  to  $A \times S^k$  that is a refinement relative to B, then T securely refines S for  $\varphi$ .

*Proof.* Let  $\xi^T$  be a winning strategy for the opponent in  $\mathcal{G}(T,\varphi)$ , and  $\xi^S = \mathcal{S}(\xi^T,B)$ . By construction, a play of  $\xi^S$  results in a bundle of k executions of  $S, y = (y_1,\ldots,y_k)$ , such that the bundle is pointwise input-equivalent to a bundle  $x = (x_1,\ldots,x_k)$  resulting from a play following  $\xi^T$ . Furthermore, by construction, x and y are pointwise related by B.

As  $\xi^T$  is a winning strategy for the opponent,  $(\operatorname{trace}(x_1), \ldots, \operatorname{trace}(x_k))$  does not satisfy  $\kappa$ . Therefore,  $w = \operatorname{zip}(\operatorname{trace}(x_1), \ldots, \operatorname{trace}(x_k))$  has an accepting run  $r^T$  on  $A \times T^k$  for which the program components are  $\operatorname{zip}(\operatorname{states}(x_1), \ldots, \operatorname{states}(x_k))$ .

By the definition of R, the initial states of  $A \times T^k$  and  $A \times S^k$  are related by R, and the program components of these states are related pointwise by B. These program components are also the initial states of  $\mathsf{zip}(x_1,\ldots,x_k)$  and  $\mathsf{zip}(y_1,\ldots,y_k)$ . By construction, the i-th entry on  $\mathsf{zip}(x_1,\ldots,x_k)$  is related by  $B^k$ to the i-th entry on  $\mathsf{zip}(y_1,\ldots,y_k)$ , for all positions i. By an inductive argument using the refinement relation, one can construct a corresponding run,  $r^S$ , of  $A \times S^k$  such that the program components are  $\mathsf{zip}(\mathsf{states}(x_1),\ldots,\mathsf{states}(x_k))$ . Moreover, for every final state in  $r^T$ , its corresponding state in  $r^S$  is also final. As  $r^T$  is accepting, so is  $r^S$ . Hence, the play y is a win for the opponent.

As this is true for arbitrary choices made by the player in  $\mathcal{G}(S,\varphi)$ , the strategy  $\xi^S$  is a winning strategy, as required.

#### A.2 Buffering Automata

The refinement relation described in Definition 4 makes use of a version of automaton-program product that assumes that every program and automaton transition has an observable label. Although this can be reasonable depending on the attack model (for example, when modeling constant-time security [3]), in many cases the program will include  $\varepsilon$ -transitions that produce no observation. This can lead to situations when two output traces become unsynchronized. At the end of Section 5 we briefly described a way to handle this issue by constructing a buffering automata. We now explain this construction in more detail.

As an example of when buffering automata are needed, consider the program below, and the property which says that the output trace is identical for any pair of inputs.

```
int x := read_input();
for(i := 0; true; i := i+1) {
   write_output(i);
   for(int t := x; t > 0; t := t-1); // delay for x steps
}
```

It is evident that this property holds, as every execution produces the output sequence  $0, 1, \ldots$  Yet, successive outputs are separated by  $|\mathbf{x}|$  steps, thus, outputs of executions with different input values are not synchronized, and the gap between corresponding outputs grows without bound.

To match this program behavior to the automaton for the negated trace property, we require converting the automaton A over  $\Sigma_e^k$  into an automaton  $\hat{A}$  over  $(\Sigma_e \cup \{\varepsilon\})^k$ , called a buffering automaton, with the following property:  $\hat{A}$  accepts  $\text{zip}(v_1, \ldots, v_k)$  iff A accepts  $\text{zip}(\text{compress}(v_1), \ldots, \text{compress}(v_k))$ . This produces an infinite-state automaton with buffers of unbounded size, that store observation histories of each trace. As the proof method is based on single-step refinement, an infinite-state automaton is not necessarily an obstacle to automated checking, so long as its structure can be represented symbolically in an SMT-supported theory. Furthermore, in many cases the refinement check can be performed without needing to reason about the structure of the automaton at all (see Section 6 for examples).

Formally, given an automaton  $A = (Q, \Sigma^k, \iota, \Delta, F)$ , we define the buffering automaton  $\hat{A} = (Q \times (\Sigma^*)^k \times \{0,1\}, (\Sigma \cup \{\varepsilon\})^k, (\iota, \varepsilon, \dots, \varepsilon, 0), \hat{\Delta}, F \times (\Sigma^*)^k \times \{1\})$ . Note that the state space of  $\hat{A}$  stores k finite sequences of observations, representing the prefix of the traces produced so far by each copy of the program. The alphabet of  $\hat{A}$  allows the copies to perform  $\varepsilon$ -transitions, representing the fact that each track might produce observations at different rates. The transition relation  $\hat{\Delta}$  is defined in the following way.  $((q, w, b), u, (q', w', b')) \in \hat{\Delta}$ , for  $q, q' \in Q$ ,  $w, w' \in (\Sigma^*)^k$ ,  $u \in (\Sigma \cup \{\varepsilon\})^k$  and  $b, b' \in \{0, 1\}$ , if:

```
1. q = q', w_i u_i = w_i' for all i \in \{1, ..., k\}, w_i' = \varepsilon for some i, and b' = 0; or 2. there is \sigma \in \Sigma^k such that w_i u = \sigma_i w_i' for all i \in \{1, ..., k\}, (q, \sigma, q') \in \Delta, and b' = 1.
```

Intuitively, whenever the k copies of the program perform a transition, producing observations  $u=(u_1,\ldots,u_k)$ ,  $\hat{A}$  concatenates  $u_i$  with the sequence  $w_i$  stored as part of the state. Then, if any of the resulting sequences is empty, the state q of the automaton does not advance. Otherwise, if all of the sequences are non-empty, the automaton reads the first position of all sequences, removing them from the buffer, and transitions to a new state q' accordingly. In this way,  $\hat{A}$  visits the same states that A would visit if all of the tracks were synchronized, possibly with delays due to having to wait until all tracks have produced the next observation.

The  $\{0,1\}$  component of the state is needed to indicate whether the automaton makes progress. It remains as 0 while the automaton is still waiting for an observation from one of the tracks, and is set to 1 whenever the automaton takes a transition to a different Q state.  $\hat{A}$  accepts iff it makes progress into an accepting Q state infinitely often.

**Lemma 3.**  $\hat{A}$  accepts  $zip(v_1, \ldots, v_k)$  iff A accepts  $zip(compress(v_1), \ldots, compress(v_k))$ .

Proof. Given an accepting run  $\hat{r}$  of  $\hat{A}$  on  $\operatorname{zip}(v_1,\ldots,v_k)$ , a corresponding accepting run r of A on  $\operatorname{zip}(\operatorname{compress}(v_1),\ldots,\operatorname{compress}(v_k))$  can be constructed in the following way. First, note that there are two kinds of transitions in  $\hat{r}$ , each corresponding to one of the two cases in the definition of  $\hat{\Delta}$ . Due to our assumption that no program has a trace that after a point only produces  $\varepsilon$ , transitions of the second kind must occur infinitely often in  $\hat{r}$ . For every such transition ((q, w, b), u, (q', w', b')) there is a corresponding transition  $(q, \sigma, q') \in \Delta$ , and between two such transitions q does not change. Therefore, the corresponding transitions in  $\Delta$  can be put end-to-end to form a run r on A. By construction of  $\hat{\Delta}$ , the trace of r is precisely  $\operatorname{zip}(\operatorname{compress}(v_1),\ldots,\operatorname{compress}(v_k))$ . Furthermore, note that accepting states (q',w',1) of  $\hat{A}$  can only be reached by transitions of the second kind, and in every such state q' is an accepting state of A. Therefore, every occurrence of an accepting state (q',w',1) in  $\hat{r}$  has a corresponding occurrence of q' in r. Since such occurrences happen infinitely often, r is an accepting run.

To prove the opposite direction, assume that r is an accepting run of A on  $\mathsf{zip}(\mathsf{compress}(v_1),\ldots,\mathsf{compress}(v_k))$ . Then, a corresponding accepting run  $\hat{r}$  of  $\hat{A}$  on  $\mathsf{zip}(v_1,\ldots,v_k)$  can be constructed in the following way. Note that the only source of nondeterminism in  $\hat{\Delta}$  is in the choice of  $(q,\sigma,q') \in \Delta$  for the second kind of transition. If this choice is always made by choosing the next transition in r, then an accepting state (q',w',1) will be visited infinitely often. Therefore,  $\hat{r}$  will be accepting. The construction of  $\hat{\Delta}$  guarantees that  $\hat{r}$  constructed in this way will be a valid run of  $\mathsf{zip}(v_1,\ldots,v_k)$  in  $\hat{A}$ .

Since the alphabet of  $\hat{A}$  is  $(\Sigma \cup \{\varepsilon\})^k$ , we can take its product with the k-fold composition of a program with  $\varepsilon$  transitions.

**Lemma 4.** A bundle of executions  $\sigma = (\sigma_1, \dots, \sigma_k)$  produced by program P does not satisfy  $\kappa$  iff  $\mathsf{zip}(\mathsf{trace}(\sigma_1), \dots, \mathsf{trace}(\sigma_k))$  is accepted by  $\hat{A} \times P^k$ .

*Proof.* Bundle  $\sigma$  does not satisfy  $\kappa$  iff (by definition)  $\mathsf{zip}(\mathsf{ctrace}(\sigma_1), \dots, \mathsf{ctrace}(\sigma_k))$ is accepted by A, iff (by Lemma 3)  $zip(trace(\sigma_1), ..., trace(\sigma_k))$  is accepted by  $\hat{A}$ , iff (as  $\mathsf{zip}(\mathsf{trace}(\sigma_1), \dots, \mathsf{trace}(\sigma_k))$  is the trace of execution  $\mathsf{zip}(\sigma_1, \dots, \sigma_k)$  of  $P^k$  and Lemma 1)  $\mathsf{zip}(\mathsf{trace}(\sigma_1), \ldots, \mathsf{trace}(\sigma_k))$  is accepted by  $\hat{A} \times P^k$ .

### Additional Examples of Refinement Relations

Example 3: Switching Instructions (Special Cases) As mentioned in Section 6, setting up a refinement relation for the switching-instructions transformation becomes harder if the instructions being switched produce observations, due to the fact that  $(q^S = q^T)$  might not hold at the location of the switch. There are some special cases, however, where this may be done more easily:

- 1. If the different copies of S and T are guaranteed to be synchronized (i.e., it is always the case that  $\ell_i^S = \ell_j^S$  and  $\ell_i^T = \ell_j^T$ , for all i and j).

  2. If only one of the switched instructions produces an observation.

The advantage of the first case is that if all k copies of the program produce corresponding observations at the same time, the automaton can compare them immediately and does not need to store them in its internal state. Then, it is likely that for many relevant security properties the state of the automaton does not change at all when executing the switched instructions, and so the relationship  $(q^S=q^T)$  still holds. In this case the constraint that  $\ell^S_i=\ell^S_j$  and  $\ell_i^T = \ell_i^T$ , for all i and j, must be added to R. Additionally, R will need to include enough information for the inductive check to confirm that the state of the automaton indeed does not change while observing the switched instructions. What this information is will depend on the property under verification.

The second case can be handled in a couple of different ways. One option, for example, is to relax the constraint  $(\ell_i^S = \ell_i^T)$  to instead relate the instruction that produces an observation in the source with the same instruction that produces an observation in the target (say, L5 in the source with L6 in the target). Then, stuttering can be used in both the source and the target to align the executions (for example, the target stutters in L5, and the source stutters in L6). This will also require changing the constraints  $(\ell_i^S \neq \ell_{switch}^S \rightarrow \alpha_i^S = \alpha_i^T) \land (\ell_i^S = \ell_{switch}^S \rightarrow \delta(\alpha_i^S, \alpha_i^T))$  keeping track of the changes in state appropriately. The modified refinement seeks to align the observations rather than the instructions. An easier alternative, instead, might be to model S and T in such a way that the two instructions are considered as a block, comprising a single transition in the transition systems. Then, since the two blocks modify the state of the program in the same way and produce the same observation, the refinement relation becomes trivial.

Example 4: Dead-Branch Elimination This optimization replaces an if statement whose guard is trivially false by only the contents of the else branch. For simplicity, assume there is always an else branch (if statements with no else branch can be modeled by leaving the else branch empty). Consider the following example programs:

The refinement relation for this source-target pair can be written as:

$$(q^S = q^T) \wedge \bigwedge_{i=1}^k (\alpha_i^S = \alpha_i^T) \wedge L(\ell_i^S, \ell_i^T)$$

where  $L = \{(L1, L1), (L2, L2), (L5, L2), (End, End)\}$  denotes which locations can be related between the source and the target. Note that since L2 in the target is related to both L2 and L5 in the source, it is necessary to allow stuttering when checking the refinement relation.

This same refinement relation can be used for arbitrary programs, as long as L is redefined appropriately. Let  $\ell^S_{if}$  be the program location of the if guard (L2 in the example), and  $\ell^S_{else}$  be the program location at the start of the else branch (L5 in the example). Let  $\ell^T_{else}$  be the location in the target corresponding to  $\ell^S_{else}$  (L2 in the example). L is then defined to be a binary relation relating program locations between the source and target in the following way: corresponding locations are related by L, and  $\ell^T_{else}$ , in addition to being related to  $\ell^S_{else}$ , is also related to  $\ell^S_{if}$ .

Note that this relation also defines a relation  $B((\alpha_i^S, \ell_i^S), (\alpha_i^T, \ell_i^T)) = (\alpha_i^S = \alpha_i^T) \wedge L(\ell_i^S, \ell_i^T)$  between single program states of S and T such that B is a (stuttering) bisimulation. Therefore, this refinement relation can also be used for security properties with arbitrary quantifier alternation.

**Example 5: Expression Flattening** This optimization "flattens" a nested expression. It can be thought of as turning an expression into three-address code. For example:

```
// Source program S
L1: int x := (f(a) + b) * (g(c) / d);
L1: int t0 := f(a);
L2: int t1 := t0 + b;
L3: int t2 := g(c);
L4: int t3 := t2 / d;
L5: int x := t1 * t3;
```

Note that many programming languages leave the evaluation order of subexpressions undefined. Different orders of evaluation may generate different result states, e.g., if subexpressions such as f(a), g(c) have side-effects. This transformation picks a certain evaluation order. Thus, while in a detailed program

semantics the original expression has an undetermined evaluation order, a specific order is fixed in the target program. This is an example of a transformation that removes non-determinism from the program.

The original and flattened evaluations are equivalent when the detailed evaluation is treated as a block, ignoring the values of intermediate variables. Thus, the correctness argument establishes that, starting from states related by  $\alpha_i^S(v) = \alpha_i^T(v)$  for all  $v \in \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\},$  the ordered execution in the target T matches the result of one of the possible non-deterministic evaluation choices in the source S, resulting in states that are also identically related, in this case  $\alpha_i^S(v) = \alpha_i^T(v)$  for all  $v \in \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{x}\}.$ 

For any automaton, we thus define the relation R as:

$$(q^S = q^T) \wedge \bigwedge_{i=1}^k (\alpha_i^S =_V \alpha_i^T)$$

where  $V = \{a, b, c, d, x\}$  denotes all non-temporary variables (i.e., excluding variables introduced by the transformation), and  $(\alpha_i^S =_V \alpha_i^T)$  denotes the pointwise equality of  $\alpha_i^S(v)$  and  $\alpha_i^T(v)$  for all  $v \in V$ .

It is easy to show that this relation is a refinement relation if the transition system representing T is modeled in such a way that the transformed segments are treated as a block of instructions defining a single transition. In this case the refinement check requires only a single inductive step, following the reasoning described above. If instead the instructions in the target are treated as individual transitions rather than a block, it is necessary to add constraints to remember the value of the intermediate variables in the target, for example  $(\ell_i^T \in \{L3, L4, L5\} \to \alpha_i^T(\mathsf{t1}) = \alpha_i^T(\mathsf{t0}) + \alpha_i^T(\mathsf{b}))$ . With this addition, the relation can then be used as a stuttering refinement, where the stuttering is bounded by the length of the expanded block.

Note that the correctness of the refinement relies on the assumption that there is no change in observations between the original single-instruction block and the new multiple-instruction block. That may not be the case depending on the attack model. For instance, if  $\mathbf{x} := (\texttt{secret} + 2) - \texttt{secret}$  is compiled to t1 := secret + 2;  $\mathbf{x} := \texttt{t1} - \texttt{secret}$  then the secret value is leaked in the value of t1, even though there is no leakage in the original assignment to  $\mathbf{x}$  in the source. Whether this matters depends on the attack model. If the attack model allows values of temporary variables to be observable, then the transformation is not secure.

Unlike previous examples, the correspondence for single traces in expression flattening is not in general a bisimulation, since the elimination of non-determinism means that there might be a transition in the source with no corresponding transition in the target. Therefore, the security preservation result applies only to universally-quantified properties.

**Example 6: Loop Peeling** This optimization peels off the first iteration of a loop, as long as the compiler can guarantee that the loop guard will be true on

entry. For simplicity, assume only *while* loops. Consider the following example programs:

```
// Source program S
                                   // Target program T
L1: int x := 0;
                                   L1: int x := 0;
L2: int k := 0;
                                   L2: int k := 0;
L3: while (k < 8) {
                                   L3: if (k == 0) {
L4:
        if (k == 0) {
                                   L4:
                                            x := secret_input();
L5:
            x := secret_input(); L5: } else {
L6:
        } else {
                                   L6:
                                           x := x + x;
                                   L7: }
L7:
            x := x + x;
                                   L8: k := k + 1;
L8:
L9:
                                   L9: public_output(x % k);
        k := k + 1;
L10:
        public_output(x % k);
                                   L10: while (k < 8) {
L11: }
                                   L11:
                                            if (k == 0) {
                                   L12:
                                                x := secret_input();
                                   L13:
                                            } else {
                                   L14:
                                                x := x + x;
                                   L15:
                                   L16:
                                            k := k + 1;
                                   L17:
                                            public_output(x % k);
                                   L18: }
```

Since the two programs produce the same traces, the following refinement relation can be used independently of the security property:

$$L = \{(L1, L1), (L2, L2), (L3, L3), (L4, L3), (L5, L4), (L7, L6), (L9, L8), (L10, L9), (L3, L10), (L4, L11), (L5, L12), (L7, L14), (L9, L16), (L10, L17), (End, End)\}$$

$$(q^S = q^T) \wedge \bigwedge_{i=1}^k (\alpha_i^S = \alpha_i^T) \wedge L(\ell_i^S, \ell_i^T) \wedge (\ell_i^T = L3 \to \alpha_i^S(k) < 8)$$

This refinement relation requires stuttering, as the target stutters in L3 while the source moves from L3 and L4. For general programs, L should be defined as follows:

- 1. Corresponding locations are related by L.
- 2. Every instruction in the peeled loop iteration in the target is related by L to the corresponding instruction within the loop body in the source.
- 3.  $\ell^T_{peel}$ , the location at the start of the peeled loop iteration in the target, is additionally related by L to  $\ell^S_{guard}$ , the location of the loop guard in the source. In the example, both  $\ell^T_{peel}$  and  $\ell^S_{guard}$  are L3.

The formula  $\varphi_0 = (\ell_i^T = L3 \to \alpha_i^S(k) < 8)$  in the example serves to state that the loop will be entered. It must be added to the refinement relation in

order to justify the execution of the peeled iteration in the target. For arbitrary programs, the general form of  $\varphi_0$  is  $(\ell_i^T = \ell_{peel}^T \to \psi(\alpha_i^S))$ , where  $\psi$  is the loop guard at location  $\ell_{guard}^S$ . In the example, adding just  $\varphi_0$  suffices, since it can be proved inductively from the previous instruction int  $\mathbf{k} := 0$ . However, in more complex cases other formulas  $\varphi_1, \ldots, \varphi_n$  of the same form might need to be added to describe the compiler's reasoning leading up to the conclusion that  $\varphi_0$  holds. For example, suppose that int  $\mathbf{k} := 0$  was replaced by int  $\mathbf{k} := \mathbf{x}$ . Then it would be necessary to add a formula such as  $\varphi_1 = (\ell_i^T = L2 \to \alpha_i^S(x) < 8)$  in order for the relation to pass the refinement check. Therefore, the general form of the refinement relation for an arbitrary program is:

$$(q^S = q^T) \wedge \bigwedge_{i=1}^k (\alpha_i^S = \alpha_i^T) \wedge L(\ell_i^S, \ell_i^T) \wedge \varphi_0 \wedge \varphi_1 \wedge \ldots \wedge \varphi_n$$

For security properties with arbitrary quantifier alternation, the bisimulation can simply be defined as  $B((\alpha_i^S, \ell_i^S), (\alpha_i^T, \ell_i^T)) = (\alpha_i^S = \alpha_i^T) \wedge L(\ell_i^S, \ell_i^T) \wedge \varphi_0 \wedge \varphi_1 \wedge \ldots \wedge \varphi_n$ .

**Example 7: Register Spilling** This transformation "spills" the contents of a register to memory, if the register has to be free for a subsequent operation. For example, consider a machine with only two registers, A and B.

```
// Source program S
L1: int t0 := a + b;
L2: public_output(t0);
L3: int t1 := a - b;
L3: int t1 := a - b;
L3: public_output(A);
L4: A := spill[0];
L5: A := A - B;
// B = b, A = t1, spill[0] = a
```

There is a simple correspondence between the source variables and the target registers and spill array entries, inferred by the register allocation and spilling method, and shown in the code comments. Note that this correspondence varies depending on the program location. Therefore, it can be expressed by a mapping  $\sigma: (\text{Reg} \cup \text{Spill}) \times \text{Loc} \to \mathcal{V}$  from a register (or position in the spill array) and a program location in the target to a corresponding variable in the source.

The two programs can thus be related by the following stuttering refinement relation R (note that the stuttering can be avoided if we allow transitions composed of blocks of instructions):

$$L = \{(L1, L1), (L1, L2), (L2, L3), (L3, L4), (L3, L5), (End, End)\}$$

$$(q^S = q^T) \wedge \bigwedge_{i=1}^k \left( L(\ell_i^S, \ell_i^T) \wedge \bigwedge_{v \in (\text{Reg} \cup \text{Spill})} (\alpha_i^S(\sigma(v, \ell_i^T)) = \alpha_i^T(v)) \right)$$

The refinement ensures that at every point of the program, every register and position in the spill array contains the same value stored in the corresponding variable of the source given by  $\sigma$ . The single-trace correspondence  $(L(\ell_i^S, \ell_i^T) \land \bigwedge_{v \in (\text{Reg} \cup \text{Sp}_{\text{ILL}})} (\alpha_i^S(\sigma(v, \ell_i^T)) = \alpha_i^T(v))$  is a stuttering bisimulation, therefore this refinement can also be used with properties with arbitrary quantifier alternation.

Note that we are making the assumption that the transformation preserves observations. This might not be the case if the attack model includes, for example, the address of memory positions accessed by the program, as the target program introduces a new array for spilling. In this case security might be preserved or not, depending on the security property under verification, and a specialized refinement relation would be necessary.