Witnessing Secure Compilation

Kedar S. Namjoshi[1(✉)] and Lucas M. Tabajara[2]

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

Abstract. Compiler optimizations may break or weaken the security properties of a source program. This work develops a translation validation methodology for secure compilation. A security property is expressed as an automaton operating over a bundle of program traces. A refinement proof scheme derived from a property automaton guarantees that the associated security property is preserved by a program transformation. This generalizes known refinement methods that apply only to specific security properties. In practice, the refinement relations ("security witnesses") are generated during compilation and validated independently with a refinement checker. This process is illustrated for common optimizations. Crucially, it is not necessary to formally verify the compiler implementation, which is infeasible for production compilers.

1 Introduction

Optimizing compilers are used to improve the run time performance of software programs. An optimization is correct if it preserves input-output behavior. A number of approaches, including automated testing (cf. [13,28]), translation validation (cf. [22,25,31]), and full mathematical proof (cf. [14]) have been developed to gain confidence in the correctness of compilation.

Correctness does not, however, guarantee the preservation of security properties. It is known that common optimizations may weaken or break security properties that hold of a source program (cf. [10,12]). A *secure compiler* is one that, in addition to being correct, also preserves security properties. This work provides a methodology for formally establishing secure compilation.

```
int x := read_secret_key();          int x := read_secret_key();
use(x);                              use(x);
x := 0; // clear secret data         skip; // dead store removed
rest_of_program();                   rest_of_program();
```

Fig. 1. Information leakage through optimization. Source program on left, optimized program on right.

© Springer Nature Switzerland AG 2020
D. Beyer and D. Zufferey (Eds.): VMCAI 2020, LNCS 11990, pp. 1–22, 2020.
https://doi.org/10.1007/978-3-030-39322-9_1

Figure 1 shows an instance of the *dead store removal* optimization. This optimization eliminates stores (i.e., assignment statements) that have no effect on the input-output behavior of the source program. If variable `x` is not referenced in `rest_of_program`, the optimization correctly replaces `x := 0` with `skip`. The replacement, however, exposes the secret key stored in `x` to the rest of the program, which may be vulnerable to an attack that leaks this secret, thus breaking a vital security property of the source program.

Compiler directives can be used to prevent this optimization from taking effect. Such fixes are unsatisfactory and brittle, however, as they assume that programmers are aware of the potential security issue and understand enough of a compiler's workings to choose and correctly place the directives. Moreover, the directives may not be portable across compilers [29].

It is far more robust to build security preservation into a compiler. The classical approach constructs a mathematical proof of secure compilation, applicable to all source programs. This is highly challenging for at least two reasons. The first is that of proof complexity. Past experience shows that such proofs can take man-years of effort, even for compact, formally designed compilers such as CompCert [4,14]. Constructing such proofs is entirely infeasible for production compilers such as GCC or LLVM, which have millions of lines of code written in hard-to-formalize languages such as `C` and `C++`. The second reason is that, unlike correctness, secure compilation is not defined by a single property: each source program may have its own notion of security. Even standard properties such as non-interference and constant-time have subtle variants.

This work addresses both issues. To tackle the issue of proof complexity, we turn to *Translation Validation* [25] (TV), where correctness is established at compile time only for the program being compiled. We use a form of TV that we call "witnessing" [21,26], where a compiler is *designed* to generate a proof (also called a "certificate" or a "witness") of property preservation. For correctness properties, this proof takes the form of a refinement relation relating single traces of source and target programs. For security preservation, it is necessary to have refinement relations that relate "bundles" of k traces ($k \geq 1$) from the source and target programs.

To address the second issue, we show how to construct property-specific refinement proof rules. A security property is defined as an automaton operating on trace bundles, a flexible formulation that encompasses standard security properties such as non-interference and constant-time. The shape of the induced refinement proof rule follows the structure of the property automaton.

Refinement rules are known for the important security properties of non-interference [3,8,17] and constant-time execution [4]. We show that these rules arise easily and directly from an automaton-based formulation. As automata can express a large class of security properties, including those in the HyperLTL logic [6], the ability to derive refinement proof rules from automata considerably expands the reach of the refinement method.

We now discuss these contributions in more detail. We use a logic akin to HyperLTL [6] to describe security hyperproperties [7,27], which are sets of sets

of sequences. A security property φ is represented by a formula of the shape $Q_1\pi_1, \ldots, Q_k\pi_k : \kappa(\pi_1, \ldots, \pi_k)$, where the π_i's represent traces over an observation alphabet, the Q_i's stand for either existential or universal quantification, and κ is a set of bundles of k program traces, represented by a Büchi automaton A_κ whose language is the *complement* of κ. The structure of this automaton is reflected in the derived refinement proof rule for φ.

A transformation from program S to program T preserves a security property φ if every violation of φ by T has a matching violation of φ by S. Intuitively, matching violations have the same inputs and are of the same type.

The first refinement scheme applies to purely universal properties, those of the form $\forall\pi_1 \ldots \forall\pi_k : \kappa(\pi_1, \ldots, \pi_k)$. The witness is a refinement relation between the product transition systems $A_\kappa \times T^k$ and $A_\kappa \times S^k$. The second refinement scheme applies to arbitrary properties ($\forall\exists$ alternation is used to express limits on an attacker's knowledge). Here, the witness is a *pair* of relations: one being a refinement relation between $A_\kappa \times T^k$ and $A_\kappa \times S^k$, as before; the second component is an input-preserving bisimulation relation between T and S.

We define refinement relations for several common compiler optimizations. Those relations are logically simple, ensuring that their validity can be checked automatically with SMT solvers. Crucially, the witnessing methodology does not require one to verify either the compiler implementation or the proof generator, considerably reducing the size of the trusted code base and making the methodology applicable to production compilers.

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);
L5:
```

In this program, `x` stores the value of a secret input. As will be described in Sect. 3.1, this program can be modeled as a transition system. The states of the system can be considered to be pairs (α, ℓ). The first component $\alpha : \mathcal{V} \rightarrow \textsc{Int}$ is a partial assignment mapping variables in $\mathcal{V} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ to values in $\textsc{Int}$, the set of values that a variable of type `int` can contain. The second component $\ell \in \textsc{Loc} = \{\mathtt{L1}, \mathtt{L2}, \mathtt{L3}, \mathtt{L4}, \mathtt{L5}\}$ is a location in the program, indicating the next instruction to be executed. In the initial state, α is empty and ℓ points to location `L1`. Transitions of the system update α according to the variable assignment instructions, and ℓ 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 (Sect. 3.2) and a security property over a set of program executions (Sect. 4). Suppose that an attacker can see the state of the memory at the end

of the program, represented by the final value of α, 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 π_1 and π_2 (cf. Section 4). The negation of κ 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* (Sect. 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;
L5:
```

By taking the product of the automaton A with two copies of S or T (one for each trace π_i considered by κ), 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 Sect. 6 for details). The inductiveness of this relation over transitions of $A \times T^2$ and $A \times S^2$ can be easily checked with an SMT solver by using symbolic representations.

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 Σ be an *alphabet*, i.e., a set of symbols, and let Γ be a subset of Σ. An infinite sequence $u = u(0), u(1), \ldots$, where $u(i) \in \Sigma$ for all i, is said to be a "sequence over Σ". For variables x, y denoting elements of Σ, the notation $x =_\Gamma y$ (read as "x and y agree on Γ") denotes the predicate where either x and y are both not in Γ, or x and y are both in Γ and $x = y$. For a sequence u over Σ, the notation $u|_\Gamma$ (read as "u projected to Γ") denotes the sub-sequence of u formed by elements in Γ. The operator $\mathsf{compress}(v) = v|_\Sigma$, applied to a sequence v over $\Sigma \cup \{\varepsilon\}$, removes all ε symbols in v to form a sequence over Σ. For a bundle of traces $w = (w_1, \ldots, w_k)$ where each trace is an infinite sequence of Σ, the operator $\mathsf{zip}(w)$ defines an infinite sequence over Σ^k obtained by choosing successive elements from each trace. In other words, $u = \mathsf{zip}(w)$ 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;
- Σ 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 ε represent internal transitions where the state of the program changes without any observable effect.

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

An execution $x = (c_0, w_0, c_1)(c_1, w_1, c_2) \ldots$ has an associated *trace*, denoted $\mathsf{trace}(x)$, given by the sequence $w_0, w_1, \ldots$ over $\Sigma \cup \{\varepsilon\}$. The compressed trace of execution x, $\mathsf{compress}(\mathsf{trace}(x))$, is denoted $\mathsf{ctrace}(x)$. 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 $\mathsf{states}(x)$.

3.2 Attack Models as Extended Transition Systems

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 [4]. 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, Σ_e, which is the union of Σ with O_e. The extensions do not alter input-output behavior; formally, the original and extended systems are bisimular with respect to Σ.

3.3 Büchi Automata over Trace Bundles

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

- Q is the state space of the automaton;
- Σ_e^k is the alphabet of the automaton, each element is a k-vector over Σ_e;
- $\iota \in Q$ is the initial state;
- $\Delta \subseteq Q \times \Sigma_e^k \times Q$ is the transition relation;
- $F \subseteq Q$ is the set of accepting states.

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, $a_i = (t_1(i), \ldots, t_k(i))$—that is, $a_0, a_1, \ldots$ equals $\mathsf{zip}(t)$—and (q_i, a_i, q_{i+1}) is in the transition relation Δ. 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 ε-transitions. Programs with ε-transitions can be handled by converting A over Σ_e^k into a new automaton $\hat{A}$ over $(\Sigma_e \cup \{\varepsilon\})^k$ (see full version [20] 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 $P = (C, \Sigma_e, \iota, \rightarrow)$ 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:

- $Q^B = Q^A \times C^k$;
- $\iota^B = (\iota^A, (\iota, \ldots, \iota))$;

– $((q, s), u, (q', s'))$ is in Δ^B if, and only if, (q, u, q') is in Δ^A, and (s_i, u_i, s'_i) is in $(\rightarrow)$ for all i;
– (q, s) is in F^B iff q is in F^A.

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

Bisimulations. For programs $S = (C^S, \Sigma_e, \iota^S, \rightarrow^S)$ and $T = (C^T, \Sigma_e, \iota^T, \rightarrow^T)$, and a subset I of Σ_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 $(\rightarrow^T)$ there is u and s' such that (s, u, s') is in $(\rightarrow^S)$ and $(t', s') \in B$ and $u =_I v$.
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 $u =_I v$.

4 Formulating Security Preservation

A temporal correctness property 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* [7,27], 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 [6].

A security property φ 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 κ is set of k-trace bundles, described by a Büchi automaton whose language is the *complement* of κ. 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 φ 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 κ and a win for Bob otherwise.

A *strategy* for Bob is a function, say ξ, 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 σ_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 φ, written $\models_P \varphi$, if the protagonist has a winning strategy in the game $\mathcal{G}(P, \varphi)$.*

4.1 Secure Program Transformation

Let $S = (C^S, \Sigma_e, \iota^S, \rightarrow^S)$ be the transition system representing the original *source* program and let $T = (C^T, \Sigma_e, \iota^T, \rightarrow^T)$ be the transition system for the transformed *target* program. Any notion of secure transformation must imply the preservation property that if S satisfies φ and the transformation from S to T is secure for φ then T also satisfies φ.

Preservation in itself is, however, too weak to serve as a definition of secure transformation. Consider the transformation shown in Fig. 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 ξ^S for the antagonist in $\mathcal{G}(S, \varphi)$ (representing a violation in S)* matches *a strategy ξ^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 ξ^T, there is a maximal play $v = v_1, \ldots, v_k$ following ξ^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 φ 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 φ and if S satisfies φ, then T satisfies φ.*

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 κ.

In the rest of the paper, we consider φ to be specified by a single automaton rather than a collection, to avoid notational clutter.

5 Refinement for Preservation of Universal Properties

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

Definition 4. *Let S, T be programs over the same alphabet, Σ_e, and A be a Büchi automaton over Σ_e^k. Let I be a subset of Σ_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 $\varphi = (\forall\pi_1, \ldots, \pi_k : \kappa(\pi_1, \ldots, \pi_k))$ 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 κ; 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 φ.*

Proof. A violation of φ by T is given by a bundle of executions of T that violates κ. We show that there is an input-equivalent bundle of executions of S that also violates κ. Let $x = (x_1, \ldots, x_k)$ be a bundle of executions of T that does not satisfy κ. By Lemma 1, $v = \mathsf{zip}(\mathsf{trace}(x_1), \ldots, \mathsf{trace}(x_k))$ 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 $u = \mathsf{zip}(\mathsf{trace}(y_1), \ldots, \mathsf{trace}(y_k))$ and y violates κ. As u and v are input equivalent, $\mathsf{trace}(x_i)$ and $\mathsf{trace}(y_i)$ are input-equivalent for all i, as required. □

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 Sect. 6, which also discusses SMT-based checking of the proof requirements.

To handle programs that include ε-transitions, we can convert the automaton A over Σ_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 full version [20]. 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 [5] and one-step formulations suitable for SMT solvers are known (cf. [14,18]).

6 Checking Transformation Security

In this section, we formulate the general construction of an SMT formula for the correctness of a given refinement relation. We then show how to express a refinement relation for several common compiler optimizations.

6.1 Refinement Checking with SMT Solvers

Assume that the refinement relation R, the transition relations Δ, $(\rightarrow_T)$ 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 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. The proof of the inductive step requires establishing that R is closed under automaton transitions. This can be expressed by an SMT query of the shape $(\forall q^T, q^S, p^T, t, s, t', \sigma^T : (\exists \sigma^S, p^S, s' : \varphi_1 \rightarrow \varphi_2))$, where:

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

$$\begin{aligned}\varphi_2 \equiv\ & \Delta(q^S, \sigma^S, p^S) \wedge \bigwedge_{i=1}^{k} (s_i \xrightarrow{\sigma_i^S}_S s_i') \wedge \bigwedge_{i=1}^{k} (\sigma_i^T =_I \sigma_i^S) \\ & \wedge R((p^T, t'), (p^S, s')) \wedge (F(p^T) \rightarrow F(p^S))\end{aligned}$$

This formula has a quantifier alternation, which is difficult for SMT solvers to handle. It can be reduced to a quantifier-free form by providing Skolem functions from the universal to the existential variables. We expect the compiler to generate these functions as part of the witness generation mechanism.

As we will see in the examples below, in many cases the compiler can choose Skolem functions that are simple enough so that the validity of the formula can be verified using only equality reasoning, making it unnecessary to even expand the definitions of Δ and F. The general expectation is that a compiler writer must have a proof in mind for each optimization and should therefore be able to provide the Skolem functions necessary to establish refinement.

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 full version [20]. All transformations are based on the examples in [4].

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 (α, ℓ), where $\alpha : \mathcal{V} \to \textsc{Int}$ and $\ell \in \textsc{Loc}$. 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 $t_i = (\alpha_i^T, \ell_i^T)$:

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

The final terms express known constant values, necessary to establish inductiveness. In general, if the transformation relies on the fact that at location ℓ 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. Since this is a simple transformation, equality between states is all that is needed to establish a refinement.

R can be checked using the SMT query described in Sect. 6.1. For this 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. This applies to several other optimizations as well, as the 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
L1: if (j < arr_size) {
L2:   a := arr[0];
L3:   b := arr[j];
L4: } else {
L5:   a := arr[0];
L6:   b := arr[arr_size-1];
L7: }
```

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

Suppose that the attack model allows the attacker to observe memory accesses, represented by the index `j` of every array access `arr[j]`. We assume that other variables are stored in registers rather than memory (see full version [20] 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 is secure regardless of the security property φ. However, because the order of instructions is different, a more complex refinement relation R is needed, compared to constant folding:

$$((t = s) \land (q^T = q^S)) \lor \bigwedge_{i=1}^{k} \Big((\ell_i^T = \texttt{L2}) \land ((\alpha_i^S(\texttt{j}) < \alpha_i^S(\texttt{arr_size}))\,?\,(\ell_i^S = \texttt{L2}) : (\ell_i^S = \texttt{L5})) \land (\alpha_i^T = \alpha_i^S[\texttt{a} := \texttt{arr[0]}]) \land \Delta(q^S, (0, \ldots, 0), q^T) \Big)$$

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 has not. At that point, T is at location $L2$ and S is either at location $L2$ or $L5$, depending on how the guard was evaluated. 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 `a := arr[0]` on every track of S the two states are identical. (The notation $\alpha[x := e]$ denotes the state α' that is identical to α except at x, where its value is given by $\alpha(e)$.) As 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
L1: int a[10], b[10], j;
L2: a[0] := secret_input();
L3: b[0] := secret_input();
L4: for (j:=1; j<10; j++) {
L5:   a[j] := b[j-1];
L6:   b[j] := a[j-1];
L7:   public_output(j);
L8: }
```

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

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

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

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 ℓ_{switch}^S where the switch happens and $\alpha_i^S[\texttt{b[j]} := \texttt{a[j-1]}] = \alpha_i^T[\texttt{a[j]} := \texttt{b[j-1]}]$ by an appropriate formula $\delta(\alpha_i^S, \alpha_i^T)$ 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 ℓ_{switch}^S, 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. In general, reordering transformations require the addition of history variables to set up an inductive refinement relation. Details can be found in the full version [20].

7 Connections to Existing Proof Rules

We establish connections to known proof rules for preservation of the non-interference [3,8,17] and constant-time [4] 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 [4] 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 [4] 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 Fig. 2, which simply looks for a sequence starting with an initial pair of input values satisfying ϕ and ending in unequal transition labels. The states are I (initial), S (sink), M (mid), and F (fail), which is also the accepting state.

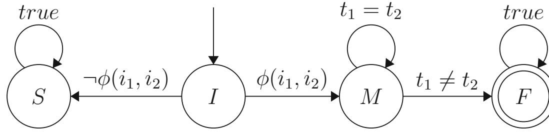

Fig. 2. A Büchi automaton for the negation of the constant-time property.

We now define the automaton-based relation, using the notation in Theorem 1 of [4]. 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 $\alpha = \hat{C}(i)$, $\alpha' = \hat{C}(i')$, $a = \hat{S}(i)$, $a = \hat{S}(i')$, 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 β, β' 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 the successor 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 [4]].

Finally, consider the interesting case where $p = q = M$. Let τ, τ' 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 q'.
(2) Otherwise, $t = t'$ and $p' = M$. By CT-simulation [Definition 6 of [4]: $a \equiv_S a'$ 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 [3,8,17]. 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 Fig. 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 $i \neq j$.)

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.

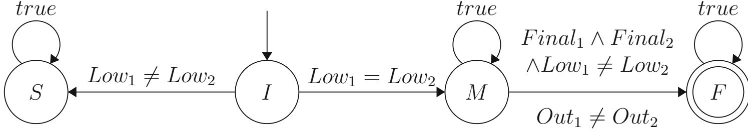

Fig. 3. A Büchi automaton for the negation of the non-interference property.

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, or (2) the outputs of the transitions from t_0, t_1 to t_0', t_1' differ. In case (1), by condition (a), s_0, s_1 are also final states, and therefore by condition (b) the values of the low variables differ in s_0, s_1 as well, so p' is also F. In case (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, since $p = M$ and M has a self-loop on *true*, then p' can just be chosen to be M as well.

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 Sect. 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, some require quantifier alternation. One example is *generalized noninterference*, as formalized in [6], 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 Sect. 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 introduce an additional

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 non-empty set of input-equivalent executions in S induced by B. 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 ξ^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 π_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 π_i is universally quantified, choose x_i in T from $\xi^T(x_1,\ldots,x_{i-1})$ and choose y_i from $B(x_i)$.*

Thus, the bisimulation helps define a strategy ξ^S to match a winning antagonist strategy ξ^T in T. We can establish that this strategy 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 φ be a general security property with automaton A representing the negation of its kernel κ. 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 φ.*

Proof. We have to show, from Definition 3, that for any winning strategy ξ^T for the antagonist in $\mathcal{G}(T,\varphi)$, there is a matching winning strategy ξ^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 ξ^S, and $x = (x_1,\ldots,x_k)$ the corresponding bundle resulting from ξ^T. By construction, y and x are input-equivalent.

Since ξ^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, ξ^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 $p' \in F$.

Refinement typically 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 φ be a general security property with automaton A representing its kernel κ. 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 φ.*

8.1 Checking General Refinement Relations

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

The optimizations discussed in Sect. 6 produce bisimular programs; the relation B in each case is defined as follows.

1. **Constant Folding:** $(t = s) \wedge (\ell^T = \texttt{L3} \rightarrow \alpha^T(\texttt{y}) = 42) \wedge (\ell^T = \texttt{L4} \rightarrow \alpha^T(\texttt{z}) = 1) \wedge (\ell^T = \texttt{L5} \rightarrow \alpha^T(\texttt{x}) = 0)$
2. **Common-Branch Factorization:** $(t = s) \vee ((\ell^T = \texttt{L2}) \wedge ((\alpha^S(\texttt{i}) < \alpha^S(\texttt{arr_size}))\,?\,(\ell^S = \texttt{L2}) : (\ell^S = \texttt{L5})) \wedge (\alpha^T = \alpha^S[\texttt{a} := \texttt{arr[0]}]))$
3. **Switching Instructions:** $(\ell^S = \ell^T) \wedge (\ell^S \neq \texttt{L6} \rightarrow \alpha^S = \alpha^T) \wedge (\ell^S = \texttt{L6} \rightarrow \alpha^S[\texttt{b[j]} := \texttt{a[j-1]}] = \alpha^T[\texttt{a[j]} := \texttt{b[j-1]}])$

There are clear similarities between the bisimulations and the corresponding refinement relations defined in Sect. 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-fold product of a bisimulation between T and S across the bundle of executions.

9 Discussion and Related Work

This work tackles the important problem of ensuring that the program transformations carried out by an optimizing 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 constructed 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 [6]. 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 [3,8,17] for non-interference and in [4] for constant-time. Manna and Pnueli show in a beautiful paper [15] how to derive custom proof rules for deductive verification of an 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 φ, 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. [23]), and trace-preserving compilation [24]. These are elegant formulations but difficult to check fully automatically, and are therefore not suitable for translation validation. The theory of hyperproperties [7] 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 this notion of refinement. Language inclusion is also not easily 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 was proposed in work on "Credible Compilation" by [16,26] and "Witnessing" by [21]. As the compiler and the witness generator do not require formal verification, the size of the trusted code base shrinks substantially. Witnessing also requires 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 [30], while refinement checking can be implemented in around 1,500 lines of code; much of this code comprises a reusable witness validator.

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. Refinement relations for correctness preserve *all* linear-time properties defined over propositions common to both programs. This is necessary as a complete specification of correctness is usually not available in practice. On the other hand, security properties are likely to be known in advance (e.g., "do not leak secret keys"). This motivates our construction of property-specific refinement relations.

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. That is the case 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 convert a higher-level language (say LLVM bytecode) to a lower-level one (say `x86` machine code). The so-called "attack surface" is then different, so it is necessary to incorporate a notion of *back-translation* of failures [9] in the refinement proof rules. How best to do so is an intriguing topic for future work.

Another question for 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.

A number of practical concerns must be addressed to implement this methodology. An important one is the development of a convenient notation for specifying the desired security properties at the source program level. It is also necessary to define how a security property is transformed through a program optimization. For instance, if a transformation introduces fresh variables, it is necessary to determine whether those variables are assigned a high or low security level for a non-interference property.

Acknowledgments. The authors were supported, in part, by NSF grant CCF-1563393 from the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Kedar Namjoshi would like to acknowledge fruitful discussions during a Dagstuhl Seminar on Secure Compilation organized in May 2018.

References

1. Abadi, M.: Protection in programming-language translations. In: Vitek, J., Jensen, C.D. (eds.) Secure Internet Programming. LNCS, vol. 1603, pp. 19–34. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48749-2_2

2. Abadi, M., Lamport, L.: The existence of refinement mappings. In: LICS 1988, pp. 165–175 (1988). https://doi.org/10.1109/LICS.1988.5115
3. de Amorim, A.A., et al.: A verified information-flow architecture. In: POPL 2014, pp. 165–178 (2014). https://doi.org/10.1145/2535838.2535839
4. Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”. In: CSF 2018, pp. 328–343 (2018). https://doi.org/10.1109/CSF.2018.00031
5. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**, 115–131 (1988). https://doi.org/10.1016/0304-3975(88)90098-9
6. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
7. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: CSF 2008, pp. 51–65 (2008). https://doi.org/10.1109/CSF.2008.7
8. Deng, C., Namjoshi, K.S.: Securing a compiler transformation. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 170–188. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_9
9. Devriese, D., Patrignani, M., Piessens, F.: Fully-abstract compilation by approximate back-translation. In: POPL 2016, pp. 164–177 (2016). https://doi.org/10.1145/2837614.2837618
10. D'Silva, V., Payer, M., Song, D.X.: The correctness-security gap in compiler optimization. In: SPW 2015, pp. 73–87 (2015). https://doi.org/10.1109/SPW.2015.33
11. Fournet, C., Guernic, G.L., Rezk, T.: A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In: CCS 2009, pp. 432–441 (2009). https://doi.org/10.1145/1653662.1653715
12. Howard, M.: 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
13. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: PLDI 2014, pp. 216–226 (2014). https://doi.org/10.1145/2594291.2594334
14. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL 2006, pp. 42–54 (2006). https://doi.org/10.1145/1111037.1111042
15. Manna, Z., Pnueli, A.: Specification and verification of concurrent programs by ∀-automata. In: Banieqbal, B., Barringer, H., Pnueli, A. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 124–164. Springer, Heidelberg (1989). https://doi.org/10.1007/3-540-51803-7_24
16. Marinov, D.: Credible compilation. Ph.D. thesis, Massachusetts Institute of Technology (2000)
17. Murray, T.C., Sison, R., Engelhardt, K.: COVERN: a logic for compositional verification of information flow control. In: EuroS&P 2018, pp. 16–30 (2018). https://doi.org/10.1109/EuroSP.2018.00010
18. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: Ramesh, S., Sivakumar, G. (eds.) FSTTCS 1997. LNCS, vol. 1346, pp. 284–296. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0058037
19. Namjoshi, K.S.: Witnessing an SSA transformation. In: VeriSure Workshop, CAV (2014). https://kedar-namjoshi.github.io/papers/Namjoshi-VeriSure-CAV-2014.pdf

20. Namjoshi, K.S., Tabajara, L.M.: Witnessing Secure Compilation (2019). https://arxiv.org/abs/1911.05866
21. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 304–323. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_17
22. Necula, G.: Translation validation of an optimizing compiler. In: (PLDI) 2000, pp. 83–95 (2000)
23. Patrignani, M., Ahmed, A., Clarke, D.: Formal approaches to secure compilation: a survey of fully abstract compilation and related work. ACM Comput. Surv. **51**(6), 125:1–125:36 (2019). https://doi.org/10.1145/3280984
24. Patrignani, M., Garg, D.: Secure compilation and hyperproperty preservation. In: CSF 2017, pp. 392–404 (2017). https://doi.org/10.1109/CSF.2017.13
25. Pnueli, A., Shtrichman, O., Siegel, M.: The Code Validation Tool (CVT)- automatic verification of a compilation process. Softw. Tools Technol. Transf. **2**(2), 192–201 (1998)
26. Rinard, M.: Credible compilation. Technical report. In: Proceedings of CC 2001: International Conference on Compiler Construction (1999)
27. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_24
28. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: PLDI 2011, pp. 283–294 (2011). https://doi.org/10.1145/1993498.1993532
29. Yang, Z., Johannesmeyer, B., Olesen, A.T., Lerner, S., Levchenko, K.: Dead store elimination (still) considered harmful. In: USENIX Security 2017, pp. 1025–1040 (2017). https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/yang
30. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formal verification of SSA-based optimizations for LLVM. In: PLDI 2013, pp. 175–186 (2013). https://doi.org/10.1145/2491956.2462164
31. Zuck, L.D., Pnueli, A., Goldberg, B.: VOC: a methodology for the translation validation of optimizing compilers. J. UCS **9**(3), 223–247 (2003)