# Robustly Safe Compilation

Marco Patrignani[1,2] and Deepak Garg[3]

[1] Stanford University
[2] CISPA Helmholz Center for Information Security
[3] Max Planck Institute for Software Systems

**Abstract.** Secure compilers generate compiled code that withstands many target-level attacks such as alteration of control flow, data leaks or memory corruption. Many existing secure compilers are proven to be fully abstract, meaning that they reflect and preserve observational equivalence. Fully abstract compilation is strong and useful but, in certain cases, comes at the cost of requiring expensive runtime constructs in compiled code. These constructs may have no relevance for security, but are needed to accommodate differences between the source and target languages that fully abstract compilation necessarily needs.
As an alternative to fully abstract compilation, this paper explores a different criterion for secure compilation called robustly safe compilation or *RSC*. Briefly, this criterion means that the compiled code preserves relevant safety properties of the source program against all adversarial contexts interacting with the compiled program. We show that *RSC* can be proved more easily than fully abstract compilation and also often results in more efficient code. We also develop two illustrative robustly-safe compilers and, through them, illustrate two different proof techniques for establishing that a compiler attains *RSC*. Based on these, we argue that proving *RSC* can be simpler than proving fully abstraction.

*To better explain and clarify notions, this paper uses colours. For a better experience, please print or view this paper in colours.*[4]

## 1 Introduction

Low-level adversaries, such as those written in C or assembly can attack co-linked code written in a high-level language in ways that may not be feasible in the high-level language itself. For example, such an adversary may manipulate or hijack control flow, cause buffer overflows, or directly access private memory, all in contravention to the abstractions of the high-level language. Specific countermeasures such as Control Flow Integrity [3] or Code Pointer Integrity [41]

[4] Specifically, in this paper we use a blue, sans-serif font for source elements, an **orange**, **bold** font for **target** elements and a *black*, *italic* font for elements common to both languages (to avoid repeating similar definitions twice). Thus, C is a source-level component, **C** is a target-level component and $C$ is generic notation for either a source-level or a target-level component.

have been devised to address some of these attacks *individually*. An alternative approach is to devise a *secure compiler*, which seeks to defend against entire *classes* of such attacks. Secure compilers often achieve security by relying on different protection mechanisms, e.g., cryptographic primitives [4,5,22,26], types [11,10], address space layout randomisation [6,37], protected module architectures [9,53,59,57] (also know as enclaves [46]), tagged architectures [39,7], etc. Once designed, the question researchers face is how to formalise that such a compiler is indeed secure, and how to prove this. Basically, we want a criterion that specifies secure compilation. A widely-used criterion for compiler security is fully abstract compilation (*FAC*) [2,52,35], which has been shown to preserve many interesting security properties like confidentiality, integrity, invariant definitions, well-bracketed control flow and hiding of local state [37,53,9,54].

Informally, a compiler is fully abstract if it preserves and reflects observational equivalence of source-level components (i.e., partial programs) in their compiled counterparts. Most existing work instantiates observational equivalence with contextual equivalence: co-divergence of two components in any larger context they interact with. Fully abstract compilation is a very strong property, which preserves *all* source-level abstractions.

Unfortunately, preserving *all* source-level abstractions also has downsides. In fact, while *FAC* preserves many relevant security properties, it also preserves a plethora of other non-security ones, and the latter may force inefficient checks in the compiled code. For example, when the target is assembly, two observationally equivalent components must compile to code of the same size [9,53], else full abstraction is trivially violated. This requirement is security-irrelevant in most cases. Additionally, *FAC* is not well-suited for source languages with undefined behaviour (e.g., `C` and `LLVM`) [39] and, if used naïvely, it can fail to preserve even simple safety properties [60] (though, fortunately, no *existing* work falls prey to this naïvety).

Motivated by this, recent work started investigating alternative secure compilation criteria that overcome these limitations. These security-focussed criteria take the form of preservation of hyperproperties or classes of hyperproperties, such as hypersafety properties or safety properties [33,8]. This paper investigates one of these criteria, namely, *Robustly Safe Compilation* (*RSC*) which has clear security guarantees and can often be attained more efficiently than FAC.

Informally, a compiler attains *RSC* if it is correct and it preserves *robust safety* of source components in the target components it produces. Robust safety is an important security notion that has been widely adopted to formalize security, e.g., of communication protocols [17,14,34]. Before explaining *RSC*, we explain robust safety as a language property.

*Robust Safety as a Language Property.* Informally, a program property is a safety property if it encodes that "bad" sequences of events do not happen when the program executes [63,13]. A program is *robustly safe* if it has relevant (specified) safety properties *despite* active attacks from adversaries. As the name suggests, robust safety relies on the notions of safety and robustness which we now explain.

**Safety**. As mentioned, safety asserts that "no bad sequence of events happens", so we can specify a safety property by the set of *finite observations* which characterise all bad sequences of events. A whole program has a safety property if its behaviours exclude these bad observations. Many security properties can be encoded as safety, including integrity, weak secrecy and functional correctness.

*Example 1 (Integrity).* Integrity ensures that an attacker does not tamper with code invariants on state. For example, consider the function `charge_account(n)` which deducts amount `n` from an account as part of an electronic card payment. A card PIN is required if `n` is larger than 10 euros. So the function checks whether `n` $> 10$, requests the PIN if this is the case, and then changes the account balance. We expect this function to have a safety (integrity) property in the account balance: A reduction of more than 10 euros in the account balance must be preceded by a call to `request_pin()`. Here, the relevant observation is a trace (sequence) of account balances and calls to `request_pin()`. Bad observations for this safety property are those where an account balance is at least 10 euros less than the previous one, without a call to `request_pin()` in between. Note that this function seems to have this safety property, but it may not have the safety property *robustly*: a target-level adversary may transfer control directly to the "else" branch of the check `n` $> 10$ after setting `n` to more than 10, to violate the safety property. ⊡

*Example 2 (Weak Secrecy).* Weak secrecy asserts that a program secret never flows *explicitly* to the attacker. For example, consider code that manages `network_h`, a handler (socket descriptor) for a sensitive network interface. This code does not expose `network_h` directly to external code but it provides an API to use it. This API makes some security checks internally. If the handler is directly accessible to outer code, then it can be misused in insecure ways (since the security checks may not be made). If the code has weak secrecy wrt `network_h` then we know that the handler is never passed to an attacker. In this case we can define bad observations as those where `network_h` is passed to external code (e.g., as a parameter, as a return value on or on the heap). ⊡

*Example 3 (Correctness).* Program correctness can also be formalized as a safety property. Consider a program that computes the `n`th Fibonacci number. The program reads `n` from an input source and writes its output to an output source. Correctness of this program is a safety property. Our observations are pairs of an input (read by the program) and the corresponding output. A bad observation is one where the input is `n` (for some `n`) but the output is different from the `n`th Fibonacci number. ⊡

These examples not only illustrate the expressiveness of safety properties, but also show that safety properties are quite *coarse-grained*: they are only concerned with (sequences of) relevant events like calls to specific functions, changes to specific heap variables, inputs, and outputs. They do not specify or constrain how the program computes between these events, leaving the programmer and the compiler considerable flexibility in optimizations. However, safety properties are

not a panacea for security, and there are security properties that are not safety. For example, noninterference [70,72], the standard information flow property, is not safety. Nonetheless, many interesting security properties are safety. In fact, many non-safety properties including noninterference can be conservatively approximated as safety properties [20]. Hence, safety properties are a meaningful goal to pursue for secure compilation.

**Robustness**. We often want to reason about properties of a component of interest that hold irrespective of any other components the component interacts with. These other components may be the libraries the component is linked against, or the language runtime. Often, these surrounding components are modelled as the *program context* whose hole the component of interest fills. From a security perspective the context represents the attacker in the threat model. When the component of interest links to a context, we have a whole program that can run. A property holds *robustly* for a component if it holds in *any* context that the component of interest can be linked to.

*Robust Safety Preservation as a Compiler Property.* A compiler attains robustly safe compilation or *RSC* if it maps any source component that has a safety property *robustly* to a compiled component that has the *same* safety property robustly. Thus, safety has to hold robustly in the target language, which often does not have the powerful abstractions (e.g., typing) that the source language has. Hence, the compiler must insert enough defensive runtime checks into the compiled code to prevent the more powerful target contexts from launching attacks (violations of safety properties) that source contexts could not launch. This is unlike correct compilation, which either considers only those target contexts that behave like source contexts [49,65,40] or considers only whole programs [43].

As mentioned, safety properties are usually quite coarse-grained. This means that *RSC* still allows the compiler to optimise code internally, as long as the sequence of observable events is not affected. For example, when compiling the `fibonacci` function of Example 3, the compiler can do any internal optimisation such as caching intermediate results, as long as the end result is correct. Crucially, however, these intermediate results must be protected from tampering by a (target-level) attacker, else the output can be incorrect, breaking *RSC*.

A *RSC*-attaining compiler focuses only on preserving security (as captured by robust safety) instead of contextual equivalence (typically captured by full abstraction). So, such a compiler can produce code that is more efficient than code compiled with a fully abstract compiler as it does not have to preserve *all* source abstractions (we illustrate this later).

Finally, robust safety scales naturally to thread-based concurrency [34,1,58]. Thus *RSC* also scales naturally to thread-based concurrency (we demonstrate this too). This is unlike *FAC*, where thread-based concurrency can introduce additional undesired abstractions that also need to be preserved.

*RSC* is a very recently proposed criterion for secure compilers. Recent work [33,8] define *RSC* abstractly in terms of preservation of program behaviours, but their development is limited to the definition only. Our goal in this paper is to examine how *RSC* can be realized and established, and to show that in certain

cases it leads to compiled code that is more efficient than what *FAC* leads to. To this end, we consider a specific setting where observations are values in specific (sensitive) heap locations at cross-component calls. We define robust safety and *RSC* for this specific setting (Section 2). Unlike previous work [33,8] which assumed that the domain of traces (bheaviours) is the same in the source and target languages, our *RSC* definition allows for different trace domains in the source and target languages, as long as they can be suitably related. The second contribution of our paper is two proof techniques to establish *RSC*.

- The first technique is an adaption of trace-based backtranslation, an existing technique for proving *FAC* [59,9,7]. To illustrate this technique, we build a compiler from an untyped source language to an untyped target language with support for fine-grained memory protection via so-called capabilities [71,23] (Section 3). Here, we guarantee that if a source program is robustly safe, then so is its compilation.
- The second proof technique shows that if source programs are *verified* for robust safety, then one can simplify the proof of *RSC* so that no backtranslation is needed. In this case, we develop a compiler from a *typed* source language where the types already enforce robust safety, to a target language similar to that of the first compiler (Section 4). In this instance, both languages also support shared-memory concurrency. Here, we guarantee that all compiled target programs are robustly safe.

To argue that *RSC* is general and is not limited to compilation targets based on capabilities, we also develop a third compiler. This compiler starts from the same source language as our second compiler but targets an untyped concurrent language with support for *coarse-grained memory isolation*, modelling recent hardware extensions such as Intel's SGX [46]. Due to space constraints, we report this result only in the companion technical report [61].

The final contribution of this paper is a comparison between *RSC* and *FAC*. For this, we describe changes that would be needed to attain *FAC* for the first compiler and argue that these changes make generated code inefficient and also complicate the backtranslation proof significantly (Section 5).

Due to space constraints, we elide some technical details and limit proofs to sketches. These are fully resolved in the companion technical report [61].

## 2 Robustly Safe Compilation

This section first discusses robust safety as a language (not a compiler) property (Section 2.1) and then presents *RSC* as a compiler property along with an informal discussion of techniques to prove it (Section 2.2).

### 2.1 Safety and Robust Safety

To explain robust safety, we first describe a general *imperative* programming model that we use. Programmers write *components* on which they want to enforce safety properties robustly. A component is a list of function definitions that

can be linked with other components (the context) in order to have a runnable whole program (functions in "other" components are like `extern` functions in C). Additionally, every component declares a set of "sensitive" locations that contain all the data that is safety-relevant. For instance, in Example 1 this set may contain the account balance and in Example 3 it may contain the I/O buffers. We explain the relevance of this set after we define safety properties.

We want safety properties to specify that a component never executes a "bad" sequence of events. For this, we first need to fix a notion of events. We have several choices here, e.g., our events could be inputs and outputs, all syscalls, all changes to the heap (as in CompCert [44]), etc. Here, we make a specific choice motivated by our interest in robustness: We define events as calls/returns that cross a component boundary, together with the state of the heap at that point. Consequently, our safety properties can constrain the contents of the heap at component boundaries. This choice of component boundaries as the point of observation is meaningful because, in our programming model, control transfers to/from an adversary happen only at component boundaries (more precisely, they happen at cross-component function call and returns). This allows the compiler complete flexibility in optimizing code within a component, while not reducing the ability of safety properties to constrain observations of the adversary.

Concretely, a component behaviour is a *trace*, i.e., a sequence of *actions* recording component boundary interactions and, in particular, the heap at these points. *Actions*, the items on a trace, have the following grammar:

$$\textit{Actions}\ \alpha ::= \texttt{call}\ f\ v\ H? \mid \texttt{call}\ f\ v\ H! \mid \texttt{ret}\ H! \mid \texttt{ret}\ H?$$

These actions respectively capture call and callback to a function $f$ with parameter $v$ when the heap is $H$ as well as return and returnback with a certain heap $H$.[5] We use ? and ! decorations to indicate whether the control flow of the action goes from the context to the component (?) or from the component to the context (!). Well-formed traces have alternations of ? and ! decorated actions, starting with ? since execution starts in the context. For a sequence of actions $\overline{\alpha}$, $\texttt{relevant}(\overline{\alpha})$ is the list of heaps $\overline{H}$ mentioned in the actions of $\overline{\alpha}$.

Next, we need a representation of safety properties. Generally, properties are sets of traces, but safety properties specifically can be specified as automata (or monitors in the sequel) [63]. We choose this representation since monitors are less abstract than sets of traces and they are closer to enforcement mechanisms used for safety properties, e.g., runtime monitors. Briefly, a safety property is a monitor that transitions states in response to events of the program trace. At any point, the monitor may refuse to transition (it gets *stuck*), which encodes property violation. While a monitor can transition, the property has not been violated. Schneider [63] argues that all properties codable this way are safety properties and that all enforceable safety properties can be coded this way.

[5] A callback is a call from the component to the context, so it generates label $\texttt{call}\ f\ v\ H!$. A returnback is a return from such a callback, i.e., the context returning to the component, and it generates the label $\texttt{ret}\ H?$.

Formally, a monitor $M$ in our setting consists of a set of abstract states $\{\sigma \cdots\}$, the transition relation $\rightsquigarrow$, an initial state $\sigma_0$, the set of heap locations that matter for the monitor, $\{l \cdots\}$, and the current state $\sigma_c$ (we indicate a set of elements of class $e$ as $\{e \cdots\}$). The transition relation $\rightsquigarrow$ is a set of triples of the form $(\sigma_s, H, \sigma_f)$ consisting of a starting state $\sigma_s$, a final state $\sigma_f$ and a heap $H$. The transition $(\sigma_s, H, \sigma_f)$ is interpreted as "*state $\sigma_s$ transitions to $\sigma_f$ when the heap is $H$*". When determining the monitor transition in response to a program action, we restrict the program's heap to the location set $\{l \cdots\}$, i.e., to the set of locations the monitor cares about. This heap restriction is written $H\big|_{\{l \cdots\}}$. We assume determinism of the transition relation: for any $\sigma_s$ and (restricted heap) $H$, there is at most one $\sigma_f$ such that $(\sigma_s, H, \sigma_f) \in \rightsquigarrow$.

Given the behaviour of a program as a trace $\overline{\alpha}$ and a monitor $M$ specifying a safety property, $M \vdash \overline{\alpha}$ denotes that the trace satisfies the safety property. Intuitively, to satisfy a safety property, the sequence of heaps in the actions of a trace must never get the monitor stuck (Rule Valid trace). Every single heap must allow the monitor to step according to its transition relation (Rule Monitor Step). Note that we overload the $\rightsquigarrow$ notation here to also denote an auxiliary relation, the *monitor small-step semantics* (Rule Monitor Step-base and Rule Monitor Step-ind).

$$\frac{M; \texttt{relevant}(\overline{\alpha}) \rightsquigarrow M'}{M \vdash \overline{\alpha}}\ (\text{Valid trace}) \qquad \frac{}{M; \varnothing \rightsquigarrow M}\ (\text{Monitor Step-base}) \qquad \frac{M; \overline{H} \rightsquigarrow M'' \quad M''; H \rightsquigarrow M'}{M; \overline{H} \cdot H \rightsquigarrow M'}\ (\text{Monitor Step-ind})$$

$$\frac{(\sigma_c, H\big|_{\{l \cdots\}}, \sigma_f) \in \rightsquigarrow}{(\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \{l \cdots\}, \sigma_c); H \rightsquigarrow (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \{l \cdots\}, \sigma_f)}\ (\text{Monitor Step})$$

With this setup in place, we can formalise safety, attackers and robust safety. In defining (robust) safety for a component, we only admit monitors (safety properties) whose $\{l \cdots\}$ agrees with the sensitive locations declared by the component. Making the set of safety-relevant locations explicit in the component and the monitor gives the compiler more flexibility by telling it precisely which locations need to be protected against target-level attacks (the compiler may choose to not protect the rest). At the same time, it allows for expressive modelling. For instance, in Example 3 the safety-relevant locations could be the I/O buffers from which the program performs inputs and outputs, and the safety property can constrain the input and output buffers at corresponding call and return actions involving the Fibonacci function.

**Definition 1 (Safety, attacker and robust safety).**

$$M \vdash C : \mathit{safe} \stackrel{\text{def}}{=} \mathit{if}\ \vdash C : \mathit{whole}\ \mathit{then}\ \mathit{if}\ \Omega_0(C) \stackrel{\overline{\alpha}}{\Longrightarrow} \_\ \mathit{then}\ M \vdash \overline{\alpha}$$

$$C \vdash A : \mathit{atk} \stackrel{\text{def}}{=} C = \{l \cdots\}, \overline{F}\ \mathit{and}\ \{l \cdots\} \cap \texttt{fn}(A) = \varnothing$$

$$M \vdash C : \mathit{rs} \stackrel{\text{def}}{=} \forall A.\ \mathit{if}\ M \frown C\ \mathit{and}\ C \vdash A : \mathit{atk}\ \mathit{then}\ M \vdash A[C] : \mathit{safe}$$

A whole program $C$ is safe for a monitor $M$, written $M \vdash C : \mathit{safe}$, if the monitor accepts any trace the program generates from its initial state ($\Omega_0(C)$).

An attacker $A$ is valid for a component $C$, written $C \vdash A : atk$, if $A$'s free names (denoted $\texttt{fn}(A)$) do not refer to the locations that the component cares about. This is a basic sanity check: if we allow an attacker to mention heap locations that the component cares about, the attacker will be able to modify those locations, causing all but trivial safety properties to not hold robustly.

A component $C$ is robustly safe wrt monitor $M$, written $M \vdash C : rs$, if $C$ composed with *any* attacker is safe wrt $M$. As mentioned, for this setup to make sense, the monitor and the component must agree on the locations that are safety-relevant. This agreement is denoted $M \frown C$.

### 2.2 Robustly Safe Compilation

Robustly-safe compilation ensures that robust safety properties *and their meanings* are preserved across compilation. But what does it means to preserve meanings across languages? If a source safety property says never write 3 to a location, and we compile to an assembly language by mapping numbers to binary, the corresponding target property should say **never write 0x11 to an address**.

In order to relate properties across languages, we assume a relation $\approx : \mathsf{v} \times \mathbf{v}$ between source and target values that is *total*, so it maps any source value $\mathsf{v}$ to a target value $\mathbf{v}$: $\forall \mathsf{v}.\exists \mathbf{v}.\mathsf{v} \approx \mathbf{v}$. This value relation is used to define a relation between heaps: $\mathsf{H} \approx \mathbf{H}$, which intuitively holds when related locations point to related values. This is then used to define a relation between actions: $\alpha \approx \alpha$, which holds when the two actions are the "same" modulo this relation, i.e., $\texttt{call} \cdot \cdot \cdot ?$ only relates to $\texttt{call} \cdot \cdot \cdot ?$ and the arguments of the action (values and heap) are related. Next, we require a relation $\mathsf{M} \approx \mathbf{M}$ between source and target monitors, which means that the source monitor $\mathsf{M}$ and the target monitor $\mathbf{M}$ code the same safety property, modulo the relation $\approx$ on values assumed above. The precise definition of this relation depends on the source and target languages; specific instances are shown in Sections 3.3 and 4.3.[6]

We denote a compiler from language $\mathsf{S}$ to language $\mathbf{T}$ by $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$. A compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ attains *RSC*, if it maps any component $\mathsf{C}$ that is robustly safe wrt $\mathsf{M}$ to a component $\mathbf{C}$ that is robustly safe wrt $\mathbf{M}$, provided that $\mathsf{M} \approx \mathbf{M}$.

**Definition 2 (Robustly Safe Compilation).**

$$\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : RSC \stackrel{\text{def}}{=} \forall \mathsf{C}, \mathsf{M}, \mathbf{M}.\ \textit{if}\ \mathsf{M} \vdash \mathsf{C} : \mathsf{rs}\ \textit{and}\ \mathsf{M} \approx \mathbf{M}\ \textit{then}\ \mathbf{M} \vdash [\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathbf{rs}$$

A consequence of the universal quantification over monitors here is that the compiler cannot be property-sensitive. A robustly-safe compiler preserves all robust safety properties, not just a specific one, e.g., it does not just enforce that `fibonacci` is correct. This seemingly strong goal is sensible as compiler writers will likely not know what safety properties individual programmers will want to preserve.

[6] Accounting for the difference in the representation of safety properties sets us apart from recent work [33,8], which assumes that the source and target languages have the same trace alphabet. The latter works only in some settings.

*Remark.* Some readers may wonder why we do not follow existing work and specify safety as "programmer-written assertions never fail" [34,31,45,68]. Unfortunately, this approach does not yield a meaningful criterion for specifying a compiler, since assertions in the compiled program (if any) are generated by the compiler itself. Thus a compiler could just erase all assertions and the compiled code it generates would be trivially (robustly) safe – no assertion can fail if there are no assertions in the first place!

**Proving *RSC*.** Proving that a compiler attains *RSC* can be done either by proving that a compiler satisfies Definition 2 or by proving something *equivalent*. To this end, Definition 3 below presents an alternative, equivalent formulation of *RSC*. We call this characterisation *property-free* as it does not mention monitors explicitly (it mentions the $\mathtt{relevant}(\cdot)$ function for reasons we explain below).

**Definition 3 (Property-Free *RSC*).**

$$\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{PF\text{-}RSC} \stackrel{\text{def}}{=} \forall \mathsf{C}, \mathbf{A}, \overline{\alpha}.$$

$$\textit{if}\ \ [\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}} \vdash \mathbf{A} : \mathbf{atk}\ \textit{and}\ \vdash \mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right] : \mathbf{whole}\ \textit{and}\ \boldsymbol{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right]\right) \stackrel{\overline{\alpha}}{\Longrightarrow} \_$$

$$\textit{then}\ \ \exists \mathsf{A}, \overline{\alpha}.\ \mathsf{C} \vdash \mathsf{A} : \mathsf{atk}\ \textit{and}\ \vdash \mathsf{A}[\mathsf{C}] : \mathsf{whole}\ \textit{and}\ \Omega_0(\mathsf{A}[\mathsf{C}]) \stackrel{\overline{\alpha}}{\Longrightarrow} \_$$
$$\textit{and}\ \ \mathtt{relevant}(\overline{\alpha}) \approx \mathtt{relevant}(\overline{\alpha})$$

Specifically, *PF-RSC* states that the compiled code produces behaviours that *refine* source level behaviours *robustly* (taking contexts into account).

*PF-RSC* and *RSC* should, in general, be equivalent (Proposition 1).

**Proposition 1 (*PF-RSC* and *RSC* are equivalent).**

$$\forall [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}, \vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{PF\text{-}RSC} \iff \vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{RSC}$$

Informally, a property is safety if and only if it implies programs not having any trace prefix from a given set of bad prefixes (i.e., finite traces). Hence, *not* having a safety property robustly amounts to some context being able to induce a bad prefix. Consequently, preserving *all* robust safety properties (*RSC*) amounts to ensuring that all target prefixes can be generated (by some context) in the source too (*PF-RSC*). Formally, since Definition 2 relies on the monitor relation, we can prove Proposition 1 only after such a relation is finalised. We give such a monitor relation and proof in Section 3.3 (see Theorem 3). However, in general this result should hold for any cross-language monitor relation that correctly relates safety properties. If the proposition does not hold, then the relation does not capture how safety in one language is represented in the other.

Assuming Proposition 1, we can prove *PF-RSC* for a compiler in place of *RSC*. *PF-RSC* can be proved with a *backtranslation* technique. This technique has been often used to prove full abstraction [50,59,39,53,9,33,7,54,8] and it aims at building a source context starting from a target one. In fact *PF-RSC*, leads directly to a backtranslation-based proof technique since it can be rewritten (eliding irrelevant details) as:

$$\text{If}\ \exists \mathbf{A}, \overline{\alpha}.\boldsymbol{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right]\right) \stackrel{\overline{\alpha}}{\Longrightarrow} \_$$

then $\exists \mathsf{A}, \overline{\alpha}.\Omega_0\,(\mathsf{A}\,[\mathsf{C}]) \overset{\overline{\alpha}}{\Longrightarrow} \_$ and $\texttt{relevant}(\overline{\alpha}) \approx \texttt{relevant}(\overline{\alpha})$

Essentially, given a target context $\mathbf{A}$, a compiled program $[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}$ and a target trace $\overline{\alpha}$ that $\mathbf{A}$ causes $[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}$ to have, we need to construct, or *backtranslate* to, a source context $\mathsf{A}$ that will cause the source program $\mathsf{C}$ to simulate $\overline{\alpha}$. Such backtranslation based proofs can be quite difficult, depending on the features of the languages and the compiler. However, backtranslation for *RSC* (as we show in Section 3.3) is not as complex as backtranslation for *FAC* (Section 5.2).

A simpler proof strategy is also viable for *RSC* when we compile only those source programs that have been *verified* to be robustly safe (e.g., using a type system). The idea is this: from the verification of the source program, we can find an invariant which is always maintained by the target code, and which, in turn, implies the robust safety of the target code. For example, if the safety property is that values in the heap always have their expected types, then the invariant can simply be that values in the target heap are always related to the source ones (which have their expected types). This is tantamount to proving type preservation in the target in the presence of an active adversary. This is harder than standard type preservation (because of the active adversary) but is still much easier than backtranslation as there is no need to map target constructs to source contexts syntactically. We illustrate this proof technique in Section 4.

*RSC* **Implies Compiler Correctness.** As stated in Section 1, *RSC* implies (a form of) compiler correctness. While this may not be apparent from Definition 2, it is more apparent from its equivalent characterization in Definition 3. We elaborate this here.

Whether concerned with whole programs or partial programs, compiler correctness states that the behaviour of compiled programs *refines* the behaviour of source programs [44,65,40,49,36,18]. So, if $\{\overline{\alpha}\cdots\}$ and $\{\overline{\alpha}\cdots\}$ are the sets of compiled and source behaviours, then a compiler should force $\{\overline{\alpha}\cdots\} \lesssim \{\overline{\alpha}\cdots\}$, where $\lesssim$ is the composition of $\subseteq$ and of the relation $\approx^{-1}$.

If we consider a source component $\mathsf{C}$ that is whole, then it can only link against empty contexts, both in the source and in the target. Hence, in this special case, *PF-RSC* simplifies to standard refinement of traces, i.e., whole program compiler correctness. Hence, assuming that the correctness criterion for a compiler is concerned with the same observations as safety properties (values in safety-relevant heap locations at component crossings in our illustrative setting), *PF-RSC* implies whole program compiler correctness.

However, *PF-RSC* (or, equivalently, *RSC*) does not imply, nor is implied by, any form of *compositional compiler correctness* (CCC) [65,40,49]. CCC requires that the behaviours produced by a compiled component linked against a target context that is related (in behaviour) to a source context can also be produced by the source component linked against the *related* source context. In contrast, *PF-RSC* allows picking *any* source context to simulate the behaviours. Hence, *PF-RSC* does not imply CCC. On the other hand, *PF-RSC* universally quantifies over all target contexts, while CCC only quantifies over target contexts related to a source context, so CCC does not imply *PF-RSC* either. Hence,

compositional compiler correctness, if desirable, must be imposed in addition to *PF-RSC*. Note that this lack of implications is unsurprising: *PF-RSC* and CCC capture two very different aspects of compilation: security (against all contexts) and compositional preservation of behaviour (against well-behaved contexts).

## 3 *RSC* via Trace-based Backtranslation

This section illustrates how to prove that a compiler attains *RSC* by means of a trace-based backtranslation technique [59,53,7]. To present such a proof, we first introduce our source language $\mathsf{L}^{\mathsf{U}}$, an untyped, first-order imperative language with abstract references and hidden local state (Section 3.1). Then, we present our target language $\mathbf{L}^{\mathbf{P}}$, an untyped imperative target language with a concrete heap, whose locations are natural numbers that the context can compute. $\mathbf{L}^{\mathbf{P}}$ provides hidden local state via a fine-grained capability mechanism on heap accesses (Section 3.2). Finally, we present the compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ and prove that it attains *RSC* (Section 3.3) by means of a trace-based backtranslation. The section conclude with an example detailing why *RSC* preserves security (Example 4).

To avoid focussing on mundane details, we deliberately use source and target languages that are fairly similar. However, they differ substantially in one key point: the heap model. This affords the target-level adversary attacks like guessing private locations and writing to them that do not obviously exist in the source (and makes our proofs nontrivial). We believe that (with due effort) the ideas here will generalize to languages with larger gaps and more features.

### 3.1 The Source Language $\mathsf{L}^{\mathsf{U}}$

$\mathsf{L}^{\mathsf{U}}$ is an untyped imperative while language [51]. Components $\mathsf{C}$ are triples of function definitions, interfaces and a special location written $\ell_{\mathsf{root}}$, so $\mathsf{C} ::= \ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}$. Each function definition maps a function name and a formal argument to a body $\mathsf{s}$: $\mathsf{F} ::= \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; $. An interface is a list of functions that the component relies on the context to provide (similar to C's `extern` declarations). The special location $\ell_{\mathsf{root}}$ defines the locations that are monitored for safety, as explained below. Attackers $\mathsf{A}$ (program contexts) are function definitions that represent untrusted code that a component interacts with. A function's body is a statement, $\mathsf{s}$. Statements are rather standard, so we omit a formal syntax. Briefly, they can manipulate the heap (location creation $\mathsf{let\ x = new\ e\ in\ s}$, assignment $\mathsf{x := e}$), do recursive function calls ($\mathsf{call\ f\ e}$), condition (if-then-else), define local variables (let-in) and loop. Statements use effect-free expressions, $\mathsf{e}$, which contain standard boolean expressions ($\mathsf{e} \otimes \mathsf{e}$), arithmetic expressions ($\mathsf{e} \oplus \mathsf{e}$), pairing ($\langle \mathsf{e}, \mathsf{e} \rangle$) and projections, and location dereference ($!\mathsf{e}$). Heaps $\mathsf{H}$ are maps from abstract locations $\ell$ to values $\mathsf{v}$.

As explained in Section 2.1, safety properties are specified by monitors. $\mathsf{L}^{\mathsf{U}}$'s monitors have the form: $\mathsf{M} ::= (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}})$. Note that in place of the set $\{l \cdots\}$ of safety-relevant locations, the description of a monitor here (as well as a component above) contains a *single* location $\ell_{\mathsf{root}}$. The interpretation is

that any location *reachable* in the heap starting from $\ell_{\mathsf{root}}$ is relevant for safety. This set of locations can change as the program executes, and hence this is more flexible than statically specifying all of $\{l \cdots\}$ upfront. This representation of the set by a single location is made explicit in the following monitor rule:

$$\frac{\mathsf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}}) \quad \mathsf{M}' = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{f}}) \quad (\sigma_{\mathsf{c}}, \mathsf{H}', \sigma_{\mathsf{f}}) \in \rightsquigarrow \quad \mathsf{H}' \subseteq \mathsf{H} \quad \mathtt{dom}(\mathsf{H}') = \mathtt{reach}(\ell_{\mathsf{root}}, \mathsf{H})}{\mathsf{M}; \mathsf{H} \rightsquigarrow \mathsf{M}'} \; (\mathsf{L}^{\mathsf{U}}\text{-Monitor Step})$$

Other than this small point, monitors, safety, robust safety and *RSC* are defined as in Section 2. In particular, a monitor and a component agree if they mention the same $\ell_{\mathsf{root}}$: $\mathsf{M} \frown \mathsf{C} \stackrel{\text{def}}{=} (\mathsf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}}))$ and $(\mathsf{C} = (\ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}))$

A program state $\mathsf{C}, \mathsf{H} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}}$ (denoted with $\Omega$) includes the function bodies $\mathsf{C}$, the heap $\mathsf{H}$, a statement $\mathsf{s}$ being executed and a stack of function calls $\overline{\mathsf{f}}$ (often omitted in the rules for simplicity). The latter is used to populate judgements of the form $\overline{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal/in/out}$. These determine whether calls and returns are internal (within the attacker or within the component), directed from the attacker to the component (in) or directed from the component to the attacker (out). This information is used to determine whether the semantics should generate a label, as in Rules $\mathsf{EL}^{\mathsf{U}}$-return to $\mathsf{EL}^{\mathsf{U}}$-retback, or no label, as in Rules $\mathsf{EL}^{\mathsf{U}}$-ret-internal and $\mathsf{EL}^{\mathsf{U}}$-call-internal since internal calls should not be observable. $\mathsf{L}^{\mathsf{U}}$ has a big-step semantics for expressions ($\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\rightarrow \mathsf{v}$) that relies on evaluation contexts, a small-step semantics for statements ($\Omega \xrightarrow{\lambda} \Omega'$) that has labels $\lambda ::= \epsilon \mid \alpha$ and a semantics that accumulates labels in traces ($\Omega \stackrel{\overline{\alpha}}{\Longrightarrow} \Omega'$) by omitting silent actions $\epsilon$ and concatenating the rest. Unlike existing work on compositional compiler correctness which only rely on having the component [40], the semantics relies on having both the component and the context.

$$\frac{\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\rightarrow \mathsf{v} \quad \ell \notin \mathtt{dom}(\mathsf{H})}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{let\ x = new\ e\ in\ s} \longrightarrow \mathsf{C}, \mathsf{H}; \ell \mapsto \mathsf{v} \triangleright \mathsf{s}[\ell \,/\, \mathsf{x}]} \; (\mathsf{EL}^{\mathsf{U}}\text{-alloc})$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \quad \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{out}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'};\mathsf{f}} \xrightarrow{\text{ret H!}} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}} \; (\mathsf{EL}^{\mathsf{U}}\text{-return})$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \quad \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; \in \mathsf{C}.\mathtt{funs} \quad \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}', \mathsf{f} : \mathsf{in} \quad \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\rightarrow \mathsf{v}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{call\ f\ e})_{\overline{\mathsf{f}'}} \xrightarrow{\text{call f v H?}} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s}; \mathsf{return};[\mathsf{v} \,/\, \mathsf{x}])_{\overline{\mathsf{f}'};\mathsf{f}}} \; (\mathsf{EL}^{\mathsf{U}}\text{-call})$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \quad \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; \in \overline{\mathsf{F}} \quad \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}', \mathsf{f} : \mathsf{out} \quad \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\rightarrow \mathsf{v}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{call\ f\ e})_{\overline{\mathsf{f}'}} \xrightarrow{\text{call f v H!}} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s}; \mathsf{return};[\mathsf{v} \,/\, \mathsf{x}])_{\overline{\mathsf{f}'};\mathsf{f}}} \; (\mathsf{EL}^{\mathsf{U}}\text{-callback})$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \quad \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{in}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'};\mathsf{f}} \xrightarrow{\text{ret H?}} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}} \; (\mathsf{EL}^{\mathsf{U}}\text{-retback})$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \quad \overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'};\mathsf{f}} \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}} \; (\mathsf{EL}^{\mathsf{U}}\text{-ret-internal})$$

$$\frac{\overline{\mathsf{C}}.\mathtt{intfs} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal} \quad \overline{\mathsf{f}'} = \overline{\mathsf{f}''}; \mathsf{f}' \quad \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; \in \mathsf{C}.\mathtt{funs} \quad \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\rightarrow \mathsf{v}}{\mathsf{C}, \mathsf{H} \triangleright (\mathsf{call\ f\ e})_{\overline{\mathsf{f}'}} \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s}; \mathsf{return};[\mathsf{v} \,/\, \mathsf{x}])_{\overline{\mathsf{f}'};\mathsf{f}}} \; (\mathsf{EL}^{\mathsf{U}}\text{-call-internal})$$

### 3.2 The Target Language $\mathbf{L^P}$

$\mathbf{L^P}$ is an untyped, imperative language that follows the structure of $\mathsf{L}^{\mathsf{U}}$ and it has similar expressions and statements. However, there are critical differences (that make the compiler interesting). The main difference is that heap locations in $\mathbf{L^P}$ are concrete natural numbers. Upfront, an adversarial context can guess locations used as private state by a component and clobber them. To support hidden local state, a location can be "hidden" explicitly via the statement $\mathbf{let\ x = hide\ e\ in\ s}$, which allocates a new capability $\mathbf{k}$, an abstract token that grants access to the location $\mathbf{n}$ to which $\mathbf{e}$ points [64]. Subsequently, all reads and writes to $\mathbf{n}$ must be authenticated with the capability, so reading and writing a location take another parameter as follows: $\mathbf{!e\ with\ e}$ and $\mathbf{x := e\ with\ e}$. In both cases, the $\mathbf{e}$ after the $\mathbf{with}$ is the capability. Unlike locations, capabilities cannot be guessed. To make a location private, the compiler can make the capability of the location private. To bootstrap this hiding process, we assume that a component has one location that can only be accessed by it, a priori in the semantics (in our formalization, we always focus on only one component and we assume that, for this component, this special location is at address $\mathbf{0}$).

In detail, $\mathbf{L^P}$ heaps $\mathbf{H}$ are maps from natural numbers (locations) $\mathbf{n}$ to values $\mathbf{v}$ and a tag $\eta$ as well as capabilities, so $\mathbf{H} ::= \varnothing \mid \mathbf{H}; \mathbf{n} \mapsto \mathbf{v} : \eta \mid \mathbf{H}; \mathbf{k}$. The tag $\eta$ can be $\bot$, which means that $\mathbf{n}$ is globally available (not protected) or a capability $\mathbf{k}$, which protects $\mathbf{n}$. A globally available location can be freely read and written but one that is protected by a capability requires the capability to be supplied at the time of read/write (Rule E$\mathbf{L^P}$-assign, Rule E$\mathbf{L^P}$-deref).

$\mathbf{L^P}$ also has a big-step semantics for expressions, a labelled small-step semantics and a semantics that accumulates traces analogous to that of $\mathsf{L}^{\mathsf{U}}$.

$$\frac{\mathbf{n} \mapsto \mathbf{v} : \eta \in \mathbf{H} \qquad (\eta = \bot) \text{ or } (\eta = \mathbf{k} \text{ and } \mathbf{v}' = \mathbf{k})}{\mathbf{H} \triangleright \mathbf{!n\ with\ v}' \hookrightarrow\!\!\!\!\rightarrow \mathbf{H} \triangleright \mathbf{v}} \text{(E}\mathbf{L^P}\text{-deref)}$$

$$\frac{\mathbf{H} = \mathbf{H_1}; \mathbf{n} \mapsto (\mathbf{v}, \eta) \qquad \mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{v} \qquad \mathbf{H'} = \mathbf{H}; \mathbf{n+1} \mapsto \mathbf{v} : \bot}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x = new\ e\ in\ s} \longrightarrow \mathbf{C}, \mathbf{H'} \triangleright \mathbf{s[n+1\ /\ x]}} \text{(E}\mathbf{L^P}\text{-new)}$$

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{n} \qquad \mathbf{k} \notin \mathtt{dom}(\mathbf{H}) \qquad \mathbf{H} = \mathbf{H_1}; \mathbf{n} \mapsto \mathbf{v} : \bot; \mathbf{H_2} \qquad \mathbf{H'} = \mathbf{H_1}; \mathbf{n} \mapsto \mathbf{v} : \mathbf{k}; \mathbf{H_2}; \mathbf{k}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x = hide\ e\ in\ s} \longrightarrow \mathbf{C}, \mathbf{H'} \triangleright \mathbf{s[k\ /\ x]}} \text{(E}\mathbf{L^P}\text{-hide)}$$

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{v} \qquad \mathbf{H} = \mathbf{H_1}; \mathbf{n} \mapsto \_ : \eta; \mathbf{H_2} \qquad \mathbf{H'} = \mathbf{H_1}; \mathbf{n} \mapsto \mathbf{v} : \eta; \mathbf{H_2} \qquad (\eta = \bot) \text{ or } (\eta = \mathbf{k} \text{ and } \mathbf{v}' = \mathbf{k})}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{n := e\ with\ v}' \longrightarrow \mathbf{C}, \mathbf{H'} \triangleright \mathbf{skip}} \text{(E}\mathbf{L^P}\text{-assign)}$$

A second difference between $\mathbf{L^P}$ and $\mathsf{L}^{\mathsf{U}}$ is that $\mathbf{L^P}$ has no booleans, while $\mathsf{L}^{\mathsf{U}}$ has them. This makes the compiler and the related proofs interesting, as discussed in the proof of Theorem 1.

In $\mathbf{L^P}$, the locations of interest to a monitor are all those that can be reached from the address $\mathbf{0}$. $\mathbf{0}$ itself is protected with a capability $\mathbf{k_{root}}$ that is assumed to occur only in the code of the component in focus, so a component is defined as $\mathbf{C} ::= \mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}$. We can now give a precise definition of component-monitor agreement for $\mathbf{L^P}$ as well as a precise definition of attacker, which must care

about the $\mathbf{k_{root}}$ capability.

$$\mathbf{M} \frown \mathbf{C} \stackrel{\text{def}}{=} (\mathbf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \mathbf{k_{root}}, \sigma_c)) \text{ and } (\mathbf{C} = (\mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}))$$

$$\mathbf{C} \vdash \mathbf{A} : \mathbf{atk} \stackrel{\text{def}}{=} \mathbf{C} = (\mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}), \mathbf{A} = \overline{\mathbf{F}'}, \mathbf{k_{root}} \notin \mathtt{fn}(\overline{\mathbf{F}'})$$

### 3.3 Compiler from $\mathsf{L}^{\mathsf{U}}$ to $\mathbf{L}^{\mathbf{P}}$

We now present $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$, the compiler from $\mathsf{L}^{\mathsf{U}}$ to $\mathbf{L}^{\mathbf{P}}$, detailing how it uses the capabilities of $\mathbf{L}^{\mathbf{P}}$ to achieve *RSC*. Then, we prove that $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ attains *RSC*.

Compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ takes as input a $\mathsf{L}^{\mathsf{U}}$ component $\mathsf{C}$ and returns a $\mathbf{L}^{\mathbf{P}}$ component (excerpts of the translation are shown below). The compiler performs a simple pass on the structure of functions, expressions and statements. Each $\mathsf{L}^{\mathsf{U}}$ location is encoded as a pair of a $\mathbf{L}^{\mathbf{P}}$ location and the capability to access the location; location update and dereference are compiled accordingly. The compiler codes source booleans $\mathsf{true}$ to $\mathbf{0}$ and $\mathsf{false}$ to $\mathbf{1}$, and the source number $\mathsf{n}$ to the target counterpart $\mathbf{n}$.

$$[\![\ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \mathbf{k_{root}}; [\![\overline{\mathsf{F}}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}; [\![\overline{\mathsf{I}}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$$

$$[\![!\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = ![\![\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}.\mathbf{1} \textbf{ with } [\![\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}.\mathbf{2}$$

$$\left[\!\!\left[\begin{array}{l}\mathsf{let\ x = new\ e}\\ \mathsf{in\ s}\end{array}\right]\!\!\right]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \begin{array}{l}\textbf{let } \mathbf{x_{loc}} = \textbf{new } [\![\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} \textbf{ in let } \mathbf{x_{cap}} = \textbf{hide } \mathbf{x_{loc}} \textbf{ in}\\ \quad \textbf{let } \mathbf{x} = \langle \mathbf{x_{loc}}, \mathbf{x_{cap}} \rangle \textbf{ in } [\![\mathsf{s}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\end{array}$$

$$[\![\mathsf{x} := \mathsf{e}']\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \textbf{let } \mathbf{x_{loc}} = \mathbf{x.1} \textbf{ in let } \mathbf{x_{cap}} = \mathbf{x.2} \textbf{ in } \mathbf{x_{loc}} := [\![\mathsf{e}']\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} \textbf{ with } \mathbf{x_{cap}}$$

This compiler solely relies on the capability abstraction of the target language as a defence mechanism to attain *RSC*. Unlike existing secure compilers, $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ needs neither dynamic checks nor other constructs that introduce runtime overhead to attain *RSC* [32,59,53,9,39].

**Proof of *RSC*.** Compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ attains *RSC* (Theorem 1). In order to set up this theorem, we need to instantiate the cross-language relation for values, which we write as $\approx_\beta$ here. The relation is parametrised by a partial bijection $\beta : \ell \times \mathbf{n} \times \eta$ from source heap locations to target heap locations which determines when a source location and a target location (and its capability) are related. On values, $\approx_\beta$ is defined as follows: $\mathsf{true} \approx_\beta \mathbf{0}$; $\mathsf{false} \approx_\beta \mathbf{n}$ when $\mathbf{n} \neq \mathbf{0}$; $\mathsf{n} \approx_\beta \mathbf{n}$; $\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle$ if $(\ell, \mathbf{n}, \mathbf{k}) \in \beta$; $\ell \approx_\beta \langle \mathbf{n}, \_ \rangle$ if $(\ell, \mathbf{n}, \bot) \in \beta$; $\langle \mathsf{v}_1, \mathsf{v}_2 \rangle \approx_\beta \langle \mathbf{v_1}, \mathbf{v_2} \rangle$ if $\mathsf{v}_1 \approx_\beta \mathbf{v_1}$ and $\mathsf{v}_2 \approx_\beta \mathbf{v_2}$. This relation is then used to define the heap, monitor state and action relations. Heaps are related, written $\mathsf{H} \approx_\beta \mathbf{H}$, when locations related in $\beta$ point to related values. States are related, written $\Omega \approx_\beta \mathbf{\Omega}$, when they have related heaps. The action relation ($\alpha \approx_\beta \alpha$) is defined as in Section 2.2.

*Monitor Relation.* In Section 2.2, we left the monitor relation abstract. Here, we define it for our two languages. Two monitors are related when they can *simulate* each other on related heaps. Given a monitor-specific relation $\sigma \approx \sigma$ on monitor states, we say that a relation $\mathcal{R}$ on source and target monitors is a

*bisimulation* if the following hold whenever $\mathsf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}})$ and $\mathbf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{k_{root}}, \sigma_{\mathbf{c}})$ are related by $\mathcal{R}$:

1. $\sigma_0 \approx \sigma_{\mathbf{0}}$, and $\sigma_{\mathsf{c}} \approx \sigma_{\mathbf{c}}$, and
2. For all $\beta$ containing $(\ell_{\mathsf{root}}, \mathbf{0}, \mathbf{k_{root}})$ and all $\mathsf{H}, \mathbf{H}$ with $\mathsf{H} \approx_\beta \mathbf{H}$:
   (a) $(\sigma_{\mathsf{c}}, \mathsf{H}, \_) \in \rightsquigarrow$ iff $(\sigma_{\mathbf{c}}, \mathbf{H}, \_) \in \rightsquigarrow$, and
   (b) $(\sigma_{\mathsf{c}}, \mathsf{H}, \sigma') \in \rightsquigarrow$ and $(\sigma_{\mathbf{c}}, \mathbf{H}, \sigma') \in \rightsquigarrow$ imply
   $(\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma') \mathcal{R} (\{\sigma \cdots\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{k_{root}}, \sigma')$.

In words, $\mathcal{R}$ is a bisimulation only if $\mathsf{M} \mathcal{R} \mathbf{M}$ implies that $\mathsf{M}$ and $\mathbf{M}$ simulate each other on heaps related by *any* $\beta$ that relates $\ell_{\mathsf{root}}$ to $\mathbf{0}$. In particular, this means that neither $\mathsf{M}$ nor $\mathbf{M}$ can be sensitive to the *specific* addresses allocated during the run of the program. However, they can be sensitive to the "shape" of the heap or the values stored in the heap. Note that the union of any two bisimulations is a bisimulation. Hence, there is a largest bisimulation, which we denote as $\approx$. Intuitively, $\mathsf{M} \approx \mathbf{M}$ implies that $\mathsf{M}$ and $\mathbf{M}$ encode the same safety property (up to the aforementioned relation on values $\approx_\beta$). With all the boilerplate for *RSC* in place, we state our main theorem.

**Theorem 1 ($[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ attains *RSC*).** $\vdash [\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} : RSC$

We outline our proof of Theorem 1, which relies on a backtranslation $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L}^{\mathbf{P}}}_{\mathsf{L}^{\mathsf{U}}}$. Intuitively, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L}^{\mathbf{P}}}_{\mathsf{L}^{\mathsf{U}}}$ takes a target trace $\overline{\alpha}$ and builds a *set* of source contexts such that *one* of them when linked with $\mathsf{C}$, produces a related trace $\overline{\alpha}$ in the source (Theorem 2). In prior work, backtranslations return a single context [53,59,11,21,50,10,28]. This is because they all, explicitly or implicitly, assume that $\approx$ is injective from source to target. Under this assumption, the backtranslation is unique: a target value $\mathbf{v}$ will be related to at most one source value $\mathsf{v}$. We do away with this assumption (e.g., the target value $\mathbf{0}$ is related to both source values $\mathsf{0}$ and $\mathsf{true}$) and thus there can be multiple source values related to any given target value. This results in a set of backtranslated contexts, of which at least one will reproduce the trace as we need it.

We bypass the lengthy technical setup for this proof and provide an informal description of why the backtranslation achieves what it is supposed to. As an example, Figure 1 contains a trace $\overline{\alpha}$ and the the output of $\langle\!\langle \overline{\alpha} \rangle\!\rangle^{\mathbf{L}^{\mathbf{P}}}_{\mathsf{L}^{\mathsf{U}}}$.

$\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L}^{\mathbf{P}}}_{\mathsf{L}^{\mathsf{U}}}$ first generates empty method bodies for all context methods called by the compiled component. Then it backtranslates each *action* on the given trace, generating code blocks that mimic that action and places that code inside the appropriate method body. Figure 1 shows the code blocks generated for each action. Backtranslated code maintains a support data structure at runtime, a list of locations denoted $\mathsf{L}$ where locations are added ($::$) and they are looked up ($\mathsf{L}(\mathsf{n})$) based on their second field $\mathsf{n}$, which is their target-level address. In order to backtranslate the first call, we need to set up the heap with the right values and then perform the call. In the diagram, dotted lines describe which source statement generates which part of the heap. The return only generates code that will update the list $\mathsf{L}$ to ensure that the context has access to all the locations

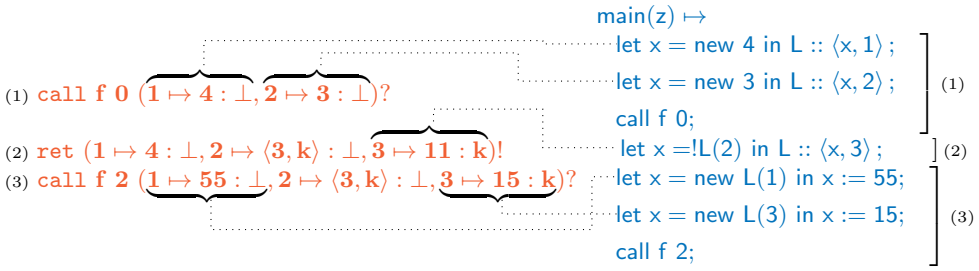


Fig. 1: Example of a trace and its backtranslated code.

it knows in the target too. In order to backtranslate the last call we lookup the locations to be updated in $\mathsf{L}$ so we can ensure that when the $\mathsf{call\ f\ 2}$ statement is executed, the heap is in the right state.

For the backtranslation to be used in the proof we need to prove its correctness, i.e., that $\langle\!\langle \overline{\alpha} \rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}}$ generates a context $\mathsf{A}$ that, together with $\mathsf{C}$, generates a trace $\overline{\alpha}$ related to the given target trace $\overline{\alpha}$.

**Theorem 2** ($\langle\!\langle \cdot \rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}}$ **is correct).**

*if* $\mathbf{A}\left[[\![\mathsf{C}]\!]_{\mathbf{L^P}}^{\mathsf{L^U}}\right] \overset{\overline{\alpha}}{\Longrightarrow} \mathbf{\Omega}$ *then* $\exists \mathsf{A} \in \langle\!\langle \overline{\alpha} \rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}}.\, \mathsf{A}[\mathsf{C}] \overset{\overline{\alpha}}{\Longrightarrow} \Omega$ *and* $\overline{\alpha} \approx_\beta \overline{\alpha}$ *and* $\Omega \approx_\beta \mathbf{\Omega}$.

This theorem immediately implies that $\vdash [\![\cdot]\!]_{\mathbf{L^P}}^{\mathsf{L^U}} : \mathit{PF\text{-}RSC}$, which, by Theorem 3 below, implies that $\vdash [\![\cdot]\!]_{\mathbf{L^P}}^{\mathsf{L^U}} : \mathit{RSC}$.

**Theorem 3** (*PF-RSC* **and** *RSC* **are equivalent for** $[\![\cdot]\!]_{\mathbf{L^P}}^{\mathsf{L^U}}$**).**

$$\vdash [\![\cdot]\!]_{\mathbf{L^P}}^{\mathsf{L^U}} : \mathit{PF\text{-}RSC} \iff \vdash [\![\cdot]\!]_{\mathbf{L^P}}^{\mathsf{L^U}} : \mathit{RSC}$$

*Example 4 (Compiling a secure program).* To illustrate *RSC* at work, let us consider the following source component $\mathsf{C_a}$, which manages an account whose balance is security-relevant. Accordingly, the balance is stored in a location ($\ell_{\mathsf{root}}$ that is tracked by the monitor. $\mathsf{C_a}$ provides functions to deposit to the account as well as to print the account balance.

$$\mathsf{deposit(x)} \mapsto \mathsf{let\ q{=}abs(x)\ in\ let\ amt} = !\ell_{\mathsf{root}}\ \mathsf{in}\ \ell_{\mathsf{root}} := \mathsf{amt} + \mathsf{q}$$
$$\mathsf{balance()} \mapsto !\ell_{\mathsf{root}}$$

$\mathsf{C_a}$ never leaks any sensitive location ($\ell_{\mathsf{root}}$) to an attacker. Additionally, an attacker has no way to decrement the amount of the balance since deposit only adds the absolute value $\mathsf{abs(x)}$ of its input $\mathsf{x}$ to the existing balance.

By compiling $\mathsf{C_a}$ with $[\![\cdot]\!]_{\mathbf{L^P}}^{\mathsf{L^U}}$, we obtain the following target program.

$$\mathbf{deposit(x)} \mapsto \mathbf{let\ q{=}abs(x)\ in}$$
$$\mathbf{let\ amt{=}!0\ with\ k_{root}\ in\ 0 := amt + q\ with\ k_{root}}$$
$$\mathbf{balance()} \mapsto \mathbf{!0\ with\ k_{root}}$$

Recall that location $\ell_{\mathsf{root}}$ is mapped to location $\mathbf{0}$ and protected by the $\mathbf{k_{root}}$ capability. In the compiled code, while location $\mathbf{0}$ is freely computable by a

target attacker, capability $\mathbf{k_{root}}$ is not. Since that capability is not leaked to an attacker, an attacker will not be able to tamper with the balance stored in location $\mathbf{0}$. ⊡

## 4 *RSC* via Bisimulation

If the source language has a verification system that enforces robust safety, proving that a compiler attains *RSC* can be simpler than that of Section 3—it may not require a back translation. To demonstrate this, we consider a specific class of monitors, namely those that enforce type invariants on a specific set of locations. Our source language, $\mathsf{L}^{\tau}$, is similar to $\mathsf{L}^{\mathsf{U}}$ but it has a type system that accepts only those source programs whose traces the source monitor never rejects. Our compiler $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ is directed by typing derivations, and its proof of *RSC* establishes a specific cross-language invariant on program execution, rather than a backtranslation. A second, independent goal of this section is to show that *RSC* is compatible with concurrency. Consequently, our source and target languages include constructs for forking threads.

### 4.1 The Source Language $\mathsf{L}^{\tau}$

$\mathsf{L}^{\tau}$ extends $\mathsf{L}^{\mathsf{U}}$ with concurrency, so it has a fork statement $(\|\ \mathsf{s})$, processes and process soups [19]. Components define a set of safety-relevant locations $\Delta$, so $\mathsf{C} ::= \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}}$ and heaps carry type information, so $\mathsf{H} ::= \varnothing \mid \mathsf{H}; \ell \mapsto \mathsf{v} : \tau$. $\Delta$ also specifies a type for each safety-relevant location, so $\Delta ::= \varnothing \mid \Delta; (\ell : \tau)$.

$\mathsf{L}^{\tau}$ has an unconventional type system that enforces *robust type safety* [34,14,1,31,45,58], which means that no context can cause the static types of sensitive heap locations to be violated at runtime. Using a special type $\mathsf{UN}$ that is described below, a program component statically partitions heap locations it deals with into those it cares about (sensitive or "trusted" locations) and those it does not care about ("untrusted" locations). Call a value *shareable* if only untrusted locations can be extracted from it using the language's elimination constructs. The type system then ensures that a program component only ever shares shareable values with the context. This ensures that the context cannot violate any invariants (including static types) of the trusted locations, since it can never gets direct access to them.

Technically, the type system considers the types $\tau ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \tau \times \tau \mid \mathsf{Ref}\ \tau \mid \mathsf{UN}$ and the following typing judgements ($\Gamma$ maps variables to types).

| | | | |
|---|---|---|---|
| $\vdash \mathsf{C} : \mathsf{UN}$ | Component $\mathsf{C}$ is well-typed. | $\Delta, \Gamma \vdash \mathsf{e} : \tau$ | Expression $\mathsf{e}$ has type $\tau$. |
| $\tau \vdash \circ$ | Type $\tau$ is shareable. | $\mathsf{C}, \Delta, \Gamma \vdash \mathsf{s}$ | Statement $\mathsf{s}$ is well-typed. |

$$\frac{}{\mathsf{Bool} \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-bool-pub}) \qquad \frac{}{\mathsf{Nat} \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-nat-pub}) \qquad \frac{\tau \vdash \circ \quad \tau' \vdash \circ}{\tau \times \tau' \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-pair-pub})$$

$$\frac{}{\mathsf{UN} \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-un-pub}) \qquad \frac{}{\mathsf{Ref}\ \mathsf{UN} \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-references-pub})$$

Type $\mathsf{UN}$ stands for "untrusted" or "shareable" and contains all values that can be passed to the context. Every type that is not a subtype of $\mathsf{UN}$ is implicitly trusted and cannot be passed to the context. Untrusted locations are explicitly marked $\mathsf{UN}$ at their allocation points in the program. Other types are deemed shareable via subtyping. Intuitively, a type is safe if values in it can only yield locations of type $\mathsf{UN}$ by the language elimination constructs. For example, $\mathsf{UN} \times \mathsf{UN}$ is a subtype of $\mathsf{UN}$. We write $\tau \vdash \circ$ to mean that $\tau$ is a subtype of $\mathsf{UN}$.

Further, $\mathsf{L}^{\tau}$ contains an *endorsement* statement ($\mathsf{endorse\ x = e\ as\ \varphi\ in\ s}$) that dynamically checks the top-level constructor of a value of type $\mathsf{UN}$ and gives it a more precise superficial type $\varphi ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{UN} \times \mathsf{UN} \mid \mathsf{Ref\ UN}$ [24]. This allows a program to safely inspect values coming from the context. It is similar to existing type casts [48] but it only inspects one structural layer of the value (this simplifies the compilation).

The operational semantics of $\mathsf{L}^{\tau}$ updates that of $\mathsf{L}^{\mathsf{U}}$ to deal with concurrency and endorsement. The latter performs a runtime check on the endorsed value [62].

Monitors $\mathsf{M} ::= (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_c)$ check at runtime that the set of trusted heap locations $\Delta$ have values of their intended static types. Accordingly, the description of the monitor includes a list of trusted locations and their expected types (in the form of an environment $\Delta$). The type $\tau$ of any location in $\Delta$ must be trusted, so $\tau \not\vdash \circ$. To facilitate checks of the monitor, every heap location carries a type at runtime (in addition to a value). The monitor transitions should therefore be of the form $(\sigma, \Delta, \sigma)$, but since $\Delta$ never changes, we write the transitions as $(\sigma, \sigma)$.

A monitor and a component agree if they have the same $\Delta$: $\mathsf{M} \frown \mathsf{C} \stackrel{\text{def}}{=} (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_c) \frown (\Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}})$. Other definitions (safety, robust safety and actions) are as in Section 2. Importantly, a well-typed component generates traces that are always accepted, so every component typed at $\mathsf{UN}$ is robustly safe.

**Theorem 4 (Typability Implies Robust Safety in $\mathsf{L}^{\tau}$).**

$$\textit{If } \vdash \mathsf{C} : \mathsf{UN} \textit{ and } \mathsf{C} \frown \mathsf{M} \textit{ then } \mathsf{M} \vdash \mathsf{C} : \mathsf{rs}$$

*Richer Source Monitors.* In $\mathsf{L}^{\tau}$, source language monitors only enforce the property of type safety on specific memory locations (robustly). This can be generalized substantially to enforce arbitrary invariants other than types on locations. The only requirement is to find a type system (e.g., based on refinements or Hoare logics) that can enforce robust safety in the source (cf. [68]). Our compilation and proof strategy should work with little modification. Another easy generalization is allowing the set of locations considered by the monitor to grow over time, as in Section 3.

### 4.2 The Target Language $\mathbf{L}^{\pi}$

Our target language, $\mathbf{L}^{\pi}$, extends the previous target language $\mathbf{L}^{\mathbf{P}}$, with support for concurrency (forking, processes and process soups), atomic co-creation of a protected location and its protecting capability ($\mathbf{let\ x = newhide\ e\ in\ s}$) and for examining the top-level construct of a value ($\mathbf{destruct\ x = e\ as\ B\ in\ s\ or\ s'}$) according to a pattern ($\mathbf{B} ::= \mathbf{nat} \mid \mathbf{pair}$).

$$\frac{\mathbf{H} \triangleright \mathbf{e} \rightarrow\!\!\!\!\!\rightarrow \mathbf{n}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{destruct\ x = e\ as\ nat\ in\ s\ or\ s'} \longrightarrow \mathbf{C}, \mathbf{H} \triangleright \mathbf{s}[\mathbf{n} \,/\, \mathbf{x}]}\ (\mathbf{E}\mathbf{L}^{\pi}\text{-destruct-nat})$$

$$\frac{\mathbf{H} = \mathbf{H_1}; \mathbf{n} \mapsto (\mathbf{v}, \eta) \quad \mathbf{H} \triangleright \mathbf{e} \rightarrow\!\!\!\!\!\rightarrow \mathbf{v} \quad \mathbf{k} \notin \mathtt{dom}(\mathbf{H}) \quad \mathbf{s'} = \mathbf{s}[\langle \mathbf{n}+\mathbf{1}, \mathbf{k} \rangle \,/\, \mathbf{x}]}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x = newhide\ e\ in\ s} \longrightarrow \mathbf{C}, \mathbf{H}; \mathbf{n}+\mathbf{1} \mapsto \mathbf{v} : \mathbf{k}; \mathbf{k} \triangleright \mathbf{s'}}\ (\mathbf{E}\mathbf{L}^{\pi}\text{-new})$$

Monitors are also updated to consider a fixed set of locations (a heap $\mathbf{H_0}$), so $\mathbf{M} ::= (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \mathbf{H_0}, \sigma_c)$. The atomic creation of capabilities is provided to match modern security architectures such as Cheri [71] (which implement capabilities at the hardware level). This atomicity is not strictly necessary and we prove that *RSC* is attained both by a compiler relying on it and by one that allocates a location and then protects it non-atomically. The former compiler (with this atomicity in the target) is a bit easier to describe, so for space reasons, we only describe that here and defer the other one to the companion report [61].

### 4.3 Compiler from $\mathsf{L}^{\tau}$ to $\mathbf{L}^{\pi}$

The high-level structure of the compiler, $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$, is similar to that of our earlier compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ (Section 3.3). However, $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ is defined by induction on the type derivation of the component to be compiled. The case for allocation (presented below) explicitly uses type information to achieve security efficiently, protecting only those locations whose type is not UN.

$$\left[\!\!\left[ \frac{\Delta, \Gamma \vdash \mathsf{e} : \tau \quad \mathsf{C}, \Delta, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}}{\mathsf{C}, \Delta, \Gamma \vdash \mathsf{let\ x = new}_{\tau}\ \mathsf{e\ in\ s}} \right]\!\!\right]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \begin{cases} \mathbf{let\ xo} = \mathbf{new}\ [\![\Delta, \Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \quad \mathbf{in\ let\ x} = \langle \mathbf{xo}, \mathbf{0} \rangle \\ \quad\ \mathbf{in}\ [\![\mathsf{C}, \Delta, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} & \text{if } \tau = \mathsf{UN} \\ \\ \mathbf{let\ x} = \mathbf{newhide}\ [\![\Delta, \Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \quad \mathbf{in}\ [\![\mathsf{C}, \Delta, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} & \text{otherwise} \end{cases}$$

*New Monitor Relation.* As monitors have changed, we also need a new monitor relation $\mathsf{M} \approx \mathbf{M}$. Informally, a source and a target monitor are related if the target monitor can always step whenever the target heap satisfies the types specified in the source monitor (up to renaming by the partial bijection $\beta$).

We write $\vdash \mathsf{H} : \Delta$ to mean that for each location $\ell \in \Delta$, $\vdash \mathsf{H}(\ell) : \Delta(\ell)$. Given a partial bijection $\beta$ from source to target locations, we say that a target monitor $\mathbf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \mathbf{H_0}, \sigma_c)$ is good, written $\vdash \mathbf{M} : \beta, \Delta$, if for all $\sigma \in \{\sigma \cdots\}$ and all $\mathsf{H} \approx_{\beta} \mathbf{H}$ such that $\vdash \mathsf{H} : \Delta$, there is a $\sigma'$ such that $(\sigma, \mathbf{H}, \sigma') \in \rightsquigarrow$. For a fixed partial bijection $\beta_0$ between the domains of $\Delta$ and $\mathbf{H_0}$, we say that the source monitor $\mathsf{M}$ and the target monitor $\mathbf{M}$ are related, written $\mathsf{M} \approx \mathbf{M}$, if $\vdash \mathbf{M} : \beta_0, \Delta$ for the $\Delta$ in $\mathsf{M}$. With this setup, we define *RSC* as in Section 2.

**Theorem 5 (Compiler $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ attains *RSC*).** $\vdash [\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} : \mathit{RSC}$

To prove that $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ attains *RSC* we do not rely on a backtranslation. Here, we know statically which locations can be monitor-sensitive: they must all be

trusted, i.e., must have a type $\tau$ satisfying $\tau \not\vdash \circ$. Using this, we set up a simple cross-language relation and show it to be an invariant on runs of source and compiled target components. The relation captures the following:

- Heaps (both source and **target**) can be partitioned into two parts, a *trusted* part and an *untrusted* part;
- The trusted source heap contains only locations whose type is trusted ($\tau \not\vdash \circ$);
- The trusted **target heap** contains only locations related to trusted source locations and these point to related values; more importantly, every **trusted target location** is protected by a capability;
- In the **target**, any capability protecting a trusted location does not occur in attacker code, nor is it stored in an untrusted heap location.

We need to prove that this relation is preserved by reductions both in compiled and in attacker code. The former follows from source robust safety (Theorem 4). The latter is simple since all trusted locations are protected with capabilities, attackers have no access to trusted locations, and capabilities are unforgeable and unguessable (by the semantics of $\mathbf{L}^{\pi}$). At this point, knowing that monitors are related, and that source traces are always accepted by source monitors, we can conclude that target traces are always accepted by target monitors too. Note that this kind of an argument requires all compilable source programs to be robustly safe and is, therefore, impossible for our first compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$. Avoiding the backtranslation results in a proof much simpler than that of Section 3.

## 5 Fully Abstract Compilation

Our next goal is to compare *RSC* to *FAC* at an intuitive level. We first define fully abstract compilation or *FAC* (Section 5.1). Then, we present an example of how *FAC* may result in inefficient compiled code and use that to present in Section 5.2 what would be needed to write a fully abstract compiler from $\mathsf{L}^{\mathsf{U}}$ to $\mathbf{L}^{\mathbf{P}}$ (the languages of our first compiler). We use this example to compare *RSC* and *FAC* concretely, showing that, at least on this example, *RSC* permits more efficient code and affords simpler proofs that *FAC*.

However, this does not imply that one should always prefer *RSC* to *FAC* blindly. In some cases, one may want to establish full abstraction for reasons other than security. Also, when the target language is typed [11,10,21,50] or has abstractions similar to those of the source, full abstraction may have no downsides (in terms of efficiency of compiled code and simplicity of proofs) relative to *RSC*. However, in many settings, including those we consider, target languages are not typed, and often differ significantly from the source in their abstractions. In such cases, *RSC* is a worthy alternative.

### 5.1 Formalising Fully Abstract Compilation

As stated in Section 1, *FAC* requires the preservation and reflection of observational equivalence, and most existing work instantiates observational equiva-

lence with contextual equivalence ($\simeq_{ctx}$). Contextual equivalence and *FAC* are defined below. Informally, two components $C_1$ and $C_2$ are contextually equivalent if no context $A$ interacting with them can tell them apart, i.e., they are *indistinguishable*. Contextual equivalence can encode security properties such as confidentiality, integrity, invariant maintenance and non-interference [53,60,9,6]. We do not explain this well-known observation here, but refer the interested reader to the survey of Patrignani *et al.* [54]. Informally, a compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ is fully abstract if it translates (only) contextually-equivalent source components into contextually-equivalent target ones.

**Definition 4 (Contextual equivalence and fully abstract compilation).**

$$C_1 \simeq_{ctx} C_2 \stackrel{\text{def}}{=} \forall A.\, A[C_1]\Uparrow \iff A[C_2]\Uparrow, \text{ where } \Uparrow \text{ means execution divergence}$$

$$\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : FAC \stackrel{\text{def}}{=} \forall \mathsf{C}_1, \mathsf{C}_2.\, \mathsf{C}_1 \simeq_{ctx} \mathsf{C}_2 \iff [\![\mathsf{C}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{C}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$$

The security-relevant part of *FAC* is the $\Rightarrow$ implication [29]. This part is security-relevant because the proof thesis concerns target contextual equivalence ($\simeq_{ctx}$). Unfolding the definition of $\simeq_{ctx}$ on the right of the implication yields a universal quantification over all possible target contexts $\mathbf{A}$, which captures malicious attackers. In fact, there may be target contexts $\mathbf{A}$ that can interact with compiled code in ways that are impossible in the source language. Compilers that attain *FAC* with untyped target languages often insert checks in compiled code that detect such interactions and respond to them securely [60], often by halting the execution [9,53,29,39,37,6,42,54]. These checks are often inefficient, but must be performed even if the interactions are not security-relevant. We now present an example of this.

*Example 5 (Wrappers for heap resources).* Consider a password manager written in an object-oriented language that is compiled to an assembly-like language. The password manager defines a `private List` object where it stores the passwords locally. Shown below are two implementations of the `newList` method inside `List` which we call $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$. The only difference between $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$ is that $\mathsf{C}_{\mathsf{two}}$ allocates two lists internally; one of these (`shadow`) is used for internal purposes only.

```
1 public newList(): List{
2
3   ell = new List();
4   return ell;
5 }
```

```
1 public newList(): List{
2   shadow = new List(); // diff
3   ell = new List();
4   return ell;
5 }
```

$\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$ are equivalent in a source language that does not allow pointer comparison (like our source languages). To attain *FAC* when the target allows pointer comparisons (as in our target languages), the pointers returned by `newList` in the two implementations must be the same, but this is very difficult to ensure since the second implementation does more allocations. A simple solution to this problem is to wrap `ell` in a proxy object and return the proxy [53,59,9,47]. Compiled code needs to maintain a lookup table mapping

the proxy to the original object and proxies must have allocation-independent addresses. Proxies work but they are inefficient due to the need to look up the table on every object access. ⊡

In this example, *FAC* forces all privately allocated locations to be wrapped in proxies. However, *RSC* does not require this. Our target languages $\mathbf{L}^{\mathbf{P}}$ and $\mathbf{L}^{\pi}$ support address comparison (addresses are natural numbers in their heaps) but $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ and $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ just use capabilities to attain security efficiently while $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\mathbf{I}}}$ relies on memory isolation. On the other hand, for attaining *FAC*, capabilities alone would be insufficient since they do not hide addresses. We explain this in detail in the next subsection.

*Remarks.* Our technical report lists many other cases of *FAC* forcing security-irrelevant inefficiency in compiled code [61]. All of these can be avoided by just replacing contextual equivalence with a different notion of equivalence in the statement of *FAC*. However, it is not clear how this can be done generally for any given kind of inefficiency, and what the security consequences of such instantiations of the statement of *FAC* are. On the other hand, *RSC* is *uniform* and it does not induce any of these inefficiencies.

A security issue that cannot be addressed just by tweaking equivalences is information leaks on side channels, as side channels are, by definition, not expressible in the language. Neither *FAC* nor *RSC* deals with side channels.

### 5.2 Towards a Fully Abstract Compiler from $\mathsf{L}^{\mathsf{U}}$ to $\mathbf{L}^{\mathbf{P}}$

To further compare *FAC* and *RSC*, we now sketch what *would* be needed to construct a fully abstract compiler from $\mathsf{L}^{\mathsf{U}}$ to $\mathbf{L}^{\mathbf{P}}$. In particular, this compiler should not suffer from the "attack" described in Example 5.

*Inefficiency.* We denote with $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ a (hypothetical) new compiler from $\mathsf{L}^{\mathsf{U}}$ to $\mathbf{L}^{\mathbf{P}}$ that attains *FAC*. We describe informally what code generated by this compiler would have to do. We know that fully abstract compilation preserves *all* source abstractions in the target language. One abstraction that distinguishes $\mathbf{L}^{\mathbf{P}}$ from $\mathsf{L}^{\mathsf{U}}$ is that locations are abstract in $\mathbf{L}^{\mathbf{P}}$, but concrete natural numbers in $\mathsf{L}^{\mathsf{U}}$. Thus, locations allocated by compiled code must not be passed directly to the context as this would reveal the allocation order. Instead of passing the location $\langle \mathbf{n}, \mathbf{k} \rangle$ to the context, the compiler arranges for an opaque handle $\langle \mathbf{n}', \mathbf{k}_{\mathbf{com}} \rangle$ (that cannot be used to access any location directly) to be passed. Such an opaque handle is often called a *mask* or *seal* in the literature [66].

To ensure that masking is done properly, $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ can insert code at entry and exit points of compiled code, *wrapping* the compiled code in a way that enforces masking [32,59]. The wrapper keeps a list $\overline{\mathbf{L}}$ of component-allocated locations that are shared with the context in order to know their masks. When a component-allocated location is shared, it is added to the list $\overline{\mathbf{L}}$. The mask of a location is its index in this list. If the same location is shared again it is not added again but its previous index is used. To implement lookup in $\overline{\mathbf{L}}$ we must compare capabilities too, so we need to add that expression to the target language. To

ensure capabilities do not leak to the context, the second field of the pair is a constant capability $\mathbf{k_{com}}$ which compiled code does not use otherwise. Clearly, this wrapping can increase the cost of all cross-component calls and returns.

However, this wrapping is not sufficient to attain *FAC*. A component-allocated location could be passed to the context on the heap, so before passing control to the context the compiled code needs to *scan the whole heap* where a location can be passed and mask all found component-allocated locations. Dually, when receiving control the compiled code must scan the heap to unmask any masked location so it can use the location. The problem now is determining what parts of the heap to scan and how. Specifically, the compiled code needs to keep track of all the locations (and related capabilities) that are shared, i.e., (i) passed from the context to the component and (ii) passed from the component to the context. Both keeping track of these locations as well as scanning them on every cross-component control transfer is likely to be *very* expensive.

Finally, masked locations cannot be used directly by the context to be read and written. Thus, compiled code must provide a **read** and a **write** function that implement reading and writing to masked locations. The additional unmasking in these functions (as opposed to native reads and writes) adds to the inefficiency.

It should be clear as opposed to the *RSC* compiler $[\![\cdot]\!]^{\mathbf{L}^{\mathbf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ (Section 3), the *FAC* compiler $[\![\boxed{\cdot}]\!]^{\mathbf{L}^{\mathbf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ just sketched is likely to generate far more inefficient code.

*Proof difficulty.* Proving that $[\![\boxed{\cdot}]\!]^{\mathbf{L}^{\mathbf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ attains *FAC* can only be done by backtranslating *traces*, not contexts alone, since the newly-added target expressions cannot be directly backtranslated to valid source ones [9,59,7]. For this, we need a trace semantics that captures all information available to the context. This is often called a fully abstract trace semantics [38,56,55]. However, the trace semantics we defined for $\mathbf{L}^{\mathbf{P}}$ is not fully abstract, as its actions record the entire heap in every action, including private parts of the heap. Hence, we cannot use this trace semantics for proving *FAC* and so we design a new one. Building a fully abstract trace semantics for $\mathbf{L}^{\mathbf{P}}$ is challenging because we have to keep track of locations that have been shared with the context in the past. This substantially complicates both the definition of traces and the proofs that build on the definition.

Finally, the source context that the backtranslation constructs from a target trace must simulate the shared part of the heap at every context switch. Since locations in the target may be masked, the source context has to maintain a map from the source locations to the corresponding masked target ones, which complicates the backtranslation and the proof substantially.

To summarize, it should be clear that the proof of *FAC* for $[\![\boxed{\cdot}]\!]^{\mathbf{L}^{\mathbf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ would be much harder than the proof of *RSC* for $[\![\cdot]\!]^{\mathbf{L}^{\mathbf{U}}}_{\mathbf{L}^{\mathbf{P}}}$, even though the source and target languages are the same and so is the broad proof technique (backtranslation).

## 6 Related Work

Recent work [33,8] presents new criteria for secure compilation that ensure preservation of subclasses of hyperproperties. Hyperproperties [25] are a formal representation of predicates on programs, i.e., they are predicates on sets of traces. Hyperproperties capture many security-relevant properties including not just conventional safety and liveness, which are predicates on traces, but also properties like non-interference, which is a predicate on pairs of traces. Modulo technical differences, our definition of *RSC* coincides with the criterion of "robust safety property preservation" in [33,8]. We show, through concrete instances, that this criterion can be easily realized by compilers, and develop two proof techniques for establishing it. We further show that the criterion leads to more efficient compiled code than does *FAC*. Additionally, the criteria in [33,8] assume that behaviours in the source and target are represented using the same alphabet. Hence, the definitions (somewhat unrealistically or ideally) do not require a translation of source properties to target properties. In contrast, we consider differences in the representation of behaviour in the source and in the target and this is accounted for in our monitor relation $\mathsf{M} \approx \mathbf{M}$. A slightly different account of this difference is presented by Patrignani and Garg [60] in the context of reactive black-box programs.

Abate *et al.* [7] define a variant of robustly-safe compilation called RSCC specifically tailored to the case where (source) components can perform undefined behaviour. RSCC does not consider attacks from arbitrary target contexts but from compiled components that can become compromised and behave in arbitrary ways. To demonstrate RSCC, Abate *et al.* [7] rely on two backends for their compiler: software fault isolation and tag-based monitors. On the other hand, we rely on capability machines and memory isolation (the latter in the companion report). RSCC also preserves (a form of) safety properties and can be achieved by relying on a trace-based backtranslation; it is unclear whether proofs can be simplified when the source is verified and concurrent, as in our second compiler.

ASLR [6,37], protected module architectures [9,53,59,42], tagged architectures [39], capability machines [69] and cryptographic primitives [4,5,22,26] have been used as targets for *FAC*. We believe all of these can also be used as targets of *RSC*-attaining compilers. In fact, some targets such as capability machines seem to be better suited to *RSC* than *FAC*, as we demonstrated.

Ahmed *et al.* prove full abstraction for several compilers between typed languages [11,10,50]. As compiler intermediate languages are often typed, and as these types often serve as the basis for complex static analyses, full abstraction seems like a reasonable goal for (fully typed) intermediate compilation steps. In the last few steps of compilation, where the target languages are unlikely to be typed, one could establish robust safety preservation and combine the two properties (vertically) to get an end-to-end security guarantee.

There are three other criteria for secure compilation that we would like to mention: securely compartmentalised compilation (SCC) [39], trace-preserving compilation (TPC) [60] and non-interference-preserving compilation (NIPC) [16,27,12,15].

SCC is a re-statement of the "hard" part of full abstraction (the forward implication), but adapted to languages with undefined behaviour and a strict notion of components. Thus, SCC suffers from much of the same efficiency drawbacks as *FAC*. TPC is a stronger criterion than *FAC*, that most existing fully abstract compilers also attain. Again, compilers attaining TPC also suffer from the drawbacks of compilers attaining *FAC*.

NIPC preserves a single property: noninterference (NI). However, this line of work does not consider active target-level adversaries yet. Instead, the focus is on compiling whole programs. Since noninterference is not a safety property, it is difficult to compare NIPC to *RSC* directly. However, noninterference can also be approximated as a safety property [20]. So, in principle, *RSC* (with adequate massaging of observations) can be applied to stronger end-goals than NIPC.

Swamy *et al.* [67] embed an F* model of a gradually and robustly typed variant of JavaScript into an F* model of JavaScript. Gradual typing supports constructs similar to our endorsement construct in $\mathsf{L}^{\tau}$. Their type-directed compiler is proven to attain memory isolation as well as static and dynamic memory safety. However, they do not consider general safety properties, nor a specific, general criterion for compiler security.

Two of our target languages rely on capabilities for restricting access to sensitive locations from the context. Although capabilities are not mainstream in any processor, fully functional research prototypes such as Cheri exist [71]. Capability machines have previously been advocated as a target for efficient secure compilation [30] and preliminary work on compiling C-like languages to them exists, but the criterion applied is *FAC* [69].

## 7 Conclusion

This paper has examined robustly safe compilation (*RSC*), a soundness criterion for compilers with direct relevance to security. We have shown that the criterion is easily realizable and may lead to more efficient code than does fully abstract compilation wrt contextual equivalence. We have also presented two techniques for establishing that a compiler attains *RSC*. One is an adaptation of an existing technique, backtranslation, and the other is based on inductive invariants.

**Acknowledgements.** The authors would like to thank Dominique Devriese, Akram El-Korashy, Cătălin Hriţcu, Frank Piessens, David Swasey and the anonymous reviewers for useful feedback and discussions on an earlier draft.

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

## References

1. Martín Abadi. Secrecy by typing in security protocols. In *TACS*, pages 611–638, 1997.

2. Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, London, UK, 1999.
3. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, 2009.
4. Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 302–315, New York, NY, USA, 2000. ACM.
5. Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174:37–83, 2002.
6. Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Transactions on Information and System Security*, 15:8:1–8:29, July 2012.
7. Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. CCS '18, 2018.
8. Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. arXiv:1807.04603, July 2018.
9. Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium*, CSF 2012, pages 171–185. IEEE, 2012.
10. Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 157–168, New York, NY, USA, 2008. ACM.
11. Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 431–444, New York, NY, USA, 2011. ACM.
12. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *ACM Conference on Computer and Communications Security*, pages 1807–1823. ACM, 2017.
13. Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.
14. Michael Backes, Catalin Hritcu, and Matteo Maffei. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security*, 22(2):301–353, 2014.
15. Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". CSF'18, 2018.
16. Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security types preserving compilation. *Computer Languages, Systems and Structures*, 33:35–59, 2007.
17. Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, February 2011.
18. Nick Benton and Chung-kil Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical report, MSR, 2010.

19. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.
20. Gérard Boudol. Secure information flow as a safety property. chapter Formal Aspects in Security and Trust, pages 20–34. Springer-Verlag, Berlin, Heidelberg, 2009.
21. William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, New York, NY, USA, 2015. ACM.
22. Michele Bugliesi and Marco Giunti. Secure implementations of typed channel abstractions. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 251–262, New York, NY, USA, 2007. ACM.
23. Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. *SIGPLAN Not.*, 29:319–327, 1994.
24. Stephen Chong. *Expressive and Enforceable Information Security Policies.* PhD thesis, Cornell University, August 2008.
25. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.
26. Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16:573–636, 2008.
27. David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for C and assembly programs. In *PLDI*, pages 648–664. ACM, 2016.
28. Dominique Devriese, Marco Patrignani, Steven Keuchel, and Frank Piessens. Modular, Fully-Abstract Compilation by Approximate Back-Translation. *Logical Methods in Computer Science*, Volume 13, Issue 4, October 2017.
29. Dominique Devriese, Marco Patrignani, and Frank Piessens. Secure Compilation by Approximate Back-Translation. In *POPL 2016*, 2016.
30. Akram El-Korashy. A Formal Model for Capability Machines – An Illustrative Case Study towards Secure Compilation to CHERI. Master's thesis, Universitat des Saarlandes, 2016.
31. Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.
32. Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 371–384, New York, NY, USA, 2013. ACM.
33. D. Garg, C. Hritcu, M. Patrignani, M. Stronati, and D. Swasey. Robust Hyperproperty Preservation for Secure Compilation (Extended Abstract). *ArXiv e-prints*, October 2017.
34. Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *J. Comput. Secur.*, 11(4):451–519, July 2003.
35. Daniele Gorla and Uwe Nestman. Full abstraction for expressiveness: History, myths and facts. *Math Struct Comp Science*, 2014.
36. Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and Assembly. *SIGPLAN Not.*, 46:133–146, January 2011.
37. Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. Local memory via layout randomization. In *Proceedings of the 2011 IEEE 24th Computer Security*

*Foundations Symposium*, CSF '11, pages 161–174, Washington, DC, USA, 2011. IEEE Computer Society.

38. Alan Jeffrey and Julian Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *ESOP'05*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.
39. Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *29th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, July 2016. To appear.
40. Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. POPL 2016, pages 178–190, 2016.
41. Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.
42. Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. A secure compiler for ML modules. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 29–48, 2015.
43. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.
44. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.
45. Sergio Maffeis, Martín Abadi, Cédric Fournet, and Andrew D. Gordon. *Code-Carrying Authorization*, pages 563–579. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
46. Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1. ACM, 2013.
47. James H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16:15–21, 1973.
48. Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. *SIGPLAN Not.*, 44(9):135–148, August 2009.
49. Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 166–178. ACM, 2015.
50. Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 103–116, New York, NY, USA, 2016. ACM.
51. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, USA, 1999.
52. Joachim Parrow. General conditions for full abstraction. *Math Struct Comp Science*, 2014.
53. Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.*, 37:6:1–6:50, April 2015.

54. Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation a survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, January 2019.
55. Marco Patrignani and Dave Clarke. Fully abstract trace semantics of low-level isolation mechanisms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1562–1569. ACM, 2014.
56. Marco Patrignani and Dave Clarke. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures*, 42(0):22 – 45, 2015.
57. Marco Patrignani, Dave Clarke, and Frank Piessens. Secure compilation of object-oriented components to protected module architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)*, volume 8301 of *LNCS*, pages 176–191, 2013.
58. Marco Patrignani, Dave Clarke, and Davide Sangiorgi. Ownership Types for the Join Calculus. In *FMOODS/FORTE 2011*, volume 6722 of *LNCS*, pages 289–303, 2011.
59. Marco Patrignani, Dominique Devriese, and Frank Piessens. On Modular and Fully Abstract Compilation. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium*, CSF 2016, 2016.
60. Marco Patrignani and Deepak Garg. Secure Compilation and Hyperproperties Preservation. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium CSF 2017, Santa Barbara, USA*, CSF 2017, 2017.
61. Marco Patrignani and Deepak Garg. Robustly safe compilation or, efficient, provably secure compilation. *CoRR*, abs/1804.00489, 2018.
62. Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, October 2009.
63. Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
64. Ian Stark. *Names and Higher-Order Functions.* PhD thesis, University of Cambridge, December 1994. Also available as Technical Report 363, University of Cambridge Computer Laboratory.
65. Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 275–287, New York, NY, USA, 2015. ACM.
66. Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *Principles of Programming Languages*, pages 161–172, 2004.
67. Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. *SIGPLAN Not.*, 49(1):425–437, January 2014.
68. David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017. October 22 - 27, 2017*, 2017.
69. Stelios Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. Towards Automatic Compartmentalization of C Programs on Capability Machines. In *Workshop on Foundations of Computer Security 2017 August 21, 2017*, FCS 2017, 2017.
70. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, 1996.

71. Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.
72. Stephan Arthur Zdancewic. *Programming Languages for Information Security.* PhD thesis, Cornell University, 2002.