Journey Beyond Full Abstraction

Exploring Robust Property Preservation for Secure Compilation

Carmine Abate[1] Roberto Blanco[1] Deepak Garg[2] Cătălin Hrițcu[1] Marco Patrignani[3,4] Jérémy Thibault[1]

[1]Inria Paris [2]MPI-SWS [3]Stanford University [4]CISPA Helmholz Center for Information Security

***Abstract*—Good programming languages provide helpful abstractions for writing secure code, but the security properties of the source language are generally not preserved when compiling a program and linking it with adversarial code in a low-level target language (e.g., a library or a legacy application). Linked target code that is compromised or malicious may, for instance, read and write the compiled program's data and code, jump to arbitrary memory locations, or smash the stack, blatantly violating any source-level abstraction. By contrast, a fully abstract compilation chain protects source-level abstractions all the way down, ensuring that linked adversarial target code cannot observe more about the compiled program than what some linked source code could about the source program. However, while research in this area has so far focused on preserving observational equivalence, as needed for achieving full abstraction, there is a much larger space of security properties one can choose to preserve against linked adversarial code. And the precise class of security properties one chooses crucially impacts not only the supported security goals and the strength of the attacker model, but also the kind of protections a secure compilation chain has to introduce.**

We are the first to thoroughly explore a large space of formal secure compilation criteria based on robust property preservation, i.e., the preservation of properties satisfied against arbitrary adversarial contexts. We study robustly preserving various classes of trace properties such as safety, of hyperproperties such as noninterference, and of relational hyperproperties such as trace equivalence. This leads to many new secure compilation criteria, some of which are easier to practically achieve and prove than full abstraction, and some of which provide strictly stronger security guarantees. For each of the studied criteria we propose an equivalent "property-free" characterization that clarifies which proof techniques apply. For relational properties and hyperproperties, which relate the behaviors of multiple programs, our formal definitions of the property classes themselves are novel. We order our criteria by their relative strength and show several collapses and separation results. Finally, we adapt existing proof techniques to show that even the strongest of our secure compilation criteria, the robust preservation of all relational hyperproperties, is achievable for a simple translation from a statically typed to a dynamically typed language.

1 Introduction

Good programming languages provide helpful abstractions for writing secure code. Even in unsafe low-level languages like C, safe programs have structured control flow and obey the procedure call and return discipline. Languages such as Java, C#, ML, Haskell, or Rust provide type and memory safety for all programs and additional abstractions such as modules and interfaces. Languages for efficient cryptography such as qhasm [16], Jasmin [9], and Low* [61] enforce a "constant-time" coding discipline to rule out certain side-channel attacks. Finally, verification languages such as Coq and F* [61, 70] provide abstractions such as dependent types, logical pre- and postconditions, and tracking side-effects, e.g., distinguishing pure from stateful computations. Such abstractions make reasoning about security more tractable and have, for instance, enabled developing high-assurance libraries in areas such as cryptography [9, 26, 34, 77].

However, such abstractions are not enforced all the way down by mainstream compilation chains. The security properties a program satisfies in the source language are generally not preserved when compiling the program and linking it with adversarial target code. High-assurance cryptographic libraries, for instance, get linked into real applications such as web browsers [19, 34] and web servers, which include millions of lines of legacy C/C++ code. Even if the abstractions of the source language ensure that the API of a TLS library cannot leak the server's private key [26], such guarantees are completely lost when compiling the library and linking it into a C/C++ application that can get compromised via a buffer overflow, simply allowing the adversary to read the private key from memory [31]. A compromised or malicious application that links in a high-assurance library can easily read and write its data and code, jump to arbitrary memory locations, or smash the stack, blatantly violating any source-level abstraction and breaking any security guarantee obtained by source-level reasoning.

An idea that has been gaining increasing traction recently is that it should be possible to build secure compilation chains that protect source-level abstractions even against linked adversarial target code, which is generally represented by target language *contexts*. Research in this area has so far focused on achieving *full abstraction* [2, 3, 5, 6, 7, 28, 37, 42, 44, 56, 59, 60], whose security-relevant direction ensures that even an adversarial target context cannot observe more about the compiled program than some source context could about the source program. In order to achieve full abstraction, the various parts of the secure compilation chain—including, e.g., the compiler, linker, loader, runtime, system, and hardware—have to work together to provide enough protection to the compiled program, so that whenever two programs are *observationally equivalent* in the source language (i.e., no source context can distinguish them), the two programs obtained by compiling them are observationally equivalent in the target language (i.e., no target context can distinguish them).

Observational equivalences are, however, not the only class of security properties one may want to *robustly preserve*, i.e., preserve against arbitrary adversarial contexts. One could instead be interested in robustly preserving, for instance, classes of trace properties such as safety [50] or liveness [10], or of hyperproperties [24] such as hypersafety, including variants of noninterference [11, 38, 53, 64, 76], which cover data confidentiality and integrity. However, full abstraction is generally not strong enough on its own to imply the robust preservation of any of these properties (as we show in §5, and as was also argued by others [57]). At the same time, the kind of protections one has to put in place for achieving full abstraction seem like overkill if all one wants is to robustly preserve safety or hypersafety. Indeed, it is significantly harder to hide the differences between two programs that are observationally equivalent but otherwise arbitrary, than to protect the internal invariants and the secret data of a single program. Thus, a secure compilation chain for robust safety or hypersafety can likely be more efficient than one for observational equivalence. Moreover, hiding the differences between two observationally equivalent programs is hopeless in the presence of any side-channels, while robustly preserving safety is not a problem and even robustly preserving noninterference seems possible in specific scenarios [14]. Finally, even when efficiency is not a concern (e.g., when security is enforced by *static* restrictions on target contexts [1, 6, 7, 56]), proving full abstraction is notoriously challenging even for simple languages, and conjectures have survived for decades before being settled [29].

Convinced that there is no "one-size-fits-all" criterion for secure interoperability with linked target code, we explore, for the first time, a large space of secure compilation criteria based on robust property preservation. Some of the criteria we introduce are strictly stronger than full abstraction and, moreover, immediately imply the robust preservation of well-studied property classes such as safety and hypersafety. Other criteria we introduce seem easier to practically achieve and prove than full abstraction. In general, the richer the class of security properties one tries to robustly preserve, the harder efficient enforcement becomes, so the best one can hope for is to strike a pragmatic balance between security and efficiency that matches each application domain.

For informing such difficult design decisions, we explore robustly preserving classes of trace properties (§2), of hyperproperties (§3), and of relational hyperproperties (§4). All these property notions are phrased in terms of execution traces, which for us are (finite or infinite) sequences of events such as inputs from and outputs to an external environment [48, 51]. Trace properties such as safety [50] restrict what happens along individual program traces, while hyperproperties [24] such as noninterference generalize this to predicates over multiple traces of a program. In this work we generalize this further to a new class we call *relational hyperproperties*, which relate the traces of *different* programs. An example of relational hyperproperty is trace equivalence, which requires that two programs produce the same set of traces. We work out many interesting subclasses that are also novel, such as *relational trace properties*, which relate *individual* traces of multiple programs. For instance, "On every input, program A's output is less than program B's" is a relational trace property.

We order the secure compilation criteria we introduce by their relative strength as illustrated by the partial order in Figure 1. In this Hasse diagram edges represent logical implication from higher criteria to lower ones, so the higher a criterion is, the harder it is to achieve and prove. Intuitively, the criteria based on the robust preservation of trace properties (in the yellow area) only require sandboxing the context (i.e., linked adversarial code) and protecting the internal invariants of the program from it, i.e., *only data integrity*. The criteria based on hyperproperties (in the red area) require additionally hiding the data of the program from the context, i.e., *data confidentiality*. Finally, the criteria based on relational hyperproperties (in the blue area) require additionally hiding the code of the program from the context, i.e., *code confidentiality*.

While most implications in the diagram follow directly from the inclusion between the property classes [24], *strict* inclusion between property classes does not imply strict implication between criteria. Robustly preserving two distinct property classes can in fact lead to equivalent criteria, as happens in general for hyperliveness and hyperproperties (§3.5) and, in the presence of source-level reflection or internal nondeterminism, for many criteria involving hyperproperties and relational hyperproperties (§4.5). To show the absence of more collapses, we also prove various separation results, for instance that *Robust Safety Property Preservation* (RSP) is *strictly* weaker than *Robust Trace Property Preservation* (RTP). For this, we design (counterexample) compilation chains that satisfy the weaker criterion but not the stronger one.

For each introduced secure compilation criterion we also discovered an *equivalent "property-free" characterization* that is generally better tailored for proofs and that provides important insights into what kind of techniques one can use to prove the criterion. For instance, for proving RSP and RTP we can produce a different source context to explain *each* attack trace, while for proving stronger criteria such as *Robust Hyperproperty Preservation* (RHP) we have to produce a single source context that works for *any* attack trace.

We also formally study the relation between our new security criteria and full abstraction (§5) proxied by the robust preservation of trace equivalence (RTEP), which in determinate languages—i.e., languages without internal nondeterminism—was shown to coincide with observational equivalence [21, 33]. In one direction, RTEP follows unconditionally from *Robust 2-relational Hyperproperty Preservation*, which is one of our stronger criteria. However, if the source and target languages are determinate and we make some mild extra assumptions (such as input totality [36, 75]) RTEP follows even from the weaker *Robust 2-relational relaXed safety Preservation* (R2rXP). Here, the challenge was identifying these extra assumptions and showing that they are sufficient to establish RTEP. In the other direction, we adapt a counterexample proposed by Patrignani and Garg [57] to show

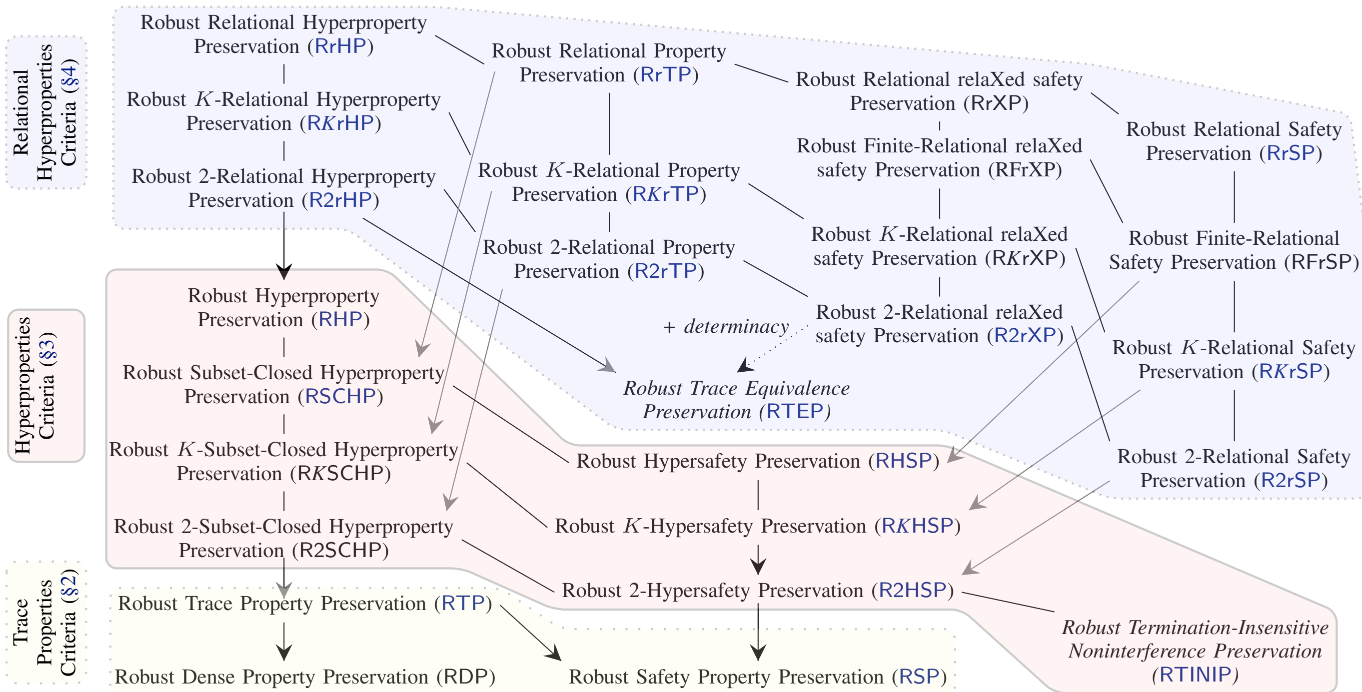

Fig. 1: Partial order with the secure compilation criteria studied in this paper. Criteria higher in the diagram imply the lower ones to which they are connected by edges. Criteria based on trace properties are grouped in a yellow area, those based on hyperproperties are in a red area, and those based on relational hyperproperties are in a blue area. Criteria with an *italics name* preserve a *single* property that belongs to the class they are connected to; the dotted edge requires an additional determinacy assumption. Finally, each edge with a thick arrow denotes a *strict* implication that we have proved as a separation result.

that RTEP (and thus full abstraction), even in conjunction with compositional compiler correctness, does *not* imply even the weakest of our criteria, RSP, RDP, and RTINIP.

Finally, we show that two proof techniques originally developed for full abstraction can be readily adapted to prove our new secure compilation criteria (§6). First, we use a "universal embedding" [56] to prove that the strongest of our secure compilation criteria, Robust Relational Hyperproperty Preservation (RrHP), is achievable for a simple translation from a statically typed to a dynamically typed first-order language with first-order functions and I/O. Second, we use the same simple translation to illustrate that for proving *Robust Finite-relational relaXed safety Preservation* (RFrXP) we can employ a "trace-based back-translation" [43, 59], a slightly less powerful but more generic technique that we extend to back-translate a finite set of finite execution prefixes into a source context. This second technique is applicable to all criteria implied by RFrXP, which includes robust preservation of safety, of hypersafety, and in a determinate setting also of trace (and thus observational) equivalence.

In summary, our paper makes five **contributions**:

C1. We phrase the formal security guarantees obtained by protecting compiled programs from adversarial contexts in terms of robustly preserving classes of properties. We are the first to explore a large space of security criteria based on this idea, including criteria that provide strictly stronger security guarantees than full abstraction, and also criteria that are easier to practically achieve and prove, which is important for building more realistic secure compilation chains.

C2. We carefully study each new secure compilation criterion and the non-trivial relations between them. For each criterion we propose a property-free characterization that clarifies which proof techniques apply. For relating the criteria, we order them by their relative strength, show several interesting collapses, and prove several challenging separation results.

C3. We introduce *relational* properties and hyperproperties, which are new property classes of independent interest, even outside of secure compilation.

C4. We formally study the relation between our security criteria and full abstraction. In one direction, we show that determinacy is enough for robustly preserving classes of relational properties and hyperproperties to imply preservation of observational equivalence. In the other direction, we show that, even when assuming compiler correctness, full abstraction does not imply even our weakest criteria.

C5. We show that two existing proof techniques originally developed for full abstraction can be readily adapted to our new criteria, which is important since good proof techniques are difficult to find in this space [56, 60].

The paper closes with discussions of related (§7) and future work (§8). The online appendix at `https://arxiv.org/abs/1807.04603` contains omitted technical details. Many of the theorems formally or informally mentioned in the paper were also mechanized in the Coq proof assistant and are marked with ; this development has around 4400 lines of code and is available at `https://github.com/secure-compilation/exploring-robust-property-preservation`

2 Robustly Preserving Trace Properties

In this section we look at robustly preserving classes of *trace properties*, and first study the robust preservation of *all* trace properties and its relation to correct compilation (§2.1). We then look at robustly preserving *safety properties* (§2.2), which are the trace properties that can be falsified by a finite trace prefix (e.g., a program never performs a certain dangerous system call). These criteria are grouped in the Trace Properties yellow area in Figure 1. We also carefully studied the robust preservation of *liveness properties*, but it turns out that the very definition of liveness is highly dependent on the specifics of the program execution traces, which makes that part more technical. For saving space and avoiding a technical detour, we relegate to the appendix the details of our CompCert-inspired trace model, as well as the part about liveness.

2.1 Robust Trace Property Preservation (RTP)

Like all secure compilation criteria we study in this paper, the RTP criterion below is a *generic* property of an arbitrary *compilation chain*, which includes a source and a target language, each with a notion of partial programs (P) and contexts (C) that can be linked together to produce whole programs ($C[P]$), and each with a trace-producing semantics for whole programs ($C[P] \rightsquigarrow t$). The sets of partial programs and of contexts of the source and target languages are unconstrained parameters of our secure compilation criteria; our criteria make no assumptions about their structure, or whether the program or the context gets control initially once linked and executed (e.g., the context could be an application that embeds a library program or the context could be a library that is embedded into an application program).[1] The traces produced by the source and target semantics[2] are arbitrary for RTP, but for RSP we have to consider traces with a specific structure (finite or infinite sequences of events drawn from an arbitrary set). Intuitively, traces capture the interaction between a whole program and its external environment, including for instance user input, output to a terminal, network communication, system calls, etc. [48, 51]. As opposed to a context, which is just a piece of a program, the environment's behavior is not (and often *cannot* be) modeled by the programming language, beyond the (often nondeterministic) interaction events that we store in the trace. Finally, a compilation chain includes a compiler: the compilation of a partial source program P is a partial target program we write $\mathsf{P}{\downarrow}$.[3]

The responsibility of enforcing secure compilation does not have to rest just with the compiler, but may be freely shared by various parts of the compilation chain. In particular, to help enforce security, the target-level *linker* could disallow linking with a suspicious context (e.g., one that is not well-typed [1, 6, 7, 56]) or could always allow linking but introduce protection barriers between the program and the context (e.g., by instrumenting the program [28, 56] or the context [4, 72, 73] to introduce dynamic checks). Similarly, the semantics of the target language can include various protection mechanisms (e.g., processes with different virtual address spaces [62], protected enclaves [59], capabilities [23, 68, 74], etc.). Finally, the compiler might have to refrain from aggressive optimizations that would break security [14, 30, 67]. Our secure compilation criteria are agnostic to the concrete enforcement mechanism used by the compilation chain to protect the compiled program from the adversarial target context.

Trace properties are defined simply as sets of allowed traces [50]. A whole program $C[P]$ *satisfies* a trace property π when the set of traces produced by $C[P]$ is included in the set π or, formally, $\{t \mid C[P] \rightsquigarrow t\} \subseteq \pi$. More interestingly, we say that a partial program P *robustly satisfies* [39, 49, 71] a trace property π when P linked with *any* (adversarial) context C satisfies π. Armed with this, *Robust Trace Property Preservation* (RTP) is defined as the preservation of robust satisfaction of all trace properties. So if a partial source program P robustly satisfies a trace property $\pi \in 2^{\mathit{Trace}}$ (wrt. all source contexts) then its compilation $\mathsf{P}{\downarrow}$ must also robustly satisfy π (wrt. all target contexts). If we unfold all intermediate definitions, a compilation chain satisfies RTP iff:

$$\begin{aligned}\mathsf{RTP}: \quad & \forall \pi \in 2^{\mathit{Trace}}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t.\ \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow \\ & \qquad (\forall \mathbf{C_T}\ t.\ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow t \in \pi)\end{aligned}$$

This definition directly captures which properties (specifically, all trace properties) of the source are robustly preserved by the compilation chain. However, in order to prove that a compilation chain satisfies RTP we propose an equivalent () "property-free" characterization, which we call RTC (for "RTP Characterization"):

$$\mathsf{RTC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall t.\ \ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \exists \mathsf{C_S}.\ \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

RTC requires that, given a compiled program $\mathsf{P}{\downarrow}$ and a target context $\mathbf{C_T}$ which together produce an attack trace t, we can generate a source context $\mathsf{C_S}$ that causes trace t to be produced by P. When proving that a compilation chain satisfies RTC we can pick a different context $\mathsf{C_S}$ for each t and, in fact, try to construct $\mathsf{C_S}$ from trace t or from the execution $\mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t$.

We present similar property-free characterizations for each of our criteria (Figure 1). However, for criteria stronger than RTP, a single context $\mathsf{C_S}$ will have to work for more than one trace. In general, the shape of the property-free character-

[1] One limitation of our formal setup, is that for simplicity we assume that any partial program can be linked with any context, irrespective of their interfaces (e.g., types or specs). One can extend our criteria to take interfaces into account, as we illustrate in the appendix for the example in §6.

[2] In this paper we assume for simplicity that traces are exactly the same in both the source and target language, as is also the case in the CompCert verified C compiler [51]. We hope to lift this restriction in the future (§8).

[3] For easier reading, we use a blue, sans-serif font for source elements, an **orange, bold** font for **target** elements and a *black, italic* font generically for elements of either language.

ization explains what information can be used to produce the source context $\mathsf{C_S}$ when proving a compilation chain secure.

Relation to compiler correctness RTC is similar to "backward simulation" (TC), a standard compiler *correctness* criterion [51]. Let W denote a whole program.

$$\mathsf{TC}: \quad \forall \mathsf{W}.\ \forall t.\ \mathsf{W}{\downarrow} \rightsquigarrow t \Rightarrow \mathsf{W} \rightsquigarrow t$$

Maybe slightly less known is that this property-free characterization of correct compilation also has an equivalent property-full characterization as the preservation of all trace properties:

$$\mathsf{TP}: \quad \forall \pi \in 2^{\mathit{Trace}}.\ \forall \mathsf{W}.\ (\forall t.\ \mathsf{W} \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall t.\ \mathsf{W}{\downarrow} \rightsquigarrow t \Rightarrow t \in \pi)$$

The major difference compared to RTP is that TP only preserves the trace properties of whole programs and does not consider adversaries. In contrast, RTP allows linking a compiled partial program with arbitrary target contexts and protects the program so that all *robustly satisfied* trace properties are preserved. In general, RTP and TP are incomparable. However, RTP strictly implies TP when whole programs (W) are a subset of partial programs (P) and, additionally, the semantics of *whole* programs is independent of any linked context (i.e., $\forall W\ t\ C.\ W \rightsquigarrow t \iff C[W] \rightsquigarrow t$, which happens, intuitively, when the whole program starts execution and, being whole, never calls into the context).

More compositional criteria for compiler correctness have also been proposed [45, 55]. At a minimum such criteria allow linking with contexts that are the compilation of source contexts [45], which can be formalized as follows:

$$\mathsf{SCC}: \quad \forall \mathsf{P}.\ \forall \mathsf{C_S}.\ \forall t.\ \mathsf{C_S}{\downarrow}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

More permissive criteria allow linking with any target context that behaves like some source context [55], which in our setting can be written as:

$$\mathsf{CCC}: \ \forall \mathsf{P}\ \mathbf{C_T}\ \mathsf{C_S}\ t.\ \mathbf{C_T}{\approx}\mathsf{C_S} \wedge \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

Here $\approx$ relates equivalent partial programs in the target and the source, and could, for instance, be instantiated with a cross-language logical relation [6, 55]. RTP is incomparable to SCC and CCC. On the one hand, RTP allows linking with *arbitrary* target-level contexts, which is not allowed by SCC and CCC, and requires inserting strong protection barriers. On the other hand, in RTP all source-level reasoning has to be done with respect to an *arbitrary* source context, while with SCC and CCC one can reason about a known source context.

2.2 Robust Safety Property Preservation (RSP)

Robust safety preservation is an interesting criterion for secure compilation because it is easier to achieve and prove than most criteria of Figure 1, while still being quite expressive [39, 71].

Recall that a trace property is a safety property if, within any (possibly infinite) trace that violates the property, there exists a finite "bad prefix" that violates it. We write $m \leq t$ for the prefix relation between a finite trace prefix m and a trace t. Using this we define safety properties in the usual way [10, 50, 66]:

$$\mathit{Safety} \triangleq \{\pi \in 2^{\mathit{Trace}} \mid \forall t \notin \pi.\ \exists m \leq t.\ \forall t' \geq m.\ t' \notin \pi\}$$

The definition of RSP simply restricts the preservation of robust satisfaction from all trace properties in RTP to only safety properties; otherwise the definition is exactly the same:

$$\mathsf{RSP}: \quad \forall \pi \in \mathit{Safety}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t.\ \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall \mathbf{C_T}\ t.\ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow t \in \pi)$$

One might wonder how safety properties can be *robustly* satisfied in the source, given that execution traces can contain events emitted not just by the partial program but also by the adversarial context, which could trivially emit "bad events" and, hence, violate any safety property. A first alternative is for the semantics of the source language to simply prevent the context from producing any events, maybe other than termination, or, at least, prevent the context from producing any events the safety properties of interest consider bad. The compilation chain has then to "sandbox" the context to restrict the events it can produce [72, 73]. A second alternative is for the source semantics to record enough information in the trace so that one can determine the origin of each event—the partial program or the context. Then, safety properties in which the context's events are never bad can be robustly satisfied. With this second alternative, the obtained global guarantees are weaker, e.g., one cannot enforce that the whole program never makes a dangerous system call, but only that the partial program cannot be tricked by the context into making it.

The equivalent ( ) property-free characterization for RSP requires one to back-translate a program (P), a target context ($\mathbf{C_T}$), and a *finite* bad trace prefix ($\mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow m$) into a source context ($\mathsf{C_S}$) producing the same finite trace prefix (m) in the source ($\mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow m$):

$$\mathsf{RSC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall m.\ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow m \Rightarrow \exists \mathsf{C_S}.\ \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow m$$

Syntactically, the only change with respect to RTC is the switch from whole traces t to finite trace prefixes m. As for RTC, we can pick a different context $\mathsf{C_S}$ for each execution $\mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow m$. (In our formalization we define $W \rightsquigarrow m$ generically as $\exists t{\geq}m.\ W \rightsquigarrow t$.) The fact that for RSC these are *finite* execution prefixes can significantly simplify the back-translation into source contexts (as we show in §6.4).

It is trivially true that RTP implies RSP, since the former robustly preserves all trace properties while the latter only robustly preserves safety properties. We have also proved that RTP *strictly implies* RSP.

Theorem 2.1. RTP $\Rightarrow$ RSP, but RSP $\not\Rightarrow$ RTP

Proof sketch. As explained above, RTP $\Rightarrow$ RSP is trivial. Showing strictness requires constructing a counterexample compilation chain to the reverse implication. We take any target language that can produce infinite traces. We take the source language to be a variant of the target with the same partial programs, but where we extend whole programs and contexts with a bound on the number of events they can produce before being terminated. Compilation simply erases this bound. This compilation chain satisfies RSP (equivalently, RSC) but not RTP. To show that it satisfies RSC, we simply back-translate a target context $\mathbf{C_T}$ and a finite trace prefix m to a source context $(\mathbf{C_T}, \mathit{length}(m))$ that uses the length of m as the allowed bound, so this context can still produce m in the source without being prematurely terminated. However, this compilation chain does not satisfy RTP, since in the source all executions are finite and, hence, no infinite target trace can be simulated by *any* source context. $\square$

3 Robustly Preserving Hyperproperties

So far, we have studied the robust preservation of trace properties, which are properties of *individual* traces of a program. In this section we generalize this to *hyperproperties*, which are properties of *multiple* traces of a program [24]. A well-known hyperproperty is noninterference [11, 38, 53, 76], which usually requires considering two traces of a program that differ on secret inputs. Another hyperproperty is bounded mean response time over all executions. We study robust preservation of many subclasses of hyperproperties: all hyperproperties (§3.1), subset-closed hyperproperties (§3.2), hypersafety and K-hypersafety (§3.3), and hyperliveness (§3.5). These criteria are in the red area in Figure 1.

3.1 Robust Hyperproperty Preservation (RHP)

While trace properties are sets of traces, hyperproperties are sets of sets of traces [24]. We call the set of traces of a whole program W the *behavior* of W: $\mathtt{Behav}\,(W) = \{t \mid W \rightsquigarrow t\}$. A hyperproperty is a set of allowed behaviors. Program W satisfies hyperproperty H if the behavior of W is a member of H, i.e., $\mathtt{Behav}\,(W) \in H$, or, equivalently, $\{t \mid W \rightsquigarrow t\} \in H$. Contrast this to W satisfying trace property π, which holds if the behavior of W is a subset of the set π, i.e., $\mathtt{Behav}\,(W) \subseteq \pi$, or, equivalently, $\forall t.\ W \rightsquigarrow t \Rightarrow t \in \pi$. So while a trace property determines whether each individual trace of a program should be allowed or not, a hyperproperty determines whether the set of traces of a program, its behavior, should be allowed or not. For instance, the trace property $\pi_{123} = \{t_1, t_2, t_3\}$ is satisfied by programs with behaviors such as $\{t_1\}$, $\{t_2\}$, $\{t_2, t_3\}$, and $\{t_1, t_2, t_3\}$, but a program with behavior $\{t_1, t_4\}$ does not satisfy π_{123}. A hyperproperty like $H_{1+23} = \{\{t_1\}, \{t_2, t_3\}\}$ is satisfied only by programs with behavior $\{t_1\}$ or with behavior $\{t_2, t_3\}$. A program with behavior $\{t_2\}$ does not satisfy H_{1+23}, so hyperproperties can express that if some traces (e.g., t_2) are possible then some other traces (e.g., t_3) should also be possible. A program with behavior $\{t_1, t_2, t_3\}$ also does not satisfy H_{1+23}, so hyperproperties can express that if some traces (e.g., t_2 and t_3) are possible then some other traces (e.g., t_1) should not be possible. Finally, trace properties can be easily lifted to hyperproperties: A trace property π becomes the hyperproperty $[\pi] = 2^{\pi}$, the powerset of π.

We say that a partial program P *robustly satisfies* a hyperproperty H if it satisfies H for any context C. Given this we define RHP as the preservation of robust satisfaction of arbitrary hyperproperties:

$$\mathsf{RHP}:\quad \forall H \in 2^{2^{\mathit{Trace}}}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}]) \in H) \Rightarrow (\forall \mathbf{C_T}.\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}]) \in H)$$

The equivalent characterization of RHP is RHC :

$$\mathsf{RHC}:\quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}])$$

$$\mathsf{RHC}:\quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall t.\ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \iff \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

This requires that, for every partial program P and target context $\mathbf{C_T}$, there is a (back-translated) source context $\mathsf{C_S}$ that perfectly preserves the set of traces of $\mathbf{C_T}\,[\mathsf{P}{\downarrow}]$ when linked to P. There are two differences from RTP: (1) the $\exists \mathsf{C_S}$ and $\forall t$ quantifiers are swapped, so the back-translated $\mathsf{C_S}$ must work for all traces t, and (2) the implication in RTC ($\Rightarrow$) became a two-way implication in RHC ($\iff$), so compilation has to perfectly preserve the set of traces. In particular the compiler cannot refine behavior (remove traces), e.g., it cannot implement nondeterministic scheduling via a deterministic scheduler.

In the following subsections we study restrictions of RHP to various subclasses of hyperproperties. To prevent duplication we define $\mathsf{RHP}(X)$ to be the robust satisfaction of a class X of hyperproperties (so RHP above is simply $\mathsf{RHP}(2^{2^{\mathit{Trace}}})$):

$$\mathsf{RHP}(X):\quad \forall H \in X.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}]) \in H) \Rightarrow (\forall \mathbf{C_T}.\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}]) \in H)$$

3.2 Robust Subset-Closed Hyperproperty Preservation (RSCHP)

If one restricts robust preservation to only subset-closed hyperproperties then refinement of behavior is again allowed. A hyperproperty H is subset-closed, written $H{\in}\mathit{SC}$, if for any two behaviors $b_1{\subseteq}b_2$, if $b_2{\in}H$ then $b_1{\in}H$. For instance, the lifting $[\pi]$ of any trace property π is subset-closed, but the hyperproperty H_{1+23} above is not. It can be made subset-closed by allowing all smaller behaviors: $H^{\mathit{SC}}_{1+23} = \{\varnothing, \{t_1\}, \{t_2\}, \{t_3\}, \{t_2, t_3\}\}$ is subset-closed.

Robust Subset-Closed Hyperproperty Preservation (RSCHP) is simply defined as $\mathsf{RHP}(\mathit{SC})$. The equivalent property-free characterization of RSCHC simply gives up the $\Leftarrow$ direction of RHC:

$$\mathsf{RSCHC}:\quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall t.\ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

The most interesting subclass of subset-closed hyperproperties is hypersafety, which we discuss next. The appendix also studies K-subset-closed hyperproperties [52], which can be seen as generalizing K-hypersafety below.

3.3 Robust Hypersafety Preservation (RHSP)

Hypersafety is a generalization of safety that is very important in practice, since several important notions of noninterference are hypersafety, such as termination-insensitive noninterference [11, 35, 64], observational determinism [53, 63, 76], and nonmalleable information flow [20].

According to Alpern and Schneider [10], the "bad thing" that a safety property disallows must be *finitely observable* and *irremediable*. For safety the "bad thing" is a finite trace prefix that cannot be extended to any trace satisfying the safety property. For hypersafety, Clarkson and Schneider [24] generalize the "bad thing" to a finite set of finite trace prefixes that they call an *observation*, drawn from the set $\mathit{Obs} = 2^{\mathit{FinPref}}_{\mathit{Fin}}$, which denotes the set of all finite subsets of finite prefixes. They then lift the prefix relation to sets: an observation $o \in \mathit{Obs}$ is a prefix of a behavior $b \in 2^{\mathit{Trace}}$, written $o{\leq}b$, if $\forall m \in o.\ \exists t \in b.\ m{\leq}t$. Finally, they define hypersafety analogously to safety, but the domains involved include an extra level of sets:

$$\mathit{Hypersafety} \triangleq \{H \mid \forall b {\notin} H.\ (\exists o {\in} \mathit{Obs}.\ o{\leq}b \wedge (\forall b'{\geq}o.\ b'{\notin}H))\}$$

Here the "bad thing" is an observation o that cannot be extended to a behavior b' satisfying the hypersafety property H. We use this to define *Robust Hypersafety Preservation* (RHSP) as RHP(*Hypersafety*) and propose the following equivalent characterization for it:

$$\mathsf{RHSC}:\ \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall o \in \mathit{Obs}.$$
$$o \leq \mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}{\downarrow}\right]\right) \Rightarrow \exists \mathsf{C_S}.\ o \leq \mathtt{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)$$

This says that to prove RHSP one needs to be able to back-translate a partial program P, a context $\mathbf{C_T}$, and a prefix o of the behavior of $\mathbf{C_T}[\mathsf{P}{\downarrow}]$, to a source context $\mathsf{C_S}$ so that the behavior of $\mathsf{C_S}[\mathsf{P}]$ extends o. It is possible to use the finite set of finite executions corresponding to observation o to drive this back-translation (as we do in §6.4).

For hypersafety the involved observations are finite sets but their cardinality is otherwise unrestricted. In practice though, most hypersafety properties can be falsified by very small sets: counterexamples to termination-insensitive noninterference [11, 35, 64] and observational determinism [53, 63, 76] are observations containing 2 finite prefixes, while counterexamples to nonmalleable information flow [20] are observations containing 4 finite prefixes. To account for this, Clarkson and Schneider [24] introduce K-hypersafety as a restriction of hypersafety to observations of a fixed cardinality K. Given $\mathit{Obs}_K = 2^{\mathit{FinPref}}_{\mathit{Fin}(K)}$, the set of observations with cardinality K, all definitions and results above can be ported to K-hypersafety by simply replacing *Obs* with Obs_K. Specifically, we denote by RKHSP the criterion RHP(K-*Hypersafety*).

The set of lifted safety properties, $\{[\pi] \mid \pi \in \mathit{Safety}\}$, is precisely the same as 1-hypersafety, since the counterexample for them is a single finite prefix. For a more interesting example, termination-insensitive noninterference (*TINI*) [11, 35, 64] can be defined as follows in our setting:

$$\begin{aligned} \mathit{TINI} \triangleq \{b \mid \forall t_1\ t_2{\in}b.\ (t_1\ \mathit{terminating} \wedge t_2\ \mathit{terminating} \\ \wedge\ \mathit{pub\text{-}inputs}(t_1){=}\mathit{pub\text{-}inputs}(t_2)) \\ \Rightarrow \mathit{pub\text{-}events}(t_1){=}\mathit{pub\text{-}events}(t_2)\} \end{aligned}$$

This requires that trace events are either inputs or outputs, each of them associated with a security level: public or secret. *TINI* ensures that for any two terminating traces of the program behavior for which the two sequences of public inputs are the same, the two sequences of public events—inputs and outputs—are also the same. *TINI* is 2-hypersafety, since $b \notin \mathit{TINI}$ implies that there exist finite traces t_1 and t_2 that agree on the public inputs but not on all public events, so we can simply take $o = \{t_1, t_2\}$. Since the traces in o are already terminated, any extension b' of o can only add extra traces, i.e., $\{t_1, t_2\} \subseteq b'$, so $b' \notin \mathit{TINI}$ as needed to conclude that *TINI* is in 2-hypersafety. In Figure 1, we write *Robust Termination-Insensitive Noninterference Preservation* (RTINIP) for RHP($\{\mathit{TINI}\}$).

3.4 Separation Between Properties and Hyperproperties

Enforcing RHSP is strictly more demanding than enforcing RSP. Because even R2HSP (robust 2-hypersafety preservation) implies RTINIP, a compilation chain satisfying R2HSP has to make sure that a target-level context cannot infer more information about the internal data of $\mathsf{P}{\downarrow}$ than a source context could infer about the data of P. By contrast, a RSP compilation chain can allow arbitrary *reads* of $\mathsf{P}{\downarrow}$'s internal data, even if P's data is private at the source level. Intuitively, for proving RSC, the source context produced by back-translation can guess any secret $\mathsf{P}{\downarrow}$ receives in the *single* considered execution, but for R2HSP the single source context needs to work for *two* different executions, potentially with two different secrets, so guessing is no longer an option. We use this idea to prove a more general separation result RTP $\not\Rightarrow$ RTINIP, by exhibiting a toy compilation chain in which private variables are readable in the target language, but not in source.

Theorem 3.1. RTP $\not\Rightarrow$ RTINIP

This implies a strict separation between all criteria based on hyperproperties (the red area in Figure 1, having RTINIP as the bottom) and all the ones based on trace properties (the yellow area in Figure 1 having RTP as the top).

Using a more complex counterexample involving a system of K linear equations, we have also shown that, for any K, robust preservation of K-hypersafety, does not imply robust preservation of $(K{+}1)$-hypersafety.

Theorem 3.2. $\forall K.$ RKHSP $\not\Rightarrow$ R$(K{+}1)$HSP

3.5 Where Is Robust Hyperliveness Preservation?

Robust Hyperliveness Preservation (RHLP) does not appear in Figure 1, because it is provably equivalent to RHP (or, equivalently, RHC). We define RHLP as RHP(*Hyperliveness*) for the following standard definition of *Hyperliveness* [24]:

$$\mathit{Hyperliveness} \triangleq \{H \mid \forall o \in \mathit{Obs}.\ \exists b{\geq}o.\ b \in H\}$$

The proof that RHLP implies RHC involves showing that $\{b \mid b{\neq}\mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}{\downarrow}\right]\right)\}$, the hyperproperty allowing all behaviors other than $\mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}{\downarrow}\right]\right)$, is hyperliveness. Another way to obtain this result is from the fact that, as in previous models [10], each hyperproperty can be decomposed as the intersection of two hyperliveness properties. This collapse of *preserving* hyperliveness and *preserving* all hyperproperties happens irrespective of the adversarial contexts.

4 Robustly Preserving Relational Hyperproperties

Trace properties and hyperproperties are predicates on the behavior of a single program. However, we may be interested in showing that compilation robustly preserves *relations* between the behaviors of two or more programs. For example, suppose we optimize a partial source program $\mathsf{P_1}$ to $\mathsf{P_2}$ such that $\mathsf{P_2}$ runs faster than $\mathsf{P_1}$ in any source context. We may want compilation to preserve this "runs faster than" *relation* between the two program behaviors against arbitrary target contexts. Similarly, in any source context, the behaviors of $\mathsf{P_1}$ and $\mathsf{P_2}$ may be equal and we may want the compiler to preserve such trace equivalence [12, 25] in arbitrary target contexts. This last criterion, which we call *Robust Trace Equivalence Preservation* (RTEP) in Figure 1, is interesting because in various determinate settings [21, 33] it coincides with preserving observational equivalence, the security-relevant part of full abstraction (see §5).

In this section, we study the robust preservation of such *relational hyperproperties* and several interesting subclasses, still relating the behaviors of multiple programs. Unlike hyperproperties and trace properties, relational hyperproperties have not been defined as a general concept in the literature, so even their definitions are new. We describe relational hyperproperties and their robust preservation in §4.1, then look at subclasses induced by what we call *relational properties* (§4.2) and *relational safety properties* (§4.3). The appendix presents a few other subclasses. The corresponding secure compilation criteria are grouped in the blue area in Figure 1. In §4.4 we show that, in general, none of these relational criteria are implied by any non-relational criterion (from §2 and §3), while in §4.5 we show two specific situations in which most relational criteria collapse to non-relational ones.

4.1 Relational Hyperproperty Preservation (RrHP)

We define a *relational hyperproperty* as a predicate (relation) on a sequence of behaviors of some length. A sequence of programs of the same length is then said to have the relational hyperproperty if their behaviors collectively satisfy the predicate. Depending on the arity of the predicate, we get different subclasses of relational hyperproperties. For arity 1, the resulting subclass describes relations on the behavior of individual programs, which coincides with hyperproperties (§3). For arity 2, the resulting subclass consists of relations on the behaviors of two programs. Both examples described at the beginning of this section lie in this subclass. This generalizes to any finite arity K (predicates on behaviors of K programs), and to the infinite arity.

Next, we define the robust preservation of these subclasses. For arity 2, *robust 2-relational hyperproperty preservation*, R2rHP, is defined as follows:

$$\begin{aligned}\mathsf{R2rHP}:\ & \forall R \in 2^{(\mathsf{Behavs}^2)}.\ \forall \mathsf{P_1}\ \mathsf{P_2}.\\ & (\forall \mathsf{C_S}.\ (\mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P_1}]),\ \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P_2}])) \in R) \Rightarrow\\ & (\forall \mathbf{C_T}.\ (\mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P_1}\!\downarrow]),\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P_2}\!\downarrow])) \in R)\end{aligned}$$

R2rHP says that for any binary relation R on behaviors of programs, if the behaviors of $\mathsf{P_1}$ and $\mathsf{P_2}$ satisfy R in every source context, then so do the behaviors of $\mathsf{P_1}\!\downarrow$ and $\mathsf{P_2}\!\downarrow$ in every target context. In other words, a compiler satisfies R2rHP iff it preserves any relation between pairs of program behaviors that hold in all contexts. In particular, such a compilation chain preserves trace equivalence in all contexts (i.e., RTEP), which we obtain by instantiating R with equality in the above definition (🐓). If execution time is recorded on program traces, then such a compilation chain also preserves relations like "the average execution time of $\mathsf{P_1}$ across all inputs is no more than the average execution time of $\mathsf{P_2}$ across all inputs" and "$\mathsf{P_1}$ runs faster than $\mathsf{P_2}$ on all inputs" (i.e., $\mathsf{P_1}$ is an improvement of $\mathsf{P_2}$). This last property can also be described as a relational predicate on pairs of traces (rather than behaviors); we return to this point in §4.2.

R2rHP has an equivalent (🐓) property-free variant that does not mention relations R:

$$\begin{aligned}\mathsf{R2rHC}:\ \forall \mathsf{P_1}\ \mathsf{P_2}\ \mathbf{C_T}.\exists \mathsf{C_S}.\ & \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P_1}\!\downarrow]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P_1}])\\ \wedge\ & \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P_2}\!\downarrow]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P_2}])\end{aligned}$$

R2rHC is a generalization of RHC from §3.1, but now the same source context $\mathsf{C_S}$ has to simulate the behaviors of *two* target programs, $\mathbf{C_T}\,[\mathsf{P_1}\!\downarrow]$ and $\mathbf{C_T}\,[\mathsf{P_2}\!\downarrow]$.

R2rHP generalizes to any finite arity K in the obvious way, yielding RKrHP. Finally, this also generalizes to the infinite arity. We call this *Robust Relational Hyperproperty Preservation* (RrHP):

$$\begin{aligned}\mathsf{RrHP}:\ & \forall R \in 2^{(\mathsf{Behavs}^\omega)}.\ \forall \mathsf{P_1}, .., \mathsf{P_K}, ...\\ & (\forall \mathsf{C_S}.\ (\mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P_1}]), .., \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P_K}]), ..) \in R) \Rightarrow\\ & (\forall \mathbf{C_T}.\ (\mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P_1}\!\downarrow]), .., \mathtt{Behav}\,(\mathbf{C_S}\,[\mathsf{P_K}\!\downarrow]), ..) \in R)\end{aligned}$$

RrHP is the strongest criterion we study and, hence, it is the highest point in Figure 1. This includes robustly preserving predicates on all programs of the language, although we have not yet found practical uses for this. More interestingly, RrHP has a very natural equivalent property-free characterization, RrHC, requiring for every target context $\mathbf{C_T}$, a source context $\mathsf{C_S}$ that can simulate the behavior of $\mathbf{C_T}$ for *any* program:

$$\mathsf{RrHC}:\quad \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall \mathsf{P}.\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}\!\downarrow]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}])$$

It is instructive to compare the property-free characterizations of the preservation of robust trace properties (RTC), hyperproperties (RHC), and relational hyperproperties (RrHC). In RTC, the source context $\mathsf{C_S}$ may depend on the target context $\mathbf{C_T}$, the source program P and a given trace t. In RHC, $\mathsf{C_S}$ may depend only on $\mathbf{C_T}$ and P. In RrHC, $\mathsf{C_S}$ may depend only on $\mathbf{C_T}$. This directly reflects the increasing expressive power of trace properties, hyperproperties, and relational hyperproperties, as predicates on traces, behaviors (set of traces), and sequences of behaviors, respectively.

4.2 Relational Trace Property Preservation (RrTP)

Relational (trace) properties are the subclass of relational hyperproperties that are fully characterized by relations on *individual* traces of multiple programs. For example, the relation "$\mathsf{P_1}$ runs faster than $\mathsf{P_2}$ on every input" is a 2-ary relational property characterized by pairs of traces, one from $\mathsf{P_1}$ and the other from $\mathsf{P_2}$, which either differ in the input or where the execution time in $\mathsf{P_1}$'s trace is less than that in $\mathsf{P_2}$'s trace. Formally, relational properties of arity K are a subclass of relational hyperproperties of the same arity. A K-ary relational hyperproperty is a relational (trace) property if there is a K-ary relation R on *traces* such that $P_1, .., P_K$ are related by the relational hyperproperty iff $(t_1, \ldots, t_k) \in R$ for any $t_1 \in \texttt{Behav}(P_1), \ldots, t_k \in \texttt{Behav}(P_K)$. Next, we define the robust preservation of relational properties of different arities. For arity 1, this coincides with RTP from §2.1. For arity 2, we define *Robust 2-relational Property Preservation*:

$$\mathsf{R2rTP} : \forall R \in 2^{(\mathit{Trace}^2)}.\ \forall \mathsf{P_1}\ \mathsf{P_2}.$$
$$\big(\forall \mathsf{C_S}\, t_1\, t_2.\, (\mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow t_1 \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow t_2) \Rightarrow (t_1, t_2) \in R\big) \Rightarrow$$
$$\big(\forall \mathbf{C_T}\, t_1\, t_2.\, (\mathbf{C_T}\,[\mathsf{P_1}\!\downarrow] \rightsquigarrow t_1 \wedge \mathbf{C_T}\,[\mathsf{P_2}\!\downarrow] \rightsquigarrow t_2) \Rightarrow (t_1, t_2) \in R\big)$$

R2rTP is weaker than its relational hyperproperty counterpart, R2rHP (§4.1): Unlike R2rHP, R2rTP does not imply the robust preservation of relations like "the average execution time of $\mathsf{P_1}$ across all inputs is no more than the average execution time of $\mathsf{P_2}$ across all inputs" (a relation between average execution times of $\mathsf{P_1}$ and $\mathsf{P_2}$ cannot be characterized by any relation between individual traces of $\mathsf{P_1}$ and $\mathsf{P_2}$).

R2rTP also has an equivalent () characterization:

$$\mathsf{R2rTC} :\ \forall \mathsf{P_1}\ \mathsf{P_2}\ \mathbf{C_T}\ t_1\ t_2.$$
$$(\mathbf{C_T}\,[\mathsf{P_1}\!\downarrow] \rightsquigarrow t_1 \wedge \mathbf{C_T}\,[\mathsf{P_2}\!\downarrow] \rightsquigarrow t_2) \Rightarrow$$
$$\exists \mathsf{C_S}.\, (\mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow t_1 \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow t_2)$$

Establishing R2rTC requires constructing a source context $\mathsf{C_S}$ that can simultaneously simulate a given trace of $\mathbf{C_T}\,[\mathsf{P_1}\!\downarrow]$ and a given trace of $\mathbf{C_T}\,[\mathsf{P_2}\!\downarrow]$. R2rTP generalizes from arity 2 to any finite arity K (yielding RKrTP) and the infinite one (yielding RrTP) in the obvious way.

4.3 Robust Relational Safety Preservation (RrSP)

Relational safety properties are a natural generalization of safety and hypersafety properties to multiple programs, and an important subclass of relational trace properties. Several interesting relational trace properties are actually relational safety properties. For instance, if we restrict the earlier relational trace property "$\mathsf{P_1}$ runs faster than $\mathsf{P_2}$ on all inputs" to terminating programs it becomes a relational safety property, characterized by pairs of bad terminating prefixes, where both prefixes have the same input, and the left prefix shows termination no earlier than the right prefix.

Formally, a relation $R \in 2^{(\mathit{Trace}^K)}$ is *K-relational safety* if for every K "bad" traces $(t_1, \ldots, t_K) \notin R$, there exist K "bad" finite prefixes $m_1, \ldots, m_k$ such that $\forall i.\ m_i \leq t_i$, and any K traces $(t'_1, \ldots, t'_K)$ pointwise extending $m_1, \ldots, m_k$ are also not in the relation, i.e., $\forall i.\ m_i \leq t'_i$ implies $(t'_1, \ldots, t'_K) \notin R$. Then, *Robust 2-relational Safety Preservation* (R2rSP) is simply defined by restricting R2rTP to only 2-relational safety properties. The equivalent () property-free characterization for R2rSP is the following:

$$\mathsf{R2rSC} :\ \forall \mathsf{P_1}\ \mathsf{P_2}\ \mathbf{C_T}\ m_1\ m_2.$$
$$(\mathbf{C_T}\,[\mathsf{P_1}\!\downarrow] \rightsquigarrow m_1 \wedge \mathbf{C_T}\,[\mathsf{P_2}\!\downarrow] \rightsquigarrow m_2) \Rightarrow$$
$$\exists \mathsf{C_S}.\, (\mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow m_1 \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow m_2)$$

The only difference from the stronger R2rTC (§4.2) is between considering full traces and only finite prefixes. Again, R2rSP generalizes to any finite arity K (yielding RKrSP) and the infinite one (yielding RrSP) in the obvious way.

4.4 Separation Between Relational and Non-Relational

Relational (hyper)properties (§4.1, §4.2) and hyperproperties (§3) are different but both have a "relational" nature: relational (hyper)properties are relations on the behaviors or traces of multiple programs, while hyperproperties are relations on multiple traces of the same program. So one may wonder whether there is any case in which the *robust preservation* of a class of relational (hyper)properties is equivalent to that of a class of hyperproperties. Could a compiler that robustly preserves all hyperproperties (RHP, §3.1) also robustly preserves at least some class of 2-relational (hyper)properties? In §4.5 we show special cases in which this is indeed the case, while here we now show that *in general* RHP does not imply the robust preservation of any subclass of relational properties that we have described so far (except, of course, relational properties of arity 1, that are just hyperproperties). Since RHP is the strongest non-relational robust preservation criterion that we study, this also means that no non-relational robust preservation criterion implies any relational robust preservation criterion in Figure 1. So, all edges from relational to non-relational criteria in Figure 1 are strict implications.

To prove this, we build a compilation chain satisfying RHP, but not R2rSP, the weakest relational criterion in Figure 1.

Theorem 4.1. $\mathsf{RHP} \not\Rightarrow \mathsf{R2rSP}$

Proof sketch. Consider a source language that lacks code introspection, and a target language that is exactly the same, but additionally has a primitive with which the context can read the code of the compiled program as data [69]. Consider the trivial compiler that is syntactically the identity. It is clear that this compiler satisfies RHP since the added operation of code introspection offers no advantage to the context when we consider properties of a single program, as is the case in RHP. More precisely, in establishing RHC, the property-free characterization of RHP, given a target context $\mathbf{C_T}$ and a program P, we can construct a simulating source context $\mathsf{C_S}$ by modifying $\mathbf{C_T}$ to hard-code P wherever $\mathbf{C_T}$ performs code introspection. This works as $\mathsf{C_S}$ can depend on P in RHC.

Now consider two programs that differ only in some dead code, that both read a value from the context and write it back verbatim to the output. These two programs satisfy the relational safety property "the outputs of the two programs are equal" in any *source* context. However, there is a trivial *target* context that causes the compiled programs to break this relational property. This context reads the code of the program it is linked to, and provides 1 as input if it happens to be the first of our two programs and 2 otherwise. Consequently, in this target context, the two programs produce outputs 1 and 2 and do not have this relational safety property in all contexts. Hence, this compiler does not satisfy R2rSP. Technically, the trick of hard-coding the program in $\mathsf{C_S}$ no longer works since there are two different programs here. □

This proof provides a fundamental insight: To robustly preserve any subclass of relational (hyper)properties, compilation must ensure that target contexts cannot learn anything about the *syntactic program* they interact with beyond what source contexts can also learn. When the target language is low-level, hiding code attributes can be difficult: it may require padding the code segment of the compiled program to a fixed size, and cleaning or hiding any code-layout-dependent data like code pointers from memory and registers when passing control to the context. These complex protections are not necessary for any non-relational preservation criteria (even RHP), but are already known to be necessary for fully abstract compilation to low-level code [44, 46, 59]. They can also be trivially circumvented if the context has access to any side-channels, e.g., it can measure time via a different thread. In fact, in such settings trying to hide the source code can be seen as a hopeless attempt at "security through obscurity", which is widely rejected by cryptographers since the early days [47].

4.5 Composing Contexts Using Full Reflection or Internal Nondeterminism in the Source Language

The proof of the previous separation theorem strongly relies on the absence of code introspection in the source language. However, if source contexts can obtain *complete* intrinsic information about the programs they are linked with, then RHP implies R2rHP. Such "full reflection" facilities are available in languages such as Lisp [69] and Smalltalk. For proving this collapse we inspect the alternative characterizations, RHC and R2rHC. The main difference between these two criteria, as explained in §4.1, is that the source context $\mathsf{C_S}$ obtained by R2rHC depends on two, possibly distinct programs $\mathsf{P_1}$ and $\mathsf{P_2}$ and a target context $\mathbf{C_T}$, while every possible source context obtained by RHC depends on one single program. Hence, by applying RHC once for $\mathsf{P_1}$ and once for $\mathsf{P_2}$, with the same context $\mathbf{C_T}$, we obtain two source contexts $\mathsf{C_{S_1}}$ and $\mathsf{C_{S_2}}$ that are a priori unrelated. Without further hypotheses, one cannot show R2rHC. However, with full reflection we can define a source context $\mathsf{C'_S}$ that behaves exactly like $\mathsf{C_{S_1}}$ when linked with $\mathsf{P_1}$, and like $\mathsf{C_{S_2}}$ otherwise. We can use this construction to show not only that RHP implies R2rHP, but also that robust preservation of each class of finite-relational properties collapses to the corresponding hyperproperty-based criterion:

Theorem 4.2. If the source language has full reflection then RHP⇒RKrHP, RSCHP⇒RKrTP, and R2HSP⇒RFrSP.

One may wonder whether some other condition exists that makes robust preservation of relational hyperproperty classes collapse even to the corresponding *trace-property-based* criteria (§2). This is indeed the case when the source language has an *internal nondeterministic choice operator* $\oplus$, such that the behavior of $\mathsf{P_1} \oplus \mathsf{P_2}$ is at least the union of the behaviors of $\mathsf{P_1}$ and $\mathsf{P_2}$. Such an operator is standard in process calculi [65]. To illustrate this we show that RTC implies R2rTC. Note that R2rTC produces a source context $\mathsf{C_S}$ that depends on a target context, two source programs $\mathsf{P_1}$ and $\mathsf{P_2}$ and two, possibly incomparable, traces t_1 and t_2. RTC produces a context depending only on a single trace of a single source program. We can apply RTC twice: once for t_1 and $\mathsf{P_1}$ obtaining $\mathsf{C_{S_1}}$ and once for t_2 and $\mathsf{P_2}$ obtaining $\mathsf{C_{S_2}}$. To prove R2rTC we need to build a source context that over-approximates the behaviors of both $\mathsf{C_{S_1}}$ and $\mathsf{C_{S_2}}$. This context can be $\mathsf{C_{S_1}} \oplus \mathsf{C_{S_2}}$. Hence, in this setting RTC (RTP) implies R2rTC (R2rTP). This result generalizes to any finite arity.

Theorem 4.3. If the source language has an internal nondeterministic choice operator on contexts then RTP ⇒ RKrTP, RSCHP ⇒ RFrSCHP, and RSP ⇒ RFrSP.

Notice that since contexts are finite objects, the techniques above only produce collapses in cases where finitely many source contexts need to be composed. Criteria relying on infinite-arity relations such as RrHP and RrTP are thus not impacted by these collapses. The appendix has more details and collapsed variants of Figure 1.

5 Where Is Full Abstraction?

Full abstraction—the preservation and reflection of observational equivalence—is a well-studied criterion for secure compilation (§7). The security-relevant direction of full abstraction is *Observational Equivalence Preservation* (OEP) [28, 60]:

$$\mathsf{OEP} : \forall \mathsf{P_1}\ \mathsf{P_2}.\ \mathsf{P_1} \approx \mathsf{P_2} \Rightarrow \mathsf{P_1}{\downarrow} \approx \mathsf{P_2}{\downarrow}$$

One natural question is how OEP relates to our criteria of robust preservation.

Here we answer this question for languages without internal nondeterminism. In such *determinate* [33, 51] settings observational equivalence coincides with trace equivalence in all contexts [21, 33] and, hence, OEP coincides with robust trace-equivalence preservation (RTEP). As explained in §4.1, it is obvious that RTEP is an instance of R2rHP, obtained by choosing equality as the relation R. However, for determinate languages with *input totality* [36, 75] (if the program accepts one input value, it has to also accept any other input value) we have proved that even the weaker R2rTP implies RTEP. This proof also requires that if a whole program can produce every finite prefix of an infinite trace then it can also produce the complete trace, but we have showed that this holds for the infinite traces produced in a standard way by *any* determinate small-step semantics. Under these assumptions, we have in fact proved that RTEP follows from the even weaker *Robust 2-relational relaXed safety Preservation* (R2rXP). The class *2-relational relaXed safety* is a variant of *2-relational Safety* from §4.3; with this relaxed variant "bad" prefixes x_1 and x_2 are allowed to end with silent divergence (denoted as *XPref*):

$$R \in \textit{2-relational relaXed safety} \iff \forall (t_1,t_2) \notin R.\ \exists x_1\, x_2 \in \textit{XPref}.\ \forall t'_1 \geq x_1\, t'_2 \geq x_2.\ (t'_1,t'_2) \notin R$$

Theorem 5.1. Assuming a determinate source language and a determinate and input total small-step semantics for the target language, R2rXP ⇒ RTEP.

In the other direction, we adapt an existing counterexample [57] to show that RTEP (and, hence, for determinate languages also OEP) does *not* imply RSP or any of the criteria above it in Figure 1. Fundamentally, RTEP only requires preserving *equivalence* of behavior. Consequently, an RTEP compiler can insert code that violates any security property, as long as it doesn't alter these equivalences [57]. Worse, even when the RTEP compiler is also required to be correct (i.e., TP, SCC, and CCC from §2.1), the compiled program only needs to properly deal with interactions with target contexts that behave like source ones, and can behave insecurely when interacting with target contexts that have no source equivalent.

Theorem 5.2. There exists a compiler between two deterministic languages that satisfies RTEP, TP, SCC, and CCC, but that does not satisfy RSP.

Proof. Consider a source language where a partial program receives a natural number or boolean from the context, and produces a number output, which is the only event. We compile to a restricted language that only has numbers by mapping booleans true and false to $\mathbf{0}$ and $\mathbf{1}$ respectively. The compiler's only interesting aspect is that it translates a source function $\mathsf{P} = \mathsf{f(x{:}Bool)} \mapsto \mathsf{e}$ that inputs booleans to $\mathsf{P}\!\downarrow\, = \mathbf{f(x{:}Nat)}{\mapsto}\mathbf{if\ x{<}2\ then\ e}\!\downarrow\ \mathbf{else\ if\ x{<}3\ then\ f(x)\ else\ 42}$. The compiled function checks if its input is a valid boolean ($\mathbf{0}$ or $\mathbf{1}$). If so, it executes $\mathsf{e}\!\downarrow$. Otherwise, it behaves insecurely, silently diverging on input 2 and outputting 42 on inputs 3 or more. This compiler does not satisfy RSP since the source program $\mathsf{f(x{:}Bool){\mapsto}0}$ robustly satisfies the safety property "never output 42", but the program's compilation does not.

On the other hand, it is easy to see that this compiler is correct since a compiled program behaves exactly like its source counterpart on correct inputs. It is also easily seen to satisfy RTEP, since the additional behaviors added by the compiler (silently diverging on input 2 and outputting 42 on inputs 3 or more) are independent of the source code (they only depend on the type), so these cannot be used by any target context to distinguish two compiled programs.

In the appendix, we use the same counterexample compilation chain to also show that RTEP does not imply the robust preservation of (our variant of) liveness properties. We also use a simple extension of this compilation chain to show that RTEP does not imply RTINIP either. The idea is similar: we add a secret external input to the languages and when receiving an out of bounds argument the compiled code simply leaks the secret input, which breaks RTINIP, but not RTEP. □

6 Proof Techniques for RrHP and RFrXP

This section demonstrates that the criteria we introduce can be proved by adapting existing back-translation techniques. We introduce a statically typed source language and a similar dynamically typed target one (§6.1), as well as a simple translation between the two (§6.2). We then describe the essence of two very different secure compilation proofs for this compilation chain, both based on techniques originally developed for showing fully abstract compilation. The first proof shows (a typed variant of) RrHP (§6.3), the strongest criterion from Figure 1, using a *context-based* back-translation, which provides a "universal embedding" of a target context into a source context [56]. The second proof shows a slightly weaker criterion, *Robust Finite-relational relaXed safety Preservation* (RFrXP; §6.4), but which is still very useful, as it implies robust preservation of arbitrary safety and hypersafety properties as well as RTEP. This second proof relies on a *trace-based* back-translation [43, 59], extended to produce a context from *a finite set* of finite execution prefixes. These finiteness restrictions are offset by a more generic proof technique that only depends on the context-program interaction (e.g., calls and returns), while ignoring all other language details. For space reasons, we leave the details of the proofs for the appendix.

6.1 Source and Target Languages

The two languages we consider are simple first-order languages with named procedures and boolean and natural values. The source language L^τ is typed while the target language $\mathbf{L^u}$ is untyped. A program in either language is a collection of function definitions, each function body is a pure expression that can perform comparison and natural operations ($\oplus$), conditional branching, recursive calls, and use let-in bindings. Expressions can also read naturals from the environment and write naturals to the environment, both of which generate trace events. $\mathbf{L^u}$ has all the features of L^τ and adds a primitive $\mathbf{e\ has}\ \tau$ to dynamically check whether an expression $\mathbf{e}$ has type τ. A context $\mathbb{C}$ can call functions and perform general computation on the returned values, but it cannot directly generate *read* and *write e* events, as those are security-sensitive. Since contexts are single expressions, we disallow callbacks from the program to the context: thus calls go from context to program, and returns from program to context.

$$
\begin{aligned}
&\textit{Programs}\ \mathsf{P} ::= \bar{\mathsf{I}}; \bar{\mathsf{F}} \qquad \textit{Contexts}\ \mathbb{C} ::= \mathsf{e}\\
&\textit{Types}\ \tau ::= \mathsf{Bool} \mid \mathsf{Nat} \qquad \textit{Interfaces}\ \mathsf{I} ::= \mathsf{f} : \tau \to \tau\\
&\textit{Functions}\ \mathsf{F} ::= \mathsf{f}(\mathsf{x} : \tau) : \tau \mapsto \mathsf{ret\ e}\\
&\textit{Expressions}\ \mathsf{e} ::= \mathsf{x} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{n} \in \mathbb{N} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{e} \geq \mathsf{e}\\
&\qquad \mid \mathsf{let\ x} : \tau = \mathsf{e\ in\ e} \mid \mathsf{if\ e\ then\ e\ else\ e}\\
&\qquad \mid \mathsf{call\ f\ e} \mid \mathsf{read} \mid \mathsf{write\ e} \mid \mathsf{fail}\\
&\textit{Programs}\ \mathbf{P} ::= \bar{\mathbf{I}}; \bar{\mathbf{F}} \qquad \textit{Contexts}\ \mathbb{C} ::= \mathbf{e}\\
&\textit{Types}\ \tau ::= \mathbf{Bool} \mid \mathbf{Nat} \qquad \textit{Interfaces}\ \mathbf{I} ::= \mathbf{f}\\
&\textit{Functions}\ \mathbf{F} ::= \mathbf{f(x)} \mapsto \mathbf{ret\ e}\\
&\textit{Expressions}\ \mathbf{e} ::= \mathbf{x} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{n} \in \mathbb{N} \mid \mathbf{e} \oplus \mathbf{e} \mid \mathbf{e} \geq \mathbf{e}\\
&\qquad \mid \mathbf{let\ x = e\ in\ e} \mid \mathbf{if\ e\ then\ e\ else\ e}\\
&\qquad \mid \mathbf{call\ f\ e} \mid \mathbf{read} \mid \mathbf{write\ e} \mid \mathbf{fail} \mid \mathbf{e\ has}\ \tau\\
&\textit{Labels}\ \lambda ::= \epsilon \mid \alpha\\
&\textit{Actions}\ \alpha ::= \mathtt{read}\ n \mid \mathtt{write}\ n \mid \Downarrow \mid \Uparrow \mid \bot
\end{aligned}
$$

Each language has a standard small-step operational semantics (omitted for brevity), as well as a big-step trace semantics ($\Omega \rightsquigarrow \overline{\alpha}$, as in previous sections). The initial state of a program P plugged into a context $\mathbb{C}$ is denoted as $P \triangleright \mathbb{C}$ and the behavior of such a program is the set of traces that can be produced by the semantics:

$$\mathtt{Behav}\,(\mathbb{C}[P]) = \{\overline{\alpha} \ \mid\ P \triangleright \mathbb{C} \rightsquigarrow \overline{\alpha}\}$$

6.2 Compiler

The compiler $\cdot\downarrow$ takes programs of L^τ and generates programs of $\mathbf{L^u}$, by replacing static type annotations with dynamic type checks of function arguments upon function invocation:

$$\mathsf{I}_1,\cdots,\mathsf{I}_m;\mathsf{F}_1,\cdots,\mathsf{F}_n\downarrow = \mathsf{I}_1\downarrow,\cdots,\mathsf{I}_m\downarrow;\mathsf{F}_1\downarrow,\cdots,\mathsf{F}_n\downarrow$$

$$\mathsf{f}:\tau\to\tau'\downarrow = \mathbf{f}$$

$$\left.\begin{array}{l}\mathsf{f}(\mathsf{x}:\tau):\tau'\mapsto\\ \mathsf{ret\ e}\end{array}\right\downarrow = \left(\begin{array}{r}\mathbf{f(x)\mapsto ret\ \ if\ x\ has\ }\tau\downarrow\\ \mathbf{then\ e\downarrow\ else\ fail}\end{array}\right)$$

$$\mathsf{Nat}\downarrow = \mathbf{Nat} \qquad \mathsf{Bool}\downarrow = \mathbf{Bool}$$

$$\mathsf{true}\downarrow = \mathbf{true} \qquad \mathsf{false}\downarrow = \mathbf{false}$$

$$\mathsf{n}\downarrow = \mathbf{n} \qquad \mathsf{x}\downarrow = \mathbf{x}$$

$$\mathsf{e}\oplus\mathsf{e}'\downarrow = \mathsf{e}\downarrow\oplus\mathsf{e}'\downarrow \qquad \mathsf{e}\geq\mathsf{e}'\downarrow = \mathsf{e}\downarrow\geq\mathsf{e}'\downarrow$$

$$\mathsf{read}\downarrow = \mathbf{read} \qquad \mathsf{write\ e}\downarrow = \mathbf{write}\ \mathsf{e}\downarrow$$

$$\mathsf{call\ f\ e}\downarrow = \mathbf{call\ f}\ \mathsf{e}\downarrow$$

$$\left.\begin{array}{l}\mathsf{let\ x}:\tau\mathsf{=e}\\ \mathsf{in\ e'}\end{array}\right\downarrow = \begin{array}{l}\mathbf{let\ x=}\mathsf{e}\downarrow\\ \mathbf{in}\ \mathsf{e}'\downarrow\end{array} \qquad \left.\begin{array}{l}\mathsf{if\ e\ then\ e'}\\ \mathsf{else\ e''}\end{array}\right\downarrow = \begin{array}{l}\mathbf{if}\ \mathsf{e}\downarrow\ \mathbf{then}\ \mathsf{e}'\downarrow\\ \mathbf{else}\ \mathsf{e}''\downarrow\end{array}$$

6.3 Proof of RrHP by Context-Based Back-Translation

To prove that $\cdot\downarrow$ attains RrHP, we need a way to back-translate target contexts into source contexts. To this end we use a universal embedding, a technique previously proposed for proving fully abstract compilation [56]. The back-translation needs to generate a source context that respects source-level constraints; in this case, the resulting source context must be well-typed. To ensure this, we use Nat as an *universal back-translation type* in the produced source contexts. The intuition of the back-translation is that it will encode $\mathbf{true}$ as $\mathsf{0}$, $\mathbf{false}$ as $\mathsf{1}$ and an arbitrary natural number $\mathbf{n}$ as $\mathsf{n+2}$. Based on this encoding, we translate values between regular source types and the back-translation type. Specifically, we define the following shorthand for the back-translation: $\mathsf{inject}_\tau(\mathsf{e})$ takes an expression e of type τ and returns an expression of back-translation type; $\mathsf{extract}_\tau(\mathsf{e})$ takes an expression e of the back-translation type and returns an expression of type τ.

$$\mathsf{inject}_{\mathsf{Nat}}(\mathsf{e}) = \mathsf{e+2}$$

$$\mathsf{inject}_{\mathsf{Bool}}(\mathsf{e}) = \mathsf{if\ e\ then\ 1\ else\ 0}$$

$$\mathsf{extract}_{\mathsf{Nat}}(\mathsf{e}) = \left(\mathsf{let\ x=e\ in\ if\ x\geq 2\ then\ x-2\ else\ fail}\right)$$

$$\mathsf{extract}_{\mathsf{Bool}}(\mathsf{e}) = \left(\begin{array}{l}\mathsf{let\ x=e\ in\ if\ x\geq 2\ then\ fail}\\ \quad\mathsf{else\ if\ x+1\geq 2\ then\ true\ else\ false}\end{array}\right)$$

$\mathsf{inject}_\tau(\mathsf{e})$ never incurs runtime errors, but $\mathsf{extract}_\tau(\mathsf{e})$ may. This mimics the ability of target contexts to write ill-typed code (e.g., $\mathbf{3+true}$) which we must be able to back-translate and whose semantics we must preserve (see Example 6.1).

Concretely, the back-translation is defined inductively on the structure of target contexts:

$$\mathbf{true}\uparrow = \mathsf{1} \qquad \mathbf{false}\uparrow = \mathsf{0} \qquad \mathbf{n}\uparrow = \mathsf{n+2} \qquad \mathbf{x}\uparrow = \mathsf{x}$$

$$\mathbf{e}\geq\mathbf{e'}\uparrow = \begin{array}{l}\mathsf{let\ x1:Nat=extract_{Nat}(}\mathbf{e}\uparrow\mathsf{)}\\ \quad\mathsf{in\ let\ x2:Nat=extract_{Nat}(}\mathbf{e'}\uparrow\mathsf{)}\\ \quad\mathsf{in\ inject_{Bool}(x1\geq x2)}\end{array}$$

$$\mathbf{e}\oplus\mathbf{e'}\uparrow = \begin{array}{l}\mathsf{let\ x1:Nat=extract_{Nat}(}\mathbf{e}\uparrow\mathsf{)}\\ \quad\mathsf{in\ let\ x2:Nat=extract_{Nat}(}\mathbf{e'}\uparrow\mathsf{)}\\ \quad\mathsf{in\ inject_{Nat}(x1\oplus x2)}\end{array}$$

$$\mathbf{let\ x=e\ in\ e'}\uparrow = \mathsf{let\ x:Nat=}\mathbf{e}\uparrow\ \mathsf{in}\ \mathbf{e'}\uparrow$$

$$\left(\begin{array}{l}\mathbf{if\ e\ then}\\ \mathbf{e'\ else\ e''}\end{array}\right)\uparrow = \mathsf{if\ extract_{Bool}(}\mathbf{e}\uparrow\mathsf{)\ then}\ \mathbf{e'}\uparrow\ \mathsf{else}\ \mathbf{e''}\uparrow$$

$$\mathbf{e\ has\ Bool}\uparrow = \mathsf{let\ x:Nat=}\mathbf{e}\uparrow\ \mathsf{in\ if\ x\geq 2\ then\ 0\ else\ 1}$$

$$\mathbf{e\ has\ Nat}\uparrow = \mathsf{let\ x:Nat=}\mathbf{e}\uparrow\ \mathsf{in\ if\ x\geq 2\ then\ 1\ else\ 0}$$

$$\mathbf{call\ f\ e}\uparrow = \mathsf{inject}_{\tau'}(\mathsf{call\ f\ extract}_\tau(\mathbf{e}\uparrow))$$
$$\text{if } \mathsf{f}:\tau\to\tau'\in\bar{\mathsf{I}}$$

$$\mathbf{fail}\uparrow = \mathsf{fail}$$

Example 6.1 (Back-Translation)**.** Through the back-translation of two simple target contexts we explain why $\cdot\uparrow$ is correct and why it needs $\mathsf{inject.}$ and $\mathsf{extract.}$.

Consider the context $\mathbb{C}_1 = \mathbf{3*5}$, which reduces to $\mathbf{15}$ irrespective of the program it links against. The back-translation must intuitively ensure that $\mathbb{C}_1\uparrow$ reduces to $\mathsf{17}$, which is the back-translation of $\mathbf{15}$. If we unfold the definition of $\mathbb{C}_1\uparrow$ we have the following (given that $\mathbf{3}\uparrow=\mathsf{5}$ and $\mathbf{5}\uparrow=\mathsf{7}$):

$$\begin{array}{l}\mathsf{let\ x1:Nat=extract_{Nat}(5)}\\ \quad\mathsf{in\ let\ x2:Nat=extract_{Nat}(7)\ in\ inject_{Nat}(x1*x2)}\end{array}$$

By examining the code of $\mathsf{extract_{Nat}}$ we see that in both cases it will just perform a subtraction by 2, turning $\mathsf{5}$ and $\mathsf{7}$ respectively into $\mathsf{3}$ and $\mathsf{5}$. So after some reduction steps we arrive at the following term: $\mathsf{inject_{Nat}(3*5)}$. The inner multiplication then returns $\mathsf{15}$ and its injection returns $\mathsf{17}$, which is also the result of $\mathbf{15}\uparrow$.

Let us now consider a different context, $\mathbb{C}_2 = \mathbf{false+3}$. We know that no matter what program links against it, it will reduce to $\mathbf{fail}$. Its statically well-typed back-translation is:

$$\begin{array}{l}\mathsf{let\ x1:Nat=extract_{Nat}(0)}\\ \quad\mathsf{in\ let\ x2:Nat=extract_{Nat}(7)\ in\ inject_{Nat}(x1*x2)}\end{array}$$

By looking at its code we can see that the execution of $\mathsf{extract_{Nat}(0)}$ will indeed result in fail, which is what we want and expect, as that is precisely the back-translation of $\mathbf{fail}$. ⊡

The RrHP proof for this compilation chain uses a simple logical relation that includes cases for both terms of source type (intuitively used for compiler correctness) and for terms of back-translation type [28, 56].

6.4 Proof of RFrXP by Trace-Based Back-Translation

Proving that this simple compilation chain attains RFrXC does not require back-translating a target context, as we only need to build a source context that can reproduce a finite set of finite trace prefixes, but that is not necessarily equivalent to the original target context. We describe this back-translation on an example leaving again details to the online appendix.

Example 6.2 (Back-Translation of Traces)**.** Consider the following two programs:

$$\mathsf{P_1} = (\mathsf{f(x{:}Nat) : Nat \mapsto ret\ x}, \mathsf{g(x{:}Nat) : Bool \mapsto ret\ true})$$
$$\mathsf{P_2} = (\mathsf{f(x{:}Nat) : Nat \mapsto ret\ read}, \mathsf{g(x{:}Nat) : Bool \mapsto ret\ true})$$

Their compiled counterparts are almost identical, with the only addition of dynamic type checks on function arguments:

$$\mathsf{P_1}{\downarrow} = \mathbf{f(x) \mapsto ret\ (if\ x\ has\ Nat\ then\ x\ else\ fail)},$$
$$\mathbf{g(x) \mapsto ret\ (if\ x\ has\ Nat\ then\ true\ else\ fail)}$$
$$\mathsf{P_2}{\downarrow} = \mathbf{f(x) \mapsto ret\ (if\ x\ has\ Nat\ then\ read\ else\ fail)},$$
$$\mathbf{g(x) \mapsto ret\ (if\ x\ has\ Nat\ then\ true\ else\ fail)}$$

Now, consider the following target context:

$$\mathbb{C} = \mathbf{let\ x1{=}call\ f\ 5}$$
$$\mathbf{in\ if\ x1 \geq 5\ then\ call\ g\ (x1)\ else\ call\ g\ (false)}$$

The two programs plugged into this context can generate (at least) the following traces (where $\Downarrow$ indicates termination and $\bot$ indicates failure):

$$\mathbb{C}[\mathsf{P_1}{\downarrow}] \rightsquigarrow \Downarrow \quad \mathbb{C}[\mathsf{P_2}{\downarrow}] \rightsquigarrow \mathtt{read}\ 5; \Downarrow \quad \mathbb{C}[\mathsf{P_2}{\downarrow}] \rightsquigarrow \mathtt{read}\ 0; \bot$$

In the execution of $\mathbb{C}[\mathsf{P_1}{\downarrow}]$, the program executes completely and terminates, producing no side effects. In the first execution of $\mathbb{C}[\mathsf{P_2}{\downarrow}]$, the program reads 5, and the *then* branch of the context's conditional is executed. In the second execution of $\mathbb{C}[\mathsf{P_2}{\downarrow}]$, the program reads 0, the *else* branch of the context's conditional is executed and the program fails in $\mathbf{g}$ after detecting a type error.

These traces alone are not enough to construct a source context since they do not record information about the control flow of program executions, specifically on which function produces which input or output. To recover this information we enrich execution prefixes with information about calls (from context to program) and returns (from program to context). The enriched rules on calls and returns now generate events to model these control flows. If a call or return occurs internally within the program, no trace event is generated since they are not relevant for back-translating the context. The revised semantics is almost identical to the original, and allows exactly the same program executions, only producing more informative traces. Hence, the original execution can be enriched in a valid way for the new semantics.

$$\textit{Labels}\ \lambda ::= \cdots \mid \beta \quad \textit{Interactions}\ \beta ::= \mathtt{call}\ f\ v \mid \mathtt{ret}\ v$$

The traces produced by the compiled programs plugged into the context become:

$$\mathbb{C}[\mathsf{P_1}{\downarrow}] \rightsquigarrow \mathtt{call}\ f\ 5; \quad \mathtt{ret}\ 5; \mathtt{call}\ g\ 5; \mathtt{ret}\ \mathit{true}; \Downarrow$$
$$\mathbb{C}[\mathsf{P_2}{\downarrow}] \rightsquigarrow \mathtt{call}\ f\ 5; \mathtt{read}\ 5; \mathtt{ret}\ 5; \mathtt{call}\ g\ 5; \mathtt{ret}\ \mathit{true}; \Downarrow$$
$$\mathbb{C}[\mathsf{P_2}{\downarrow}] \rightsquigarrow \mathtt{call}\ f\ 5; \mathtt{read}\ 0; \mathtt{ret}\ 0; \mathtt{call}\ g\ \mathit{false}; \bot$$

In our languages, reads and writes can only be performed by programs, while the context only performs a sequence of calls to the program, possibly performing some computation and branching on return values. Thus, the role of the back-translated source is to perform the appropriate calls to the program, depending of the values returned. The inner workings of the programs, that is inputs, outputs, and internal calls and returns, are not a concern of the back-translation and are obtained through compiler correctness. Furthermore, the context is shared by all executions, but each execution has its own program. Hence, since I/O occurs only in the program, the only source of variation among all executions come from the program.

From this, one can conclude that the context is a deterministic expression, calling the program, and branching on the returned values. This can be seen in the way traces are organized: ignoring the I/O, the traces form a tree (Figure 2, on the left). This tree can be translated to a source context using nested conditionals as depicted below (Figure 2, on the right, dotted lines indicated what the back-translation generates for each action in the tree). When additional branches are missing (e.g., there is no third trace that analyzes the first return or no second trace that analyses the second return on the left execution), the back-translation inserts fail in the code – they are dead code branches (marked with a **).

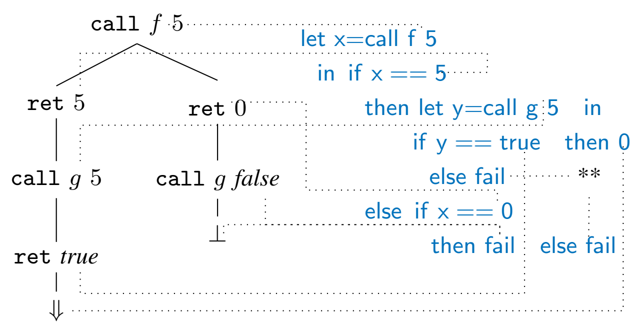

Fig. 2: Example of a back-translation of traces.

To prove RFrXP we show correctness of the back-translation, which ensures that the back-translated source context produces exactly the original non-informative traces. This is, however, not completely true of informative traces (that track calls and returns). Since calling g with a boolean is ill-typed, our back-translation shifts the failure from the program to the context, so the picture links $\mathtt{call}\ g\ \mathit{false}$ action to a fail. The call is never executed at the source level. ⊡

7 Related Work

Full Abstraction was originally used as a criterion for secure compilation in the seminal work of Abadi [1] and has since received a lot of attention [60]. Abadi [1] and, later, Kennedy [46] identified failures of full abstraction in the Java to JVM and C# to CIL compilers, some of which were fixed, but also others for which fixing was deemed too costly compared to the perceived practical security gain. Abadi et al. [3] proved full abstraction of secure channel implementations using cryptography, but to prevent network traffic attacks they had to introduce noise in their translation, which in practice would consume network bandwidth. Ahmed et al. [6, 7, 56] proved the full abstraction of type-preserving compiler passes for simple functional languages. Abadi and Plotkin [2] and Jagadeesan et al. [42] expressed the protection provided by address space layout randomization as a probabilistic variant of full abstraction. Fournet et al. [37] devised a fully abstract compiler from a subset of ML to JavaScript. Patrignani et al. [59] studied fully abstract compilation to machine code, starting from single modules written in simple, idealized object-oriented and functional languages and targeting a hardware isolation mechanism similar to Intel's SGX [41].

Until recently, most formal work on secure interoperability with linked target code was focused only on fully abstract compilation. The goal of our work is to explore a diverse set of secure compilation criteria, some of them formally stronger than (the interesting direction of) full abstraction at least in various determinate settings, and thus potentially harder to achieve and prove, some of them apparently easier to achieve and prove than full abstraction, but most of them not directly comparable to full abstraction. This exploration clarifies the trade-off between security guarantees and efficient enforcement for secure compilation: On one extreme, RTP robustly preserves only trace properties, but does not require enforcing confidentiality; on the other extreme, robustly preserving relational properties gives very strong guarantees, but requires enforcing that both the private data and the code of a program remain hidden from the context, which is often much harder to achieve. The best criterion to apply depends on the application domain, but our framework can be used to address interesting design questions such as the following: *(1) What secure compilation criterion, when violated, would the developers of practical compilers be willing to fix at least in principle?* The work of Kennedy [46] indicates that fully abstract compilation is not such a good answer to this question, and we wonder whether RTP or RHP could be better answers. *(2) What secure compilation criterion would the translations of Abadi et al. [3] still satisfy if they did not introduce (inefficient) noise to prevent network traffic analysis?* Abadi et al. [3] explicitly leave this problem open in their paper, and we believe one answer could be RTP, since it does not require preserving any confidentiality.

We also hope that our work can help eliminate common misconceptions about the security guarantees provided (or not) by full abstraction. For instance, Fournet et al. [37] illustrate the difficulty of achieving security for JavaScript code using a simple example policy that (1) restricts message sending to only correct URLs and (2) prevents leaking certain secret data. Then they go on to prove full abstraction apparently in the hope of preventing contexts from violating such policies. However, part (1) of this policy is a safety property and part (2) is hypersafety, and as we showed in §4.5 fully abstract compilation does not imply the robust preservation of such properties. In contrast, proving RHSP would directly imply this, without putting any artificial restrictions on code introspection, which are unnecessarily required by full abstraction. Unfortunately, this is not the only work in the literature that uses full abstraction even when it is not the right hammer.

Development of RSP Two pieces of concurrent work have examined more carefully how to attain and prove one of the weakest of our criteria, RSP (§2.2). Patrignani and Garg [58] show RSP for compilers from simple sequential and concurrent languages to capabilities [74]. They observe that if the source language has a verification system for robust safety and compilation is limited to verified programs, then RSP can be established without directly resorting to back-translation. (This observation has also been made independently by Dave Swasey in private communication to us.) Abate et al. [4] aim at devising secure compilation chains for protecting mutually distrustful components written in an unsafe language like C. They show that by moving away from the full abstraction variant used in earlier work [44] to a variant of our RSP criterion from §2.2, they can support a more realistic model of dynamic component compromise, while at the same time obtaining a criterion that is easier to achieve and prove than full abstraction.

Hypersafety Preservation The high-level idea of specifying secure compilation as the preservation of properties and hyperproperties goes back to the work of Patrignani and Garg [57]. However, that work's technical development is limited to one criterion—the preservation of finite prefixes of program traces by compilation. Superficially, this is similar to one of our criteria, RHSP, but there are several differences even from RHSP. First, Patrignani and Garg [57] do not consider adversarial contexts explicitly. This might suffice for their setting of closed reactive programs, where traces are inherently fully abstract (so considering the adversarial context is irrelevant), but not in general. Second, they are interested in designing a criterion that accommodates specific fail-safe like mechanisms for low-level enforcement, so the preservation of hypersafety properties is not perfect, and one has to show, for every relevant property, that the criterion is meaningful. However, Patrignani and Garg [57] consider translations of trace symbols induced by compilation, an extension that would also be interesting for our criteria (§8).

Proof techniques New et al. [56] present a back-translation technique based on a universal type embedding in the source for the purpose of proving full abstraction of translations from typed to untyped languages. In §6.3 we adapted the same technique to show RrHP for a simple translation from a statically typed to a dynamically typed language with first-order functions and I/O. Devriese et al. [28] show that even when a precise universal type does not exist in the source, one can use an approximate embedding that only works for a certain number of execution steps. They illustrate such an approximate back-translation by proving full abstraction for a compiler from the simply-typed to the untyped λ-calculus.

Jeffrey and Rathke [43] introduced a "trace-based" back-translation technique. They were interested in proving full abstraction for so-called trace semantics. This technique was then adapted to show full abstraction of compilation chains to low-level target languages [59]. In §6.4, we showed how these trace-based techniques can be extended to prove all the criteria below RFrXP in Figure 1, which includes robust preservation of safety, of noninterference, and in a determinate setting also of observational equivalence.

While many other proof techniques have been previously proposed [2, 3, 7, 37, 42], proofs of full abstraction remain notoriously difficult, even for simple translations, with apparently simple conjectures surviving for decades before being finally settled [29]. It will be interesting to investigate which existing full abstraction techniques can be repurposed to show the stronger criteria from Figure 1. For instance, it will be interesting to determine the strongest criterion from Figure 1 for which an approximate back-translation [28] can be used.

Source-level verification of robust satisfaction While this paper studies the *preservation* of robust properties in compilation chains, formally verifying that a partial source program robustly satisfies a specification is a challenging problem too. So far, most of the research has focused on techniques for proving observational equivalence [25, 43] or trace equivalence [12, 21]. Robust satisfaction of trace properties has been model checked for systems modeled by nondeterministic Moore machines and properties specified by branching temporal logic [49]. Robust safety, the robust satisfaction of safety properties, was studied for the analysis of security protocols [39], and more recently for compositional verification [71]. Verifying the *robust* satisfaction of relational hyperproperties beyond observational equivalence and trace equivalence seems to be an open research problem. For addressing it, one can hopefully take inspiration in extensions of relational Hoare logic [15] for dealing with cryptographic adversaries represented as procedures parameterized by oracles [13].

Other Kinds of Secure Compilation In this paper we investigated the various kinds of security guarantees one can obtain from a compilation chain that protects the compiled program against linked adversarial low-level code. While this is an instance of *secure compilation* [8], this emerging area is much broader. Since there are many ways in which a compilation chain can be "more secure", there are also many different notions of secure compilation, with different security goals and attacker models. A class secure compilation chains is aimed at providing a "safer" semantics for unsafe low-level languages like C and C++, for instance ensuring memory safety [22, 32, 54]. Other secure compilation work is targeted at closing down side-channels: for instance by preserving the secret independence guarantees of the source code [14], or making sure that the code erasing secrets is not simply optimized away by the unaware compilers [17, 27, 30, 67]. Closer to our work is the work on building compartmentalizing compilation chains [4, 18, 40, 74] for unsafe languages like C and C++. In particular, as mentioned above, Abate et al. [4] have recently showed how RSP can be extended to express the security guarantees obtained by protecting mutually distrustful components against each other.

8 Conclusion and Future Work

This paper proposes a foundation for secure interoperability with linked target code by exploring many different criteria based on robust property preservation (Figure 1). Yet the road to building *practical* secure compilation chains achieving any of these criteria remains long and challenging. Even for RSP, scaling up to realistic programming languages and efficiently enforcing protection of the compiled program without restrictions on the linked context is challenging [4, 58]. For R2HSP the problem is even harder, because one also needs to protect the secrecy of the program's data, which is especially challenging in a realistic model in which the context can observe side-channels like timing. Here, an RTINIP-like property might be the best one can hope for in practice.

In this paper we assumed for simplicity that traces are exactly the same in both the source and target language, and while this assumption is currently true for other work like CompCert [51] as well, it is a restriction nonetheless. We plan to lift this restriction in the future.

Acknowledgments

We are grateful to Akram El-Korashy, Arthur Azevedo de Amorim, Ştefan Ciobâcă, Dominique Devriese, Guido Martínez, Marco Stronati, Dave Swasey, Éric Tanter, and the anonymous reviewers for their valuable feedback and in many cases also for participating in various discussions. This work was in part supported by the ERC under ERC Starting Grant SECOMP (715753), by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), and by DARPA grant SSITH/HOPE (FA8650-15-C-7558).

References

[1] M. Abadi. Protection in programming-language translations. *Secure Internet Programming*. 1999.
[2] M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM TISSEC*, 15(2), 2012.
[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1), 2002.
[4] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hriţcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. *CCS*. 2018.
[5] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. *CSF*. 2012.
[6] A. Ahmed. Verified compilers for a multi-language world. *SNAPL*. 2015.
[7] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. *ICFP*. 2011.
[8] A. Ahmed, D. Garg, C. Hriţcu, and F. Piessens. Secure Compilation (Dagstuhl Seminar 18201). *Dagstuhl Reports*, 8(5), 2018.
[9] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. *CCS*. 2017.
[10] B. Alpern and F. B. Schneider. Defining liveness. *IPL*, 21(4), 1985.
[11] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. *ESORICS*. 2008.
[12] D. Baelde, S. Delaune, and L. Hirschi. A reduced semantics for deciding trace equivalence. *LMCS*, 13(2), 2017.
[13] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. EasyCrypt: A tutorial. In *FOSAD 2012/2013*. 2013.
[14] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". *CSF*. 2018.
[15] N. Benton. Simple relational correctness proofs for static analyses and program transformations. *POPL*. 2004.
[16] D. Bernstein. Writing high-speed software. http://cr.yp.to/qhasm.html.

[17] F. Besson, A. Dang, and T. Jensen. Securing compilation against memory probing. *PLAS*. 2018.
[18] F. Besson, S. Blazy, A. Dang, T. Jensen, and P. Wilke. Compiling sandboxes: Formally verified software fault isolation. In *ESOP*, 2019.
[19] B. Beurdouche, K. Bhargavan, F. Kiefer, J. Protzenko, E. Rescorla, T. Taubert, M. Thomson, and J.-K. Zinzindohoue. HACL* in Mozilla Firefox: Formal methods and high assurance applications for the web. Real World Crypto Symposium, 2018.
[20] E. Cecchetti, A. C. Myers, and O. Arden. Nonmalleable information flow control. *CCS*, 2017.
[21] V. Cheval, V. Cortier, and S. Delaune. Deciding equivalence-based properties using constraint solving. *TCS*, 492, 2013.
[22] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. *ASPLOS*. 2015.
[23] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. D. Son, M. Roe, S. W. Moore, P. G. Neumann, B. Laurie, and R. N. M. Watson. CHERI JNI: sinking the Java security model into the C. *ASPLOS*. 2017.
[24] M. R. Clarkson and F. B. Schneider. Hyperproperties. *JCS*, 18(6), 2010.
[25] S. Delaune and L. Hirschi. A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. *JLAMP*, 87, 2017.
[26] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *S&P*. 2017.
[27] C. Deng and K. S. Namjoshi. Securing a compiler transformation. *FMSD*, 53(2), 2018.
[28] D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. *POPL*, 2016.
[29] D. Devriese, M. Patrignani, and F. Piessens. Parametricity versus the universal type. *PACMPL*, 2(POPL), 2018.
[30] V. D'Silva, M. Payer, and D. X. Song. The correctness-security gap in compiler optimization. *S&P Workshops*. 2015.
[31] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. *IMC*. 2014.
[32] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. Checked C: Making C safe by extension. SecDev, 2018.
[33] J. Engelfriet. Determinacy implies (observation equivalence = trace equivalence). *TCS*, 36, 1985.
[34] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. *S&P*, 2019.
[35] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2), 1974.
[36] R. Focardi and R. Gorrieri. A taxonomy of security properties for process algebras. *JCS*, 3(1), 1995.
[37] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. *POPL*. 2013.
[38] J. A. Goguen and J. Meseguer. Security policies and security models. *S&P*, 1982.
[39] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *JCS*, 12(3-4), 2004.
[40] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP. *CCS*. 2015.
[41] Intel. Software guard extensions (SGX) programming reference, 2014.
[42] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. *CSF*. 2011.
[43] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. *ESOP*. 2005.
[44] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. *CSF*, 2016.
[45] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. *POPL*, 2016.
[46] A. Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, 364(3), 2006.
[47] A. Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX, 1883.
[48] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. *POPL*. 2014.
[49] O. Kupferman and M. Y. Vardi. Robust satisfaction. *CONCUR*, 1999.
[50] L. Lamport and F. B. Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, 1984.
[51] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7), 2009.
[52] I. Mastroeni and M. Pasqua. Verifying bounded subset-closed hyperproperties. *SAS*. Springer, 2018.
[53] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1), 1992.
[54] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything you want to know about pointer-based checking. *SNAPL*. 2015.
[55] G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. *ICFP*. 2015.
[56] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. *ICFP*, 2016.
[57] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. *CSF*. 2017.
[58] M. Patrignani and D. Garg. Robustly safe compilation. *ESOP*, 2019.
[59] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *TOPLAS*, 2015.
[60] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019.
[61] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP), 2017.
[62] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*. 2003.
[63] A. W. Roscoe. CSP and determinism in security modelling. *S&P*. 1995.
[64] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *HOSC*, 14(1), 2001.
[65] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.
[66] F. Schneider. *On Concurrent Programming*. Texts in Computer Science. Springer New York, 1997.
[67] L. Simon, D. Chisnall, and R. J. Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. *EuroS&P*. 2018.
[68] L. Skorstengaard, D. Devriese, and L. Birkedal. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *PACMPL*, 3(POPL), 2019.
[69] B. C. Smith. Reflection and semantics in Lisp. *POPL*. 1984.
[70] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.
[71] D. Swasey, D. Garg, and D. Dreyer. Robust and compositional verification of object capability patterns. *PACMPL*, 1(OOPSLA), 2017.
[72] G. Tan. Principles and implementation techniques of software-based fault isolation. *FTSEC*, 1(3), 2017.
[73] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP*, 1993.
[74] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. *S&P*. 2015.
[75] A. Zakinthinos and E. S. Lee. A general theory of security properties. *S&P*. 1997.
[76] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. *CSFW*. 2003.
[77] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. *CCS*, 2017.