Endangered by the Language But Saved by the Compiler: Robust Safety via Semantic Back-Translation

NIKLAS MÜCK, MPI-SWS, Germany
AÏNA LINN GEORGES, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany
DEEPAK GARG, MPI-SWS, Germany
MICHAEL SAMMLER, Institute of Science and Technology Austria (ISTA), Austria

It is common for programmers to assemble their programs from a combination of trusted and untrusted components. In this context, a trusted program component is said to be *robustly safe* if it behaves safely when linked against arbitrary untrusted code. Prior work has shown how various encapsulation mechanisms (in both high- and low-level languages) can be used to protect code so that it is robustly safe, but none of the existing work has explored how robust safety can be achieved in a patently *unsafe* language like C.

In this paper, we show how to bring robust safety to a simple yet representative C-like language we call Rec. Although Rec (like C) is inherently "dangerous" and thus not robustly safe, we can "save" Rec programs via compilation to Cap, a CHERI-like *capability machine*. To formalize the benefits of such a *hardening compiler*, we develop Reckon, a separation logic for verifying robust safety of Rec programs. Reckon is *not* sound under Rec's unsafe, C-like semantics, but it *is* sound when Rec programs are hardened via compilation and linked against untrusted code running on Cap. As a crucial step in proving soundness of Reckon, we introduce a novel technique of *semantic back-translation*, which we formalize by building on the DimSum framework for multi-language semantics. All our results are mechanized in the Rocq prover.

CCS Concepts: • **Security and privacy → Logic and verification**; • **Theory of computation → Separation logic**.

Additional Key Words and Phrases: Secure Compilation, Hardening Compilation, Robust Safety, DimSum, Iris

ACM Reference Format:
Niklas Mück, Aïna Linn Georges, Derek Dreyer, Deepak Garg, and Michael Sammler. 2026. Endangered by the Language But Saved by the Compiler: Robust Safety via Semantic Back-Translation. *Proc. ACM Program. Lang.* 10, POPL, Article 40 (January 2026), 30 pages. https://doi.org/10.1145/3776682

1 Introduction

It is common for programmers to assemble their programs from a combination of trusted and untrusted components. When a trusted module T is linked against a module U from an untrusted source, how can one ensure that U does not violate the invariants that T maintains on its internal data representation, so that T continues to behave safely and correctly?

There are several, well-known approaches to answering this question, which employ a variety of built-in *encapsulation* mechanisms present in different (high- and low-level) programming

Authors' Contact Information: Niklas Mück, MPI-SWS, Saarland Informatics Campus, Germany, mueck@mpi-sws.org; Aïna Linn Georges, MPI-SWS, Saarland Informatics Campus, Germany, algeorges@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Deepak Garg, MPI-SWS, Saarland Informatics Campus, Germany, dg@mpi-sws.org; Michael Sammler, Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria, michael.sammler@ista.ac.at.

ACM 2475-1421/2026/1-ART40
https://doi.org/10.1145/3776682

```
1 bool password_check() {
2   int pwd = 0; int usr = 0;
3   read_hash_from_db(&pwd);
4   let saved = pwd; ■
5   adv_io(&usr);
6   assert(pwd == saved); ■
7   return hash(usr) == pwd;
8 }
```

Fig. 1. Password check example

languages. For example, when linking T against untrusted code in a safe, higher-level language, one can use *object capability patterns*, which "wrap" all objects passed from T to untrusted code so that the latter does not have direct access to T's private state [29, 42, 64]. Alternatively, if one is linking against low-level untrusted code, one can physically *sandbox* the untrusted code so that it can only manipulate memory locations in its own "compartment" [68, 37, 73, 53], or one can use *capability machines* to enforce fine-grained memory isolation at the hardware level [71].

Prior work has shown how the semantic benefits of all these various encapsulation mechanisms can be understood in terms of the concept of *robust safety* [23, 22, 16, 64]. Given some notion of "safety" that one is interested in—*e.g.*, absence of undefined behavior—a module T is said to be "robustly safe" if it remains safe when linked against arbitrary untrusted modules U. Object capability patterns, sandboxes, and capability machines are all tools that enable one to take a module T that is safe on its own, and encapsulate it so that it becomes robustly safe [64, 53, 18].

In all of the aforementioned prior work, since robust safety is enforced via built-in encapsulation mechanisms of the (high- or low-level) language under consideration, it can be characterized as a "contextual" syntactic property—*i.e.*, a property that holds when a module in the language is linked against any possible syntactic program context in the same language. By so reducing robust safety to a contextual syntactic property on programs, one can then establish it using standard proof methods for compositional program verification, such as logical relations and program logics. Indeed, that is exactly what Swasey et al. [64], Sammler et al. [53], and Georges et al. [18] all do.

1.1 Robust Safety for an Unsafe, C-Like Language via Hardening Compilation

In this paper, we explore how robust safety can also be achieved in an *unsafe, C-like language*. Such a language poses a fundamental challenge: we cannot simply formulate robust safety in this setting as a contextual syntactic property because the syntactic program contexts of a C-like language can incur *undefined behavior* and are thus *too powerful*. To illustrate the problem, consider the simple `password_check` function shown in Fig. 1. (Technically, this is written in an idealized version of C we call Rec; the formal semantics of Rec is given in §2.)

The function `password_check` first calls the *trusted* function `read_hash_from_db` to read the correct password hash into `pwd` (line 3). (Ignore the two `gray` ■ lines, line 4 and line 6, for now.) Then, it calls an *untrusted* IO function (`adv_io`) to read some input from the user (line 5). Finally, it computes the hash of the user input, compares it against the hash of the correct password from the database, and returns a Boolean signalling whether they match (line 7).

We would like to be able to prove that `password_check` is robustly safe against an arbitrary untrusted implementation of `adv_io`. By "safe" here, we mean that `password_check` should conduct a *bona fide* password check, regardless of how `adv_io` is implemented. Towards that end, it is essential that `adv_io` should not be able to modify `pwd`, since otherwise it could just store the hash of the user input into `pwd` and force the subsequent check on line 7 to trivially succeed. To formalize this safety condition, we save the value of pwd into the variable saved (line 4) and assert after the call to adv_io that it did not change (line 6).[1] We would like to be able to prove that this assertion holds for all possible implementations of adv_io.

Unfortunately, it doesn't! Concretely, consider the following implementation:

```
void adv_io(int* usr) { *usr = 0; *(usr + 1) = hash(0); }
```

This function sets the input to 0, then guesses that pwd is located adjacent to usr in memory and uses pointer arithmetic to override pwd with the hash of 0, making the password check succeed. While it is not guaranteed how variables are laid out in memory, this attack in fact succeeds when password_check and adv_io are compiled by GCC[2].

Technically, this attack works since the out-of-bounds access to (usr + 1) in adv_io incurs *undefined behavior*. Concretely, Rec (like C) does not give a defined semantics to out-of-bounds memory accesses, but rather allows the program to behave arbitrarily when they occur. This undefined behavior allows adv_io to modify the local variable pwd of password_check, even though adv_io does not have direct access to it.

At first glance, it thus appears that in an unsafe, C-like language, we simply cannot write robustly safe code due to the inherent insecurity of the language, which gives syntactic program contexts (such as the naughty implementation of adv_io shown above) too much adversarial power. But in fact we *can* recover robust safety—by leveraging a *hardening compiler*. A hardening compiler is one that introduces security measures during compilation in order to make the target of compilation more secure than the original source program.

There are many ways in which a compiler can harden code against an attacker [2, 36, 44, 68]; in this paper, we focus on hardening via compilation to a *capability machine*. As mentioned above, capability machines are a new type of hardware architecture providing fine-grained memory protection. This protection is achieved by enriching pointers with capabilities that are used by the hardware to dynamically control access to privileged code paths and regions of memory. In particular, by compiling Rec to a capability machine in the right way (see §5), we can prevent the attack of adv_io on password_check by ensuring that the capability to access pwd is never passed to adv_io. Thus, if (as shown above) adv_io attempts to access pwd vicariously by offsetting from the usr address that *is* passed to it, the capability machine will flag this as an illegal memory access and halt the program safely. More generally, such a hardening compiler should allow us to prove that the assertion on line 6 always succeeds when password_check is linked against any program context.

At least, that is the intuition. But how can we formalize it?

1.2 How Our Goal Differs from Secure Compilation

A natural starting point is the growing body of work on *secure compilation* [1, 3, 48, 15, 62, 6, 47, 12, 46, 49, 50, 14, 4, 30, 5, 65, 8, 9]. Broadly speaking, the goal of secure compilation is to develop formal foundations for compilers that *preserve* key security properties of the source program throughout compilation, in particular when the resulting compiled code is linked with untrusted code in the target language of the compiler. The key word here is "preserve": secure compilation *assumes* that some property of interest holds for source-language programs, and aims to show that it *still* holds after compilation to the target language. Examples include *full abstraction* [1] and *robust safety preservation* [4, 50].

[1]Note that we use a let-binding for saved to make sure that it is not stored in memory and cannot be affected by the attacker. Concretely, such let-bindings have a substitution-based semantics in Rec (§2).
[2]GCC 15.1 on Linux x86_64 with default options

On the one hand, our goal in this paper is closely related to that of secure compilation in that we, too, aim to reason about the safety of compiled code linked with arbitrary untrusted target-language code. On the other hand, our goal is fundamentally different in that:

- We are concerned with a C-like source language, Rec, that does *not* enjoy any useful robust safety properties to begin with—it is inherently "dangerous", for example incurring undefined behavior at out-of-bounds memory accesses. So there is no useful property for the compiler to preserve, and the basic assumption of secure compilation does not apply.
- We want to formalize how a hardening compiler for Rec can nevertheless "save" us by *guaranteeing* robust safety for Rec modules under certain conditions. For example, we want to be able to establish that `password_check` is robustly safe when compiled with such a hardening compiler—even though it passes the address of a local variable (`usr`) to untrusted code—because it never passes the address of the secret local variable `pwd`.

To sharpen this distinction, we briefly mention some important prior work on secure compilation, and refer the reader to §8 for further details. In particular, the line of work on compartmentalizing compilation [30, 5, 65] explores secure compilation for an unsafe, C-like source language. However, in order to ensure that source-language programs enjoy a useful property to be preserved, their approach relies crucially on extending the source language with *compartments*, which prevent undefined behavior in an untrusted module from propagating to trusted modules. Furthermore, they do not allow pointer-passing between trusted and untrusted code, and thus cannot account for our `password_check` example. Other researchers [15, 14, 9] have developed secure compilation frameworks that *do* permit pointers to be passed between trusted and untrusted code. However, the source languages of their compilers are safe (*i.e.*, do not have undefined behavior).

1.3 Our Contribution: Formalizing How Hardening Compilation Enables Robust Safety

In this paper, we present the first formal framework for establishing how hardening compilation enables robustly safe reasoning in a C-like language that does not *a priori* support such reasoning.

The setup. Concretely, we consider a simple compiler, Rec2Cap: its source language is Rec, an idealized, C-like language with a CompCert-style block-based memory model [39, 38], and its target language is **Cap**, an idealized yet representative capability machine architecture based on Cerise [18]. The Rec2Cap compiler hardens Rec programs by transforming the pointers of Rec into capabilities: unforgeable fat pointers that enable dynamic tracking (by the **Cap** machine) of which memory region the pointer is permitted to access. The details of how Rec2Cap achieves this hardening are fairly straightforward (see §5) and are not a significant contribution of this paper.

High-level structure of our framework. We aim to prove that the Rec2Cap compiler enables a form of robust safety for Rec modules. Moreover, we want to make it possible for Rec programmers to reason about robust safety of their code at a high level of abstraction, *without* having to understand the details of Rec2Cap or the **Cap** machine. Therefore, we present our framework in two stages:

(1) We first present **Reckon**, a separation logic based on Iris [33, 35, 32] and OCPL [64], which provides a high-level method for verifying robust safety of Rec modules. Reasoning in Reckon requires no knowledge of Rec2Cap or the **Cap** machine.
(2) We then present **RobustDimSum**, a semantic framework for establishing the soundness of Reckon. Crucially, RobustDimSum does *not* establish that Reckon is sound under the unsafe, C-like semantics of Rec—rather, it establishes that Reckon is sound for Rec modules that are compiled by Rec2Cap and linked against untrusted code running on the **Cap** machine.

Reckon: A program logic for robust safety of Rec modules. Reckon is directly inspired by prior logics for robust safety, in particular OCPL [64]. As with OCPL, the most interesting aspect of Reckon is how it supports safe interaction with untrusted code. In order to formalize this interaction, Reckon (like OCPL) distinguishes between *high* and *low* values. *High* values are fully controlled by the trusted module T we are verifying, whereas *low* values are those that may be (or may have been) shared with untrusted code U we link against. In particular, for values that are memory locations, high means that the location cannot be accessed by untrusted code (because it has never been shared with U), whereas low means that the location may be accessed and written to by U. Hence, during verification, we may impose custom Iris invariants on the contents of high locations, but not on the contents of low locations.

Corresponding to the distinction between high and low memory locations, Reckon provides two kinds of assertions: standard points-to assertions ($\ell \mapsto v$) for high locations, and a low predicate ($\mathsf{low}(\ell)$) for low locations. Note that the $\mathsf{low}(\ell)$ predicate does not specify the contents of ℓ—this is because the contents of a low location may not be preserved around a call to an untrusted function. In order to support reasoning about the contents of a low location within trusted code, Reckon thus provides rules to allow a low location to be temporarily borrowed as high so long as untrusted code is not invoked during the borrow.

The most important rule of Reckon is the following one, which enables safe invocation of an *untrusted* function f:[3]

$$\textsc{spec-call-un}\quad \left\{\mathop{\ast}_{v\in\overline{v}} \mathsf{low}(v)\right\}\ f(\overline{v})\ \{v.\ \mathsf{low}(v)\}$$

The rule states that we may safely invoke f so long as we only pass it low values, and in return the value we get back from f is also low. While this rule clearly places a restriction on interaction with untrusted code, it is still useful. For example, as we demonstrate in §2, we can easily use Reckon to verify robust safety of the `password_check` example (*i.e.*, that the assertion on line 6 always succeeds regardless of how `adv_io` is implemented). In §2, we also present further details of the Reckon logic, along with several other representative examples of its expressive power.

RobustDimSum: Proving soundness of Reckon via semantic back-translation. As explained above, Reckon is *not* sound under the C-like semantics of Rec—we already saw a counterexample to the soundness of spec-call-un with the adversarial implementation of `adv_io` in §1.1. Nevertheless, we will prove that Reckon *is* sound for Rec modules that are compiled by Rec2Cap and linked against untrusted Cap code. To establish this formally, we need to do two things:

- We need to prove that Rec2Cap correctly compiles Rec modules to Cap, so that specifications proven in Reckon continue to hold when the compiled code runs on the Cap machine.
- We need to prove that the proof rule for invoking untrusted functions, spec-call-un, is in fact validated by all possible Cap implementations of those functions.

The first of these goals is essentially a compiler correctness result: non-trivial, but also not requiring fundamentally new technology. The second, however, poses a major challenge. To understand why, let us consider a well-known proof strategy employed by prior work on robust safety preservation. To validate spec-call-un against all possible Cap implementations of the function f, we might hope to define a *syntactic back-translation* from Cap implementations to Rec implementations of f. Defining such a syntactic back-translation (in a semantics-preserving way) is notoriously painful, but in prior work, it was at least a sufficient technique for establishing robust safety preservation because the source languages under consideration in such work were safe. In contrast, in our setting, we would need to take an additional step of formally establishing

[3]This rule is a simplification. See §2.2 for the full rule.

that all Rec implementations in the image of the syntactic back-translation actually validate the SPEC-CALL-UN rule. Unfortunately, it is not at all clear how to do so.

To overcome this problem, we therefore move beyond syntax and develop a *semantic* characterization of how untrusted Cap code may behave. Towards that end, we build on DimSum [54], a recently developed Rocq-based framework for multi-language semantics and compiler verification. In DimSum, all modules of a program are modeled semantically as labeled state-transition systems, where the labels represent events, and modules interact with one another (à la process calculi) via matching sends and receives on these events. By building on DimSum, we are able to define a *universal contract*, *i.e.*, a semantic overapproximation of the possible behaviors of untrusted Cap code. In fact, we define *two* universal contracts: UNIV, a universal contract for Cap modules (interacting with other modules via Cap-level events), and SIM, a universal contract that *simulates* the behavior of Cap modules but interacts with other modules via Rec-level events. We then prove a novel *semantic back-translation* theorem[4] which relates the two universal contracts, ensuring (intuitively) that SIM is a valid Rec-level simulation of UNIV. With this semantic back-translation in hand, the soundness proof for Reckon reduces to a proof of compiler correctness for Rec2Cap, which we formalize in Rocq following the style of prior work on DimSum.

We call the resulting semantic framework RobustDimSum. In §3, we describe the high-level structure of RobustDimSum's proof strategy. Then, after introducing Cap in §4 and our compiler Rec2Cap and its correctness statement in §5, we get to the heart of RobustDimSum in §6 when we introduce SIM, UNIV, and the semantic back-translation between them. Finally, we put everything together to provide the soundness statement for Reckon in §7 and conclude with a discussion of related work in §8. The accompanying Rocq development can be found in the supplementary material [43].

Non-goals and limitations. The main contribution of this paper is the RobustDimSum approach to bringing robust safety to an unsafe source language via semantic back-translation. In contrast, our separation logic Reckon is intended merely as a proof of concept, with just enough features to verify interesting examples (§2.3) that showcase what RobustDimSum can support. As a logic, Reckon is deliberately derivative of prior work, and in fact there are technical reasons why it does not support all the bells and whistles of state-of-the-art Iris-based logics. In particular, Reckon provides only first-order ghost state and invariants, not higher-order ghost state [31] and impredicative invariants [63], because DimSum is incompatible with the countable step-indexing used by Iris. (We believe one could overcome this limitation by using transfinite step-indexing [60], but we leave this to future work.) In addition, Rec and Reckon consider only sequential programs since DimSum presently lacks support for concurrency. Finally, the capability language Cap assumes a somewhat idealized calling convention. §8 discusses how this calling convention could be implemented following recent literature.

2 Overview of Rec and Reckon

This section formally introduces the Rec language, as well as Reckon, our separation logic for it.

2.1 Rec: A Simple, Unsafe, C-Like Language

As our source language, we use the Rec language from the original DimSum paper [54]. The syntax of Rec is shown in Fig. 2. Values in Rec can be integers z, Booleans b, or memory locations ℓ. Rec has a block-based memory model (following CompCert [39, 38]) whereby locations are a pair of a block-identifier and an offset into the block. A Rec memory m is a finite map from locations to

[4]Although we do not literally back-translate Cap code (*i.e.*, syntax) to Rec code, we do back-translate the observable behaviors (*i.e.*, semantics) of Cap modules (modeled by UNIV) to corresponding observable behaviors of the Rec-level SIM.

$$\begin{array}{rl}
\mathsf{Val} \ni \mathsf{v} ::= z : \mathbb{Z} \mid b : \mathbb{B} \mid \ell : \mathsf{Loc} & \mathsf{BinOp} \ni \oplus ::= + \mid < \mid == \mid \leq \\
\mathsf{Memory} \ni \mathsf{m} \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val} & \mathsf{Loc} \ni \ell ::= \{\mathsf{id} : \mathsf{Id}, \mathsf{offset} : \mathbb{Z}\} \\
\end{array}$$

$$\mathsf{Expr} \ni \mathsf{e} ::= \mathsf{v} \mid x \mid \mathsf{e}_1 \oplus \mathsf{e}_2 \mid \mathsf{let}\ x := \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 \mid \mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ \mathsf{e}_2\ \mathsf{else}\ \mathsf{e}_3 \mid \mathsf{e}_1(\overline{\mathsf{e}_2}) \mid \mathsf{!e} \mid \mathsf{e}_1 \leftarrow \mathsf{e}_2$$

$$\mathsf{Library} \ni \mathsf{R} ::= (\mathsf{fn}\ \mathsf{f}(\overline{x}) \triangleq \overline{\mathsf{static}\ y[n]}; \overline{\mathsf{local}\ y[n]}; \mathsf{e}), \mathsf{R} \mid \emptyset$$

Fig. 2. Grammar of Rec (based on Sammler et al. [54]).

$$\boxed{\{P\}\ \mathsf{e}\ \{\mathsf{v}.\ Q\}_{ps} \triangleq P \twoheadrightarrow \mathsf{wp}^{ps}\ \mathsf{e}\ \{\mathsf{v}.\ Q\}}$$

WP-LOAD
$$\{\ell \mapsto \mathsf{v}_1\}\ !\ell\ \{\mathsf{v}.\ \mathsf{v} = \mathsf{v}_1 * \ell \mapsto \mathsf{v}_1\}_{ps}$$

WP-STORE
$$\{\ell \mapsto \mathsf{v}_2\}\ \ell \leftarrow \mathsf{v}_1\ \{\mathsf{v}.\ \mathsf{v} = \mathsf{v}_1 * \ell \mapsto \mathsf{v}_1\}_{ps}$$

WP-SHARE
$$\frac{\ell.\mathsf{id} \notin ps \qquad \{\ell \rightarrowtail D * \mathsf{low}(\ell)\}\ \mathsf{e}\ \{Q\}_{ps}}{\left\{\ell \rightarrowtail D * \mathop{\huge\ast}_{i \in D} \exists \mathsf{v}, \ell_i \mapsto \mathsf{v} * \mathsf{low}(\mathsf{v})\right\}\ \mathsf{e}\ \left\{Q\right\}_{ps}}$$

WP-BORROW
$$\frac{\begin{array}{c}\mathsf{b} = \ell.\mathsf{id} \\ \mathsf{b} \notin ps\end{array} \qquad \left\{\ell \rightarrowtail D * \mathop{\huge\ast}_{i \in D} \exists \mathsf{v}, \ell_i \mapsto \mathsf{v} * \mathsf{low}(\mathsf{v})\right\}\ \mathsf{e}\ \left\{Q\right\}_{\mathsf{b}::ps}}{\{\ell \rightarrowtail D *\ \mathsf{low}(\ell)\}\ \mathsf{e}\ \{Q\}_{ps}}$$

WP-RETURN
$$\frac{\mathsf{b} = \ell.\mathsf{id} \qquad \{\ell \rightarrowtail D\}\ \mathsf{e}\ \{Q\}_{ps}}{\left\{\ell \rightarrowtail D * \mathsf{low}(\ell) * \mathop{\huge\ast}_{i \in D} \exists \mathsf{v}, \ell_i \mapsto \mathsf{v} * \mathsf{low}(\mathsf{v})\right\}\ \mathsf{e}\ \left\{Q\right\}_{\mathsf{b}::ps}}$$

WP-CALL-UN
$$\frac{\mathsf{UnFn}(\mathsf{f}, n_\mathsf{f})}{\left\{|\overline{\mathsf{v}}| = n_\mathsf{f} * (\mathop{\huge\ast}_{\mathsf{v} \in \overline{\mathsf{v}}} \mathsf{low}(\mathsf{v}))\right\}\ \mathsf{f}(\overline{\mathsf{v}})\ \left\{\mathsf{v}.\ \mathsf{low}(\mathsf{v})\right\}^{\mathsf{Cond}}}$$

Fig. 3. Excerpt of the Reckon Separation Logic. ℓ_i is shorthand for $\{\ell.\mathsf{id}, i\}$

values. A Rec expression e can either be a value, variable, binary operation, let binding, if-statement, (recursive) function call, load, or store. While there are no primitive loops, Rec supports (mutual) recursion. Rec functions are collected into libraries R. Each function has a name f, a list of arguments $\overline{x}$, a list of static and local variables and a body (given by an expression e). Local variables are freshly allocated on each invocation of a function, whereas static variables are allocated at the beginning of the execution and retain their values between invocations.[5]

As highlighted in the introduction, Rec is an unsafe language. In particular, like C's semantics, Rec's semantics uses undefined behavior for various purposes, notably in response to memory-safety violations. Furthermore, Rec provides no built-in sandboxing mechanism to constrain this undefined behavior, nor any other way to safely interact with untrusted code. This makes Rec a good vehicle for showcasing our approach to proving robust safety for unsafe languages.

2.2 Reckon: A Separation Logic for Proving Robust Safety of Rec Programs

In this section, we present Reckon, a proof-of-concept separation logic for reasoning about robust safety of Rec programs.

At heart, Reckon is a simplified variant of Iris without step-indexing (for reasons discussed at the end of §1). While the absence of step-indexing means that we lose Iris's higher-order ghost state and impredicative invariants, we retain a simpler first-order variant of Iris invariants $\boxed{P}^{\gamma}$ with the restriction that P itself may not contain invariant assertions. This is still sufficiently useful to verify a number of interesting examples, as we will see in §2.3.

[5]Static variables are a new addition compared to the original Rec of Sammler et al. [54]. We use them in the counter example in §2.3.

As explained in §1.3, a key concept in Reckon (following OCPL [64]) is the distinction between *high* and *low* locations, with the former being private to trusted code and the latter being shareable with untrusted code. (This is generalized trivially to a high/low distinction on values by asserting that integer and Boolean values are vacuously low.) Reckon correspondingly includes different forms of assertions (we refer to them as recProp-assertions) to describe ownership of high vs. low locations.

$\ell \mapsto \mathsf{v}$	The location ℓ points to the value v.
$\mathsf{low}(\ell)$	The block with identifier $\ell.\mathsf{id}$ is shared with the untrusted code.
$\ell \rightarrowtail D$	The block with identifier $\ell.\mathsf{id}$ has D as the domain of block offsets.

The high assertion $\ell \mapsto \mathsf{v}$ expresses exclusive ownership of a high location ℓ, whereas the low assertion $\mathsf{low}(\ell)$ expresses the persistent knowledge that ℓ is low, meaning that the memory block with identifier $\ell.\mathsf{id}$ (*i.e.*, the memory block to which the location ℓ belongs) has been or can be shared with untrusted code. Note that in order for ℓ to be low, all values stored in ℓ's memory block must also be low since they will be accessible to untrusted code. In order to enforce this property (and more generally to track the scope of ℓ's memory block), we use the domain assertion $\ell \rightarrowtail D$, which expresses that the memory block to which ℓ belongs has domain D—*i.e.*, D is the set of valid offsets in that block.

Following Iris, the program logic of Reckon is based on a weakest precondition assertion $\mathsf{wp}^{ps}\ \mathsf{e}\ \{Q\}$. Roughly, $\mathsf{wp}^{ps}\ \mathsf{e}\ \{Q\}$ states that the expression e is robustly safe, and if it terminates in a value v, then $Q(\mathsf{v})$ holds.[6] The parameter *ps* tracks the shared blocks that are currently "borrowed" (see the discussion of borrowing below). We will use $\mathsf{wp}\ \mathsf{e}\ \{Q\}$ as a shorthand for $\mathsf{wp}^{[]}\ \mathsf{e}\ \{Q\}$ (*i.e.*, with empty *ps*), which we will call a *closed* weakest precondition.

Fig. 3 presents an excerpt of the program logic rules.

To share a block with untrusted code, Reckon provides the rule WP-SHARE. To apply the rule, the user must give up exclusive ownership of each location $\ast_{i \in D} \exists \mathsf{v}, \ell_i \mapsto \mathsf{v}$ in the memory block, whose domain is determined by $\ell \rightarrowtail D$. Furthermore, a location can only be shared if its content is safe to share, as expressed by requiring $\mathsf{low}(\mathsf{v})$ for each value in the block.

Given exclusive ownership of a location, the rules for load (WP-LOAD) and store (WP-STORE) are standard from separation logic. These are, however, not always directly applicable, for as we have seen, once a block is shared with untrusted code, exclusive ownership of its locations is given up. Thus, to access a location after it has been shared, the program logic provides a *borrowing* mechanism inspired by Simuliris [17], which allows the user to temporarily treat a low location as a high location. WP-BORROW grants temporarily exclusive access to the shared block b containing a low location ℓ, and adds the block to the list of borrowed blocks ($\mathsf{b} :: ps$). The list *ps* prevents borrowing the same block twice via the sidecondition $\mathsf{b} \notin ps$. In addition, one learns that all values v contained within the block b satisfy $\mathsf{low}(\mathsf{v})$. Once borrowed, a location can be returned using WP-RETURN, by which ownership of b is given back and b is removed from *ps*.

Finally, we come to the *pièce de résistance*: the rule WP-CALL-UN for *calls to untrusted functions*. This rule states that we can call the function f with n_{f} arguments, so long as all the arguments are low, in which case the return value is guaranteed to be low as well. Note that WP-CALL-UN uses a *well-bracketed Hoare triple* $\{P\}\ \mathsf{e}\ \{Q\}^{\mathsf{Cond}}_{ps}$ defined as follows:

$$\{P\}\ \mathsf{e}\ \{Q\}^{\mathsf{Cond}}_{ps} \triangleq \forall \Phi.\ \{P \ast \mathsf{Cond}(\Phi) \ast \Phi\}\ \mathsf{e}\ \{\mathsf{v}.\ Q(\mathsf{v}) \ast \Phi\}_{ps}$$

Intuitively, well-bracketed Hoare triples encode the property that the called function maintains well-bracketed control-flow, *i.e.*, calls and returns follow a stack-like discipline, and *e.g.*, returns cannot be reordered by untrusted code. They do so by taking a condition Φ from the set Cond as a

[6] More details on the definition of $\mathsf{wp}\ \mathsf{e}\ \{Q\}$ can be found in §7.1.

precondition and returning the same Φ. In §2.3, we will see how we can instantiate Cond to verify the classic "very awkward" example [13], which relies on well-bracketed control flow.

Note that WP-CALL-UN requires that there be no outstanding borrowed blocks (*i.e.*, the weakest-pre is closed). This is important since the untrusted code may overwrite the contents of any shared locations. The rule additionally depends on the predicate $\mathsf{UnFn}(\mathsf{f}, n_{\mathsf{f}})$ stating that the untrusted code provides the function f expecting n_{f} arguments. For now, we will keep this predicate abstract, and assume it as part of example specifications. §7.1 describes how we obtain it when applying the soundness theorem of Reckon.

2.3 Examples

To show the expressiveness of Reckon, we use it to verify some interesting examples. The proofs for all examples are mechanized in Rocq and included in our supplementary material [43].

The password-checking example. Let us prove a Reckon specification for our motivating example: the password-checking example from Fig. 1 (§1.1).

Lemma 2.1. *Choose* $\mathsf{Cond}(\Phi) \triangleq \Phi \equiv \top$. *Let* `read_hash_from_db` *and* `hash` *be two known functions satisfying the following specifications:*

$$\forall \ell.\ \{\exists \mathrm{v}, \ell \mapsto \mathrm{v}\}\ \mathtt{read\_hash\_from\_db}(\ell)\ \{\mathrm{v}.\ \exists z, \ell \mapsto z\}$$

$$\forall \mathrm{v}.\ \{\mathsf{low}(\mathrm{v})\}\ \mathtt{hash}(\mathrm{v})\ \{\mathrm{v}'.\ \exists z, \mathrm{v}' = z\}$$

We then prove the following:

$$\{\mathsf{UnFn}(\mathsf{adv\_io}, 1)\}\ \mathtt{password\_check}\ \{\mathrm{v}.\ \exists b, \mathrm{v} = b\}$$

Since the example does not depend on well-bracketed control flow, we use a trivial invariant condition for Cond. The precondition of `password_check` assumes $\mathsf{UnFn}(\mathsf{adv\_io}, 1)$ to reason about the call to `adv_io` (since `adv_io` takes one argument). The specification for `password_check` is otherwise quite simple: `password_check` returns some Boolean b. (We do not know whether it will be true or false since the password is given by `adv_io`.) However, recall that the body of `password_check` uses an assert statement to check that `adv_io` does not override pwd. We employ a semantics for the assert whereby, if it fails, it emits a function call to the `assert_failed` subroutine, which is unimplemented. The soundness theorem for Reckon ensures that such unimplemented function calls never occur, so by proving this spec we guarantee that the assert always succeeds.

The proof is largely straightforward. Upon initialization, the proof begins with ownership of two singleton blocks:

$$\begin{gathered}\mathsf{usr} \mapsto 0 * \mathsf{usr} \rightarrowtail \{0\}\\ \mathsf{pwd} \mapsto 0 * \mathsf{pwd} \rightarrowtail \{0\}\end{gathered}$$

Next, the specification for `read_hash_from_db` is applied, and pwd is updated to point to some integer z. We then arrive at the interesting part of the proof: the call to `adv_io`. Here we must apply WP-CALL-UN. Since the call takes `usr` as argument, we must establish $\mathsf{low}(\mathsf{usr})$. We do this by applying WP-SHARE, after which we have:

$$\begin{gathered}\mathsf{low}(\mathsf{usr}) * \mathsf{usr} \rightarrowtail \{0\}\\ \mathsf{pwd} \mapsto z * \mathsf{pwd} \rightarrowtail \{0\}\end{gathered}$$

We then frame ownership of pwd around the call to `adv_io`, so that afterwards we know that pwd still points to z and we can prove the assert succeeds. Finally, we use WP-BORROW to borrow the contents of `usr`, so that we can pass it to `hash` and conclude the proof.

Monotone counter: Invariants. To illustrate invariant-based reasoning, we verify a simple monotone counter corresponding to the following code.

```
void counter() { static int c = 0; c += 1; assert(0 <= c); } void main() { adv(); }
```

The counter function increments the static variable `c` and asserts that it is always positive.[7] The `main` function invokes the untrusted code `adv` that can arbitrarily call `counter`. To prove that `counter` is robustly safe, we use the *invariant* that `c` always remains positive $\boxed{\exists z, \mathtt{c} \mapsto z * z \geq 0}^{\gamma}$. Our program logic allows us to establish and maintain this invariant since `c` is never shared with the untrusted code.

Very awkward example: Well-bracketed control flow. Well-bracketed control flow is a safety property which guarantees that function calls return exactly to their call-site. By ensuring well-bracketed control flow, we can prevent control-flow hijacking attacks that divert program executions by replacing a return with some unwanted execution [2]. Thankfully, recent work has shown how capability machines can be used to enforce well-bracketed control flow, thus eliminating this family of attacks [55]. By supporting well-bracketed reasoning in Reckon, we show furthermore how these safety measures can be soundly lifted to the Rec level.

To showcase well-bracketed reasoning, we verify the well-known *"very awkward" example* [13]:

```
void awk() { static int x; x = 0; adv(); x = 1; adv(); assert(x == 1); }
```

In this example, `awk` calls `adv`, and `adv` could call `awk` back, so `x` could be flip-flopped back and forth between 0 and 1. However, well-bracketed control flow ensures that, since `x` is private to `awk`, each (recursive) call to `awk` within `adv` can only have the *end-to-end* effect of setting `x` to 1. As a result, after `x` is set to 1 in `awk`, the subsequent call to `adv()` can only have the end-to-end effect of leaving `x` as it is (either it never calls `awk`, in which case `x` stays at 1, or it calls `awk`, in which case it flip-flops `x` but returns with `x` still at 1). Hence, the assert must succeed.

Our approach to proving this follows Timany et al. [66] except that we use Cond to track the stack of ghost names. Concretely, we pick $\mathrm{Cond}(\Phi) \triangleq \exists \overline{\gamma}.\ \Phi \equiv \boxed{\bullet \overline{\gamma}}$. Here, $\overline{\gamma}$ represents a stack of ghost names, each corresponding to a recursive invocation of `awk`. The fact that WP-CALL-UN provides the same Φ in the postcondition as given in the argument allows us to reason that the untrusted code maintains well-bracketed control flow (since the stack is the same after the call as before). This allows us to prove robust safety of the very awkward example.

3 Overview of RobustDimSum

Background: DimSum. Our approach to proving soundness of Reckon builds on DimSum [54], a semantic framework for modular, refinement-based reasoning about multi-language programs. DimSum models program modules denotationally as *semantic modules*. A semantic module is a labeled transition system, whose states are the possible internal states of the program module, and whose transitions can be labeled with *external interaction events*, e.g., calls to and from other modules. Each language (*i.e.*, Rec and Cap in our case), comes with a fixed set of events. As an example, for Rec, these events denote incoming ($\cdot ?$) and outgoing ($\cdot !$) calls and returns, carrying the arguments/return value and the full state of the memory:

$$\mathrm{Events} \ni \mathrm{e} ::= \mathrm{Call!}(\mathrm{f}, \overline{\mathrm{v}}, \mathrm{m}) \mid \mathrm{Call?}(\mathrm{f}, \overline{\mathrm{v}}, \mathrm{m}) \mid \mathrm{Return!}(\overline{\mathrm{v}}, \mathrm{m}) \mid \mathrm{Return?}(\overline{\mathrm{v}}, \mathrm{m})$$

The DimSum module corresponding to a syntactic module P is written $[\![P]\!]$. We annotate the semantic brackets $[\![\cdot]\!]$ with subscripts to distinguish the modules of different languages, e.g., $[\![\mathrm{P}]\!]_{\mathrm{r}}$ denotes a Rec module whereas $[\![\mathbf{P}]\!]_{\mathbf{c}}$ denotes a Cap module.[8]

[7]Note that integers in Rec are unbounded, unlike C.
[8]We consistently use different colors to denote Rec and Cap entities.

Two DimSum modules of the same language can be *semantically linked* through the operator $\oplus$, as in $[\![\mathbf{P_1}]\!]_\mathbf{c} \oplus_\mathbf{c} [\![\mathbf{P_2}]\!]_\mathbf{c}$ (again, the subscript $\mathbf{c}$ indicates the language). Semantic linking produces a new module that eliminates interactions that occur between the two modules, but leaves interactions with other modules intact. Each language also comes with a *syntactic linking operator* $\cup$ that combines two syntactic modules. These syntactic and semantic linking operators coincide on syntactic modules as in the following equations.

$$[\![\mathbf{P_1} \cup_\mathbf{c} \mathbf{P_2}]\!]_\mathbf{c} \equiv [\![\mathbf{P_1}]\!]_\mathbf{c} \oplus_\mathbf{c} [\![\mathbf{P_2}]\!]_\mathbf{c} \qquad [\![\mathsf{P}_1 \cup_\mathsf{r} \mathsf{P}_2]\!]_\mathsf{r} \equiv [\![\mathsf{P}_1]\!]_\mathsf{r} \oplus_\mathsf{r} [\![\mathsf{P}_2]\!]_\mathsf{r} \tag{1}$$

To reason across languages, DimSum introduces the concept of *wrappers*. A wrapper converts a module over one set of events to a module over a different set of events. It does this by translating the events emitted by the converted module. This translation may be stateful, thus allowing for nontrivial transformations of sequences of events. In our setup, we define a specific wrapper from Rec to **Cap**, written $\lceil \cdot \rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$. This wrapper converts Rec events (shown above) to **Cap** events.

Proofs in DimSum establish (termination-insensitive) *refinement* between two modules of the same language, written $[\![\mathbf{P_1}]\!]_\mathbf{c} \preceq [\![\mathbf{P_2}]\!]_\mathbf{c}$ ("$[\![\mathbf{P_1}]\!]_\mathbf{c}$ refines $[\![\mathbf{P_2}]\!]_\mathbf{c}$"). This relationship should be understood in terms of labeled transition systems as "every labeled execution trace of $[\![\mathbf{P_1}]\!]_\mathbf{c}$ is also a trace of $[\![\mathbf{P_2}]\!]_\mathbf{c}$". To relate the modules of two different languages, we combine refinement and wrappers. For example, we may say $[\![\mathbf{P_1}]\!]_\mathbf{c} \preceq \lceil [\![\mathsf{P}_2]\!]_\mathsf{r} \rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$.

We use the notation $\downarrow\mathsf{P}$ to denote the target-language program obtained by compiling the source program P. Specifically, in this paper we use $\downarrow\cdot$ for the compiler Rec2Cap from Rec to **Cap**. In DimSum, compositional compiler correctness is stated succinctly as the statement

$$[\![\downarrow\mathsf{P}]\!]_\mathbf{c} \preceq \lceil [\![\mathsf{P}]\!]_\mathsf{r} \rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}} \tag{2}$$

which means that any semantic behavior of the compiled module $\downarrow\mathsf{P}$ can be simulated by the source module P translated with the source-to-target wrapper.

Overview. Next, we provide an overview of RobustDimSum, our novel proof method for establishing robust safety for Rec programs compiled to **Cap** using Rec2Cap. Given a Rec module P, our goal is to establish that $\downarrow\mathsf{P} \cup_\mathbf{c} \mathbf{un}$ does not have failed asserts. (This implies also that the program does not exhibit undefined behavior, since undefined behavior could result in failed asserts.) We model a failed assert as a call to an undefined function `assert_failed` that would appear on the trace as an undefined call event, and we prove that the trace does not contain any such undefined call events.

To formalize this property in DimSum, we define a simple **Cap** semantic specification module, $\mathbf{safe_c}$, which does not make any undefined calls (see §7). Then, robust safety can be restated as the following refinement:

$$\forall \mathbf{un}.\ [\![\downarrow\mathsf{P} \cup_\mathbf{c} \mathbf{un}]\!]_\mathbf{c} \preceq \mathbf{safe_c} \tag{3}$$

Our goal is to obtain Refinement (3) from Reckon. However, we run into a key challenge: Reckon is defined purely on Rec, but the soundness statement needs to reason about the untrusted **Cap** module **un**. How can we bridge this gap?

As explained in §1, we would like to somehow "back-translate" **un** to a member of a class of Rec modules that are "well-behaved" in the sense that they validate the rules of Reckon. But it is not at all clear how to do so. This is where moving to a semantic domain like DimSum's modules pays off generously: Rather than characterize this class of Rec modules that behave like untrusted **Cap** modules syntactically, we characterize the class semantically as a *single* Rec module in DimSum that can simulate the behavior of all **Cap** modules. This Rec module, called SIM, is relatively easy to define in DimSum, which is unconstrained by syntax. Specifically, we define SIM using logical assertions that can be encoded into an operational semantics thanks to DimSum's support for

$$\boxed{\text{Soundness of Reckon w.r.t. SIM}}$$

$$P \text{ verified using Reckon} \Longrightarrow [\![P]\!]_r \oplus_r \mathsf{SIM} \oplus_r \mathsf{prim}_{\mathsf{spec}} \preceq \mathsf{safe}_r \quad \ldots \tag{4}$$

$$\boxed{\text{Semantic refinements}}$$

$$\begin{array}{lll}
\forall \mathbf{un}. & [\![\downarrow \mathsf{P} \cup_c \mathbf{un} \cup_c \mathbf{prim}]\!]_c & \\
& \preceq [\![\downarrow \mathsf{P}]\!]_c \oplus_c [\![\mathbf{un}]\!]_c \oplus_c [\![\mathbf{prim}]\!]_c & \text{Semantic linking, Eq. (1)} \quad (5) \\
& \preceq \lceil [\![\mathsf{P}]\!]_r \rceil_{r \rightleftharpoons c} \oplus_c [\![\mathbf{un}]\!]_c \oplus_c [\![\mathbf{prim}]\!]_c & \text{Compiler Correctness, Theorem 5.1 (§5)} \quad (6) \\
& \preceq \lceil [\![\mathsf{P}]\!]_r \rceil_{r \rightleftharpoons c} \oplus_c \mathbf{UNIV} \oplus_c [\![\mathbf{prim}]\!]_c & \text{Universal Contract, Theorem 6.1 (§6.2)} \quad (7) \\
& \preceq \lceil [\![\mathsf{P}]\!]_r \rceil_{r \rightleftharpoons c} \oplus_c \lceil \mathsf{SIM} \rceil_{r \rightleftharpoons c} \oplus_c [\![\mathbf{prim}]\!]_c & \text{Simulation, Theorem 6.2 (§6.3)} \quad (8) \\
& \preceq \lceil [\![\mathsf{P}]\!]_r \rceil_{r \rightleftharpoons c} \oplus_c \lceil \mathsf{SIM} \rceil_{r \rightleftharpoons c} \oplus_c \lceil \mathsf{prim}_{\mathsf{spec}} \rceil_{r \rightleftharpoons c} & \text{Primitive} \quad (9) \\
& \preceq \lceil [\![\mathsf{P}]\!]_r \oplus_r \mathsf{SIM} \oplus_r \mathsf{prim}_{\mathsf{spec}} \rceil_{r \rightleftharpoons c} & \text{Linking of Wrappers, Lemma 5.2 (§5)} \quad (10) \\
& \preceq \lceil \mathsf{safe}_r \rceil_{r \rightleftharpoons c} & \text{Soundness, Theorem 7.1 (§7.1)} \quad (11) \\
& \preceq \mathbf{safe}_c & \text{Theorem 7.2 (§7.2)} \quad (12)
\end{array}$$

Fig. 4. A summary of RobustDimSum

angelic nondeterminism. We show that SIM can simulate the behavior of any Cap context up to the Rec-to-Cap wrapper. In DimSum's notation,

$$\forall \mathbf{un}.\ [\![\mathbf{un}]\!]_c \preceq \lceil \mathsf{SIM} \rceil_{r \rightleftharpoons c} \tag{13}$$

Having defined SIM, we prove Reckon sound w.r.t. it, as expressed by Refinement (4) in Fig. 4. Here, safe_r is a straightforward Rec module in DimSum that does not make undefined function calls. Consequently, Refinement (4) says that a verified module P, after being linked semantically with SIM, will never make an undefined function call.

Proof outline. We now explain how all these pieces come together in RobustDimSum to establish Refinement (3). The outline of the proof is given in Fig. 4. The gray parts about **prim** can be ignored now; we will return to them later.

The first two proof steps are straightforward: Step (5) decomposes the syntactic Cap module into semantic modules using Eq. (1). Step (6) applies the compiler correctness statement (Refinement (2)).

Next, we lift **un** to SIM by proving Refinement (13). We actually do this in two steps, labeled (7) and (8) in Fig. 4. First, we introduce a DimSum Cap module **UNIV** that over-approximates all Cap modules, *i.e.*, $[\![\mathbf{un}]\!]_c \preceq \mathbf{UNIV}$ for all $[\![\mathbf{un}]\!]_c$ (Step (7)). Then, we perform *semantic back-translation* to show that **UNIV** refines SIM, *i.e.*, $\mathbf{UNIV} \preceq \lceil \mathsf{SIM} \rceil_{r \rightleftharpoons c}$ (Step (8)). This two-step decomposition of Refinement (13) breaks the proof into two manageable steps: Relating $[\![\mathbf{un}]\!]_c$ and **UNIV** requires a large but straightforward case distinction over all instructions of Cap to show that **UNIV** can simulate them, while the semantic back-translation, which lies at the heart of RobustDimSum, is made significantly easier by the fact that it does not need to refer to any syntax (**UNIV**, SIM and $\lceil \cdot \rceil_{r \rightleftharpoons c}$ are all defined semantically).

Finally, we apply DimSum structural rules to commute the wrapper with the linking operator (Step (10)) and the soundness of Reckon w.r.t. SIM (Step (11)). The last step is a straightforward refinement proof in DimSum that relates safe_r and $\mathbf{safe}_c$ (Step (12)).

Exposing Cap primitives to Rec. Let us now turn to the gray parts of Fig. 4. They pertain to a feature of RobustDimSum not discussed so far: the ability to expose primitives to Rec programs that have been defined directly in Cap and verified manually. To illustrate this feature, consider implementing the hash function used by password_check in Fig. 1. The challenge here is that we need to compute the hash of a low value received from the untrusted code: we cannot implement hash naively as usr % 1337, because, as it is common for untyped languages, in Rec, % is only defined on integers. It would lead to undefined behavior if adv_io returns a pointer! Thus, we need an operation that distinguishes integers from pointers. However, Rec does not provide such an operation since not all targets to which Rec may be compiled would support the operation. This is where DimSum saves the day: we define **to_int** as a syntactic Cap program (code shown below) and back-translate it to a Rec *semantic* module, which can then be linked *semantically* against Rec programs.

```
isptr R1, R0 # store 1 to R1 if R0 stores a capability
beq 2, R1, 0  # increase pc by 2 if R1 = 0 (skip next instruction if not a capability)
mov R0, 0     # store 0 as default value to R0
ret           # return current value in R0
```

This function uses the isptr instruction of Cap (§4) to check if the argument (stored in R0) is a capability and returns 0 in this case. Otherwise, it is the identity function. Using **to_int**, we can implement a safe hash function Rec:

```
int hash(int usr) { return to_int(usr) % 1337; }
```

To reason about a program that uses hash and **to_int**, we first need to back-translate the latter to Rec. This is where DimSum comes in: While we cannot back-translate **to_int** to a syntactic Rec program, we can represent it as a *semantic* module in a special DimSum language, Spec, which is a mathematical specification language inspired by interaction trees [72]. In this language, we can specify **to_int** by directly matching on the structure of the argument. Note that Rec functions can pass Booleans as arguments, that are compiled to 0 and 1, which needs to be reflected accordingly:[9]

$$\mathsf{to\_int}_{\mathsf{spec}}(\mathsf{v}) \triangleq \text{if } \mathsf{v} \text{ is } z \text{ then } z \text{ else if } \mathsf{v} \text{ is true then } 1 \text{ else } 0$$

Next, we prove $[\![\mathbf{to\_int}]\!]_{\mathbf{c}} \preceq \lceil \mathsf{to\_int}_{\mathsf{spec}} \rceil_{\mathsf{r} \rightleftharpoons \mathbf{c}}$. This step is integrated in Step (9) in Fig. 4, where $\mathbf{prim} = \mathbf{to\_int}$ and $\mathsf{prim}_{\mathsf{spec}} = \mathsf{to\_int}_{\mathsf{spec}}$. The module $\mathsf{prim}_{\mathsf{spec}}$ is also exposed to Reckon as seen in Refinement (4), which allows us to verify hash.

4 Cap: Target Capability Language

In this section we present our target language Cap, which will provide the necessary security primitives to implement a hardening compiler in §5. Cap is a simplified capability machine language in the style of Cerise [19]. To simplify matters, we assume an abstract calling convention based on an idealized stack that ensures well-bracketed control flow and temporal safety similar to the overlay semantics defined and verified in prior work [56, 21].

We can divide the security primitives provided by Cap into three distinct safety features. First is fine-grained memory protection, second is encapsulation of a closure (a function and its private state), and third is well-bracketed control flow.

Fine-grained memory protection. Capabilities are first and foremost a primitive for memory protection. Unlike pointers on typical hardware, a capability is unforgeable and may only access specific memory regions. We represent capabilities as tuples defining their authority and their current pointer value.

Fig. 5 shows the full syntax of capabilities. We define two kinds of capabilities: heap capabilities and stack capabilities. A heap capability has the form $(\mathbf{p}, b, e, a)$ where $\mathbf{p}$ is a permission, b is the lower bound of authority, e is the upper bound of authority, and a is the current address. The

[9]We omit the call and return events around this implementation for simplicity of presentation.

$$
\begin{array}{llcl}
\mathrm{HeapCap} & \ni \ \mathrm{c} & \triangleq & \{(\mathrm{p}, b, e, a) \mid b, e, a \in \mathbb{Z}\} \\
\mathrm{StackCap} & \ni \ \mathrm{sc} & \triangleq & \{\mathrm{Stk}(d, \mathrm{p}, b, e, a) \mid b, e, a \in \mathbb{Z}, d \in \mathbb{N}\} \\
\mathrm{Word} & \ni \ \mathrm{w} & \triangleq & \mathrm{HeapCap} \cup \mathrm{StackCap} \cup \mathbb{Z} \\
 & & & \\
\mathrm{Memory} & \ni \ \mathrm{m} & \triangleq & \mathbb{Z} \rightharpoonup \mathrm{Word} \\
\mathrm{StkFrame} & \ni \ \mathrm{f} & \triangleq & \mathbb{B} \times \mathrm{Memory} \times \mathrm{Word} \times \mathrm{HeapCap} \\
\mathrm{Stack} & \ni \ \mathrm{s} & \triangleq & \mathcal{L}\,(\mathrm{StkFrame}) \\
\mathrm{Registers} & \ni \ \mathrm{r} & \triangleq & \mathrm{RegisterName} \rightharpoonup \mathrm{Word} \\
\mathrm{ExecutionState} & \ni \ \mathrm{I} & ::= & \mathrm{Wait} \mid \mathrm{Run}(\mathrm{r}, \mathrm{m}, \mathrm{s})
\end{array}
$$

$$\begin{tikzcd}
 & \mathrm{RWX} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{RW} \arrow[dr, dash] & & \mathrm{RX} \arrow[dl, dash] \arrow[d, dash] \\
 & \mathrm{RO} \arrow[d, dash] & \mathrm{E} \arrow[dl, dash] \\
 & \mathrm{O} &
\end{tikzcd}$$

Permissions $\ni$ p
defines the $\preccurlyeq$ order

$$
\begin{array}{llcl}
\mathrm{Operand} & \ni \ \mathrm{o} & ::= & \mathrm{x} : \mathrm{RegisterName} \mid i : \mathbb{Z} \\
\mathrm{Instr} & \ni \ \mathrm{i} & ::= &
\end{array}
$$

$$
\begin{array}{l}
\texttt{mov}\ \mathrm{x},\ \mathrm{o} \mid \texttt{add}\ \mathrm{x1},\ \mathrm{x2},\ \mathrm{o} \mid \texttt{mul}\ \mathrm{x1},\ \mathrm{x2},\ \mathrm{o} \mid \texttt{seq}\ \mathrm{x1},\ \mathrm{x2},\ \mathrm{o} \mid \texttt{sle}\ \mathrm{x1},\ \mathrm{x2},\ \mathrm{o} \mid \texttt{slt}\ \mathrm{x1},\ \mathrm{x2},\ \mathrm{o} \mid \\
\texttt{beq}\ \mathrm{o},\ \mathrm{o}_1,\ \mathrm{o}_2 \mid \texttt{beq}\ \mathrm{x1},\ \mathrm{x2},\ \mathrm{o} \mid \texttt{restrict}\ \mathrm{x1},\ \mathrm{x2},\ \mathrm{o} \mid \texttt{subreg}\ \mathrm{x1},\ \mathrm{x2},\ \mathrm{o}_1,\ \mathrm{o}_2 \mid \texttt{isptr}\ \mathrm{x1},\ \mathrm{x2} \mid \\
\texttt{getp}\ \mathrm{x1},\ \mathrm{x2} \mid \texttt{getb}\ \mathrm{x1},\ \mathrm{x2} \mid \texttt{gete}\ \mathrm{x1},\ \mathrm{x2} \mid \texttt{geta}\ \mathrm{x1},\ \mathrm{x2} \mid \texttt{getl}\ \mathrm{x1},\ \mathrm{x2} \mid \texttt{getd}\ \mathrm{x1},\ \mathrm{x2} \mid \\
\texttt{ldr}\ \mathrm{x1},\ [\mathrm{x2} + i] \mid \texttt{str}\ [\mathrm{x} + i],\ \mathrm{o} \mid \texttt{call}\ \mathrm{x}\ [\mathrm{x1}; \ldots; \mathrm{xn}] \mid \texttt{ret} \mid \texttt{allocStack}\ z
\end{array}
$$

Fig. 5. Capability Language Syntax

$$\mathit{decode} : \mathbb{Z} \xrightarrow{\text{fin}} \mathrm{Instr} \qquad \mathrm{Library} \ni \mathrm{A} \subseteq \mathbb{Z}$$

$$[\![\mathrm{r}(\mathrm{pc})]\!]_{\mathrm{A}} \triangleq \begin{cases} \mathit{decode}(\mathrm{m}(a)) & \text{if } \mathrm{r}(\mathrm{pc}) = (p, b, e, a) \land b \le a < e \land p \in \{\mathrm{RX}, \mathrm{RWX}\} \land a \in \mathrm{A} \\ \text{undef.} & \text{otherwise} \end{cases}$$

ASM-CALL-EXTERNAL

$$\frac{\begin{array}{c}[\![\mathrm{r}(\mathrm{pc})]\!]_{\mathrm{A}} = \texttt{call}\ \mathrm{x}\ [\mathrm{x1}; \ldots; \mathrm{xn}] \quad \mathrm{r}(\mathrm{x}) = (\mathrm{E}, b, e, a) \quad \mathrm{r}(\mathrm{pc}) = (p_A, b_A, e_A, a_A) \quad \mathit{last}(\mathrm{s}).1 = \text{true} \quad a \notin \mathrm{A} \\ \mathrm{r}' = [\mathrm{pc} \mapsto (\mathrm{RX}, b, e, a), \mathrm{x1} \ldots 9 \mapsto r(\mathrm{x1} \ldots 9)] \quad \mathrm{s}' = \mathrm{s} +\!\!+ [(\text{false}, \emptyset, \mathrm{r}(\mathrm{sp}), (p_A, b_A, e_A, a_A + 1))]\end{array}}{(\mathrm{Run}(\mathrm{r}, \mathrm{m}, \mathrm{s}), \mathrm{A}) \xrightarrow{\mathrm{Jump!}(\mathrm{r}', \mathrm{m}, \mathrm{s}')} \{(\mathrm{Wait}, \mathrm{A})\}}$$

ASM-RET-EXTERNAL

$$\frac{[\![\mathrm{r}(\mathrm{pc})]\!]_{\mathrm{A}} = \texttt{ret} \quad \mathrm{r}' = [\mathrm{pc} \mapsto \mathit{cont}, \mathrm{x0} \mapsto \mathrm{r}(\mathrm{x0}), \mathrm{sp} \mapsto \mathit{sp}] \quad \mathrm{s}' = \mathrm{s} +\!\!+ [(b, m, \mathit{sp}, \mathit{cont})] \quad \mathit{cont}.a \notin \mathrm{A}}{(\mathrm{Run}(\mathrm{r}, \mathrm{m}, \mathrm{s}'), \mathrm{A}) \xrightarrow{\mathrm{Jump!}(\mathrm{r}', \mathrm{m}, \mathrm{s})} \{(\mathrm{Wait}, \mathrm{A})\}}$$

ASM-ALLOCSTACK

$$\frac{\begin{array}{c}[\![\mathrm{r}(\mathrm{pc})]\!]_{\mathrm{A}} = \texttt{allocStack}\ z \quad \mathrm{s} = \mathrm{s}' +\!\!+ [(\text{false}, \emptyset, \mathit{sp}, \mathit{cont})] \quad \mathrm{r}(\mathrm{pc}) = (p_A, b_A, e_A, a_A) \\ \mathrm{r}' = \mathrm{r}[\mathrm{pc} \mapsto (p_A, b_A, e_A, a_A + 1)][\mathrm{sp} \mapsto \mathrm{Stk}(|\mathrm{s}'|, \mathrm{RW}, 0, z, 0)] \quad \mathrm{s}'' = \mathrm{s}' +\!\!+ [(\text{true}, [0 \ldots z \mapsto 0], \mathit{sp}, \mathit{cont})]\end{array}}{(\mathrm{Run}(\mathrm{r}, \mathrm{m}, \mathrm{s}), \mathrm{A}) \xrightarrow{\tau} (\mathrm{Run}(\mathrm{r}', \mathrm{m}, \mathrm{s}''), \mathrm{A})}$$

Fig. 6. Excerpt of the Capability Language Operation Semantics

permission p ranges over a lattice described in Fig. 5, where the top of the lattice RWX describes the permission to read r, write w, and execute x a capability (the program counter capability must have the permission to execute), while the bottom of the lattice O describes zero authority. Meanwhile, a stack capability $\mathrm{Stk}(d, \mathrm{p}, b, e, a)$ additionally points to a specific stack frame index d.

The bounds and permission of a capability are dynamically checked whenever a load `ldr` or store `str` instruction is executed. If the check fails, the machine halts safely.

All capability instructions are listed in Fig. 5. Some (such as `str` and `ldr`) are standard assembly instructions, while others are specific to capability machines: `restrict` (resp. `subseg`) lowers the permission (resp. range of authority) of a capability, `isptr` determines whether a word is an integer or a capability, and `get` instructions read a field from the capability tuple.

Encapsulation of function pointers. To keep capabilities encapsulated from other compartments, capability architectures like CHERI [70] offer a range of special capability permissions and seals to enable fast and secure context switches. These special capabilities describe the authority to *jump* to a specific target, without being able to access that target's internal state. We follow Cerise [18] and model one such capability, namely the so-called *sentry capability*, also called *enter capability*. Enter capabilities are capabilities with a special permission **E** (short for enter), which may be jumped to, but may not be changed, nor used to read from or write to memory. In §5, we will see how a compiled program has both a program part and a data part, and how enter capabilities can be used to encapsulate them together into an opaque entry point. The **E** permission will both prevent callers from skipping over program instructions, and will prevent them from reading the data part of a compiled program. If it is jumped to, the permission is changed to **RX**, and the callee gets full access to their private state.

Well-bracketed control flow. Finally, Cap ensures well-bracketed control flow to prevent any control-flow hijack attack. Enforcing well-bracketed control flow on a capability machine is notoriously difficult, and not in the scope of this work. Instead, we assume an abstract stack and a secure calling convention, implemented over three instructions that manipulate the stack: call, return and stack frame allocation, described in Fig. 6.

Before presenting these instruction, let us first describe the anatomy of the stack. We represent the stack as a list of stack frames, denoted ($b : \mathbb{B}$, $\mathbf{m}$: **Memory**, sp : **Word**, $cont$: **HeapCap**), which are made up of a boolean b (indicating whether the frame has been allocated), a state $\mathbf{m}$, a back link sp (pointing to the previous frame), and a continuation $cont$ (the capability to jump to the return address).

Let us next describe the three stack manipulating instructions. Calls `call` $\mathbf{x}$ $[\mathbf{x1}; \ldots; \mathbf{xn}]$ jump to the capability in register $\mathbf{x}$ (by loading it into the PC and changing **E** to **RX**) with parameters from registers $\mathbf{x1}$ to $\mathbf{xn}$. This instruction pushes a new empty frame onto the stack, with the caller's stack pointer as the back link and the next instruction as the continuation. Calls that go outside the address space of the module emit a jump event $\mathbf{Jump!(r, m, s)}$ with the current machine state.

Returns `ret` pop the topmost stack frame, reinstate the back link, and jump to the program continuation. Crucially, `ret` is the only instruction that accesses the return address or the stack pointer fields of a stack frame. Since it is always the topmost frame that is popped, calls are inherently well-bracketed. We here present a simplified version of the rule, omitting additional constraints on the return value to prevent dangling stack pointers and enforce temporal safety.

Since the caller may not know the stack size needed by the callee, newly pushed frames start out as empty. Cap thus offers a third instruction `allocStack` z used by the callee to allocate the memory of a stack frame. This instruction can only be invoked once per frame, as enforced by the boolean flag b.

5 Compiler Correctness

This section describes our hardening compiler Rec2Cap from Rec to Cap. Fig. 7 outlines the passes of our compiler. *SSA* renames variables such that each variable name is used once, *Linearize* puts the program into A-normal form (LinearRec is a subset of Rec where all programs are in A-normal form), and *Codegen* takes a LinearRec and generates Cap machine code.

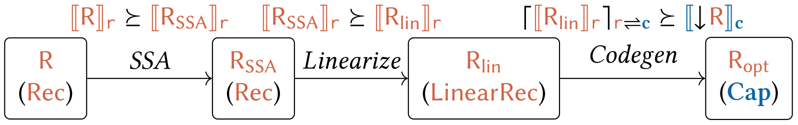
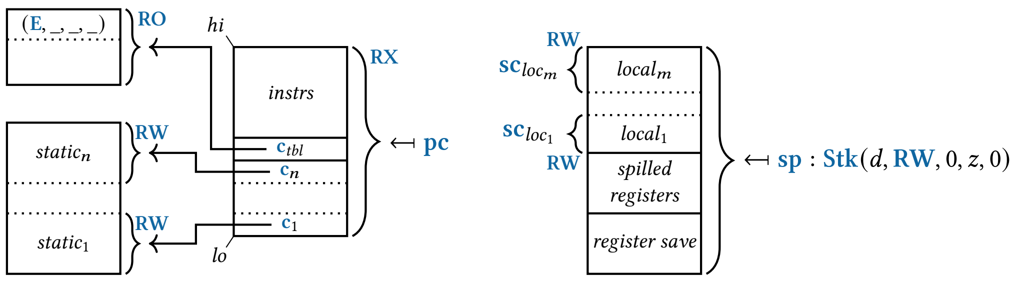


Fig. 7. Structure of the Rec to Cap Compiler



(a) Memory Layout of a Generated Cap Function (b) Stack Layout During Execution

Fig. 8. Static and Dynamic Memory Layout. Addresses increase upwards.

This structure follows the Rec to Asm compiler by Sammler et al. [54] with the main difference being that the code generation pass of Rec2Cap targets the capability language Cap instead of the more traditional assembly language Asm. Additionally, Rec2Cap needs to take the addition of static variables to Rec into account and it omits the *Mem2Reg* optimization pass.

For the remainder of this section, we will focus on the *Codegen* pass, which compiles LinearRec programs into Cap code. This pass must enforce robust safety against arbitrary Cap programs. It does so by following the "pointers as capabilities" (*PAC*) principle [70, 15]. In order to fully take advantage of this principle, the compiler must additionally follow the *principle of least privilege*—meaning each capability only grants the exact authority needed to access a single variable.

Let us now look at the memory and stack layout used by the compiler to see these principles in action. Fig. 8a describes the static memory layout of a generated Cap program. Simply put, a Cap program is a RX capability which points to capability machine instructions and some local state. At the top of this capability, we have the instructions of the program. Next, we have a RO capability pointing to a linking table, which contains enter capabilities for each entry point of the fully linked program. Finally, we have heap capabilities pointing to the static variables of the program, each with a RW permission. Following the principle of least privilege, each of the static variables has its own capability. This means that giving the untrusted code access to one of the static variables does *not* give it access to other static variables.

During execution, the compiled program uses the stack to manage control flow and local state. Fig. 8b describes the stack frame layout during execution. The top of the frame is used to store local variables. Following the principle of least privilege, the compiler derives a distinct stack capability for each local variable. Next, the compiler uses the frame for spilled registers. Finally, stack space is reserved to store local state during a potential call (referred to in Fig. 8b as the *register save*). This is important for security: To prevent leaking capabilities to the untrusted code via registers, all non-argument registers must be cleared upon call and their values saved on the stack.

Compiler correctness. A crucial step in the proof methodology outlined in § 3 is to prove *compiler correctness*. Roughly, compiler correctness states that a compiled program ↓P refines the source program P. The full statement is slightly more involved: the compiled program must be laid out in memory, and the initial memory layout is constructed according to the program's static memory and linking table, leading to the following theorem:

Theorem 5.1 (Compiler Correctness). *Let* $\mathbf{m}$ *be the initial memory layout as outlined in Fig. 8a, storing* $\downarrow$P *as the program instructions. Let* $\mathbf{A}$ *refer to the domain of the program counter range. Let* f *be the name of program* P*. Let tbl refer to a mapping from function names to entry points (*i.e.*, enter capabilities) in the linking table.*

Additionally, assume the following:

- *A valid entry point to* f *is stored in tbl, pointing to the first instruction of the program and ranging over* $[lo, hi)$ *as depicted in* $\mathbf{m}$.
- *Each other entry point in tbl is disjoint from* $\mathbf{A}$.
- *Each static has a non-empty range.*
- *Each function name of entry points in tbl is unique.*

We then have:

$$[\![\downarrow \mathsf{R}]\!]_{\mathbf{c}} \preceq {}^{\text{🔒}}\lceil [\![\mathsf{R}]\!]_{\mathsf{r}} \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}^{\mathbf{A},\{\mathsf{f}\},tbl,\mathbf{m},\mathsf{m}_0,\mathcal{A}_?,\mathcal{E}_!}$$

Wrapper. Let us now introduce the wrapper $\lceil\cdot\rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}$ used in this theorem. Intuitively, the wrapper takes a module emitting Rec events (*i.e.*, Call and Return) and transforms it into a module emitting **Cap** events (*i.e.*, **Jump**). §6 describes how this transformation works in more detail—for now one can think of the wrapper as a semantic representation of the compiler that describes the compilation as a (separation-logic) relation that describes how incoming **Jump** events can be translated to Call and Return events and vice versa.

For the wrapper to work, it requires a bunch of arguments ${}^{\text{🔒}/\text{🔓}}\lceil\cdot\rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}^{\mathbf{A},\mathsf{A},tbl,\mathbf{m_0},\mathsf{m}_0,\mathcal{A}_?,\mathcal{E}_!}$: $\mathbf{A}$ describes the instruction range that the compiled **Cap** program occupies, A describes the set of function names of the wrapped library, *tbl* describes a map from Rec function names to **Cap** entry points (*i.e.*, enter capabilities), while $\mathbf{m_0}$ and m_0 describe the layout of the initial **Cap** heap and Rec memory. $\mathcal{A}_?$ is the address space of the untrusted code and $\mathcal{E}_!$ is the set of enter capabilities that the untrusted code can have access to. (The last two are explained in more detail in §6.2.)

The 🔒/🔓 parameter of the wrapper is necessary to deal with a tension between the two use-cases of the wrapper: On the one hand, it is used in the compiler correctness statement $[\![\downarrow \mathsf{R}]\!]_{\mathbf{c}} \preceq {}^{\text{🔒}}\lceil [\![\mathsf{R}]\!]_{\mathsf{r}} \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}$ (Theorem 5.1) Here the wrapper should express the hardening properties of the compiler, in particular that it clears all non-argument registers (see the discussion of *register save* from Fig. 8b above). However, the wrapper is also used in the semantic back-translation statement $\mathbf{UNIV} \preceq {}^{\text{🔓}}\lceil \mathsf{SIM} \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}$. Crucially, **UNIV** is a universal specification of *all* capability machine programs, and not just those *emitted by the compiler.* Thus, in this case, the wrapper should *not* include the hardening properties of our compiler. The wrapper distinguish these two cases with the 🔒/🔓 parameter: 🔒 means that the wrapped code is hardened, while 🔓 translates the events without enforcing hardening.

One important property of $\lceil\cdot\rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}$ is how it interacts with linking: After applying the compiler correctness and semantic back-translation, we have ${}^{\text{🔒}}\lceil [\![\mathsf{P}]\!]_{\mathsf{r}} \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}$ and ${}^{\text{🔓}}\lceil \mathsf{SIM} \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}$ linked by $\oplus_{\mathbf{c}}$ (see Step (9) in Fig. 4). To convert this to Rec, we need to turn the **Cap** linking $\oplus_{\mathbf{c}}$ into the Rec linking $\oplus_{\mathsf{r}}$. For this, we use the following lemma:

Lemma 5.2 (Linking of Wrappers).
(1) ${}^{\text{🔒}}\lceil \mathsf{M}_1 \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}} \oplus_{\mathbf{c}} {}^{\text{🔒}}\lceil \mathsf{M}_2 \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}} \preceq {}^{\text{🔒}}\lceil \mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2 \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}$
(2) ${}^{\text{🔒}}\lceil \mathsf{M}_1 \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}} \oplus_{\mathbf{c}} {}^{\text{🔓}}\lceil \mathsf{M}_2 \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}} \preceq {}^{\text{🔓}}\lceil \mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2 \rceil_{\mathsf{r}\rightleftarrows\mathbf{c}}$

For the aforementioned case of linking P and SIM we use case (2) of the lemma: Since SIM is not hardened, the combined program is also not hardened. Case (1) is used for linking two hardened programs, for example two functions that were separately compiled by the hardening compiler.

$$\mathsf{inv}^{ps}_{\mathsf{SIM}}(\mathsf{m}) \triangleq \mathsf{SI}_{\mathsf{Rec}}(\mathsf{m}) * \exists L.\, \mathsf{LowAuth}(L) * \mathop{\Huge *}_{\mathsf{b} \in \mathsf{m} \setminus ps} \exists D.\, (\mathsf{b}, 0) \rightarrowtail_{\circ} D * \text{if } \mathsf{b} \in L\text{: } \mathop{\Huge *}_{i \in D} \exists \mathsf{v}.\, \ell_i \mapsto \mathsf{v} * \mathsf{low}(\mathsf{v})$$

$$\mathsf{pre}^{\mathcal{F}_?}_{\mathsf{SIM}} \xrightarrow{\mathsf{e}} \triangleq \exists \overline{\mathsf{v}}, \mathsf{m}.\, (\exists \mathsf{f} \in \mathcal{F}_?.\, \mathsf{e} = \mathsf{Call?}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}) \wedge n_{\mathsf{f}} = |\overline{\mathsf{v}}|) \vee (\mathsf{e} = \mathsf{Return?}(\mathsf{v}, \mathsf{m}))$$
$$* \mathop{\Huge *}_{\mathsf{v} \in \overline{\mathsf{v}}} \mathsf{low}(\mathsf{v}) * \mathsf{inv}^{\emptyset}_{\mathsf{SIM}}(\mathsf{m}) \qquad \mathsf{init}^{\mathsf{m}_0}_{\mathsf{SIM}} \triangleq \mathop{\Huge *}_{(\ell, \mathsf{v}) \in \mathsf{m}_0} \ell \mapsto \mathsf{v}$$

$$\mathsf{post}^{\mathcal{F}_!}_{\mathsf{SIM}} \xleftarrow{\mathsf{e}} \triangleq \exists \overline{\mathsf{v}}, \mathsf{m}.\, (\exists \mathsf{f} \in \mathcal{F}_!.\, \mathsf{e} = \mathsf{Call!}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}) \wedge n_{\mathsf{f}} = |\overline{\mathsf{v}}|) \vee (\mathsf{e} = \mathsf{Return!}(\mathsf{v}, \mathsf{m}))$$
$$* \mathop{\Huge *}_{\mathsf{v} \in \overline{\mathsf{v}}} \mathsf{low}(\mathsf{v}) * \mathsf{inv}^{\emptyset}_{\mathsf{SIM}}(\mathsf{m})$$

Fig. 9. Definition of $\mathsf{SIM}^{\mathcal{F}_?, \mathsf{m}_0}_{\mathcal{F}_!}$ parameterized over a set $\mathcal{F}_?$ of function names SIM listens to, a set $\mathcal{F}_!$ of function names SIM can call, and the initial memory m_0 for static variables. $\ell \rightarrowtail_{\circ} D$ is the complementary fraction to $\ell \rightarrowtail D$, *i.e.*, we have $\ell \rightarrowtail_{\circ} D * \ell \rightarrowtail D' \vdash D = D'$.

Proof of Theorem 5.1. With this wrapper definition, the proof of compiler correctness is mostly straightforward. For each pass inherited from DimSum, we apply the existing correctness theorem. For *Codegen*, we prove correctness from scratch. To do this, we must show that the events emitted by the target program are emitted by the source program. We prove this by stepping through the compiled code (assuming it was called by an arbitrary module), while maintaining various invariants over memory, both local and shared, between calls. Note that by the definition of $^{🔒}\lceil\cdot\rceil_{\mathsf{r} \rightleftharpoons \mathbf{c}}$, part of the proof involves showing that the compiler indeed hardens the source program by clearing non-argument and non-return registers.

6 Simulator and Universal Contract

This section introduces the two semantic characterizations of the untrusted code **un** we use: First, the simulator SIM (§6.1) that represents **un** at the Rec level. Then, the universal contract **UNIV** (§6.2), a semantic **Cap** module that over-approximates arbitrary **Cap** code. Finally, §6.3 presents the *semantic back-translation* establishing that SIM (wrapped in $\lceil\cdot\rceil_{\mathsf{r} \rightleftharpoons \mathbf{c}}$) simulates **UNIV**.

6.1 Simulator for Rec

As discussed in §3, the simulator SIM is a *semantic*, Rec-level description of the **Cap**-level untrusted code. Concretely, we use the fact that DimSum's semantic linking operator $\oplus_{\mathsf{r}}$ can link not just syntactic Rec modules, but arbitrary modules (*i.e.*, labeled transition systems) that interact using the Rec events Call and Return. This allows us to construct SIM semantically without building a syntactic Rec program (which, as noted in §1.3, would be impossible).

Concretely, we define SIM using DimSum's ability to encode separation logic pre- and post-conditions into labeled transition systems (by leveraging angelic non-determinism, inspired by CCR [58]). Intuitively, the idea is to define a semantic module that assumes an incoming event fulfilling the precondition and non-deterministically picks an outgoing event fulfilling the post-condition. DimSum uses this technique to encode wrappers like $\lceil\cdot\rceil_{\mathsf{r} \rightleftharpoons \mathbf{c}}$. We observe that we can also use it to define stand-alone modules like SIM (and **UNIV**). As we will see in this section, this technique allows us to define SIM as basically a direct encoding of the WP-CALL-UN rule of Reckon.

To define a semantic module like SIM, we follow a three-part recipe: First, we pick a separation logic assertion language to define SIM with. Luckily, we already have the perfect assertion language for this purpose: recProp, the assertion language of our program logic Reckon (§2.2), providing the $\ell \mapsto \mathsf{v}$ and $\mathsf{low}(\mathsf{v})$ assertions, among other. Second, we define the invariant and initial ownership

$$\mathbf{low}^{\mathbf{shd}}(z) \triangleq \text{True} \quad \mathbf{low}^{\mathbf{shd}}((\mathbf{E}, b, e, a)) \triangleq \text{if } a \notin \mathcal{A}_? \text{ then } (\mathbf{E}, b, e, a) \in \mathcal{E}_! \text{ else } \mathbf{low}^{\mathbf{shd}}((\mathbf{RX}, b, e, a))$$

$$\mathbf{low}^{\mathbf{shd}}((\mathbf{p}, b, e, a)) \triangleq \exists b' \leq b, e \leq e'.\, (b', e') \mapsto ? \land (\mathbf{RX} \preccurlyeq \mathbf{p} \to [b, e) \subseteq \mathcal{A}_?)$$

$$\mathbf{low}^{\mathbf{shd}}(\mathbf{Stk}(d, \mathbf{p}, b, e, a)) \triangleq \mathbf{p} \preccurlyeq \mathbf{RW} * \exists \iota.\, \mathbf{shd}[d] = \iota * \exists b' \leq b, e \leq e'.\, (\iota, b', e') \mapsto ?$$

$$\mathbf{sb} ::= (b, e) \mid (\iota, b, e) \qquad \mathbf{sb} \mapsto ? * \mathbf{LowAuth}(L) \vdash \mathbf{sb} \in L \qquad \mathbf{sb} \mapsto ? \vdash \mathbf{sb} \mapsto ? * \mathbf{sb} \mapsto ?$$

Fig. 10. Definition of $\mathbf{low}^{\mathbf{shd}}$ (implicitly parametrized over $\mathcal{E}_!$ and $\mathcal{A}_?$)

of SIM. And finally, we define the pre- and postcondition (pre_{SIM} and post_{SIM}) of SIM using the invariant and recProp.

So let us introduce the invariant $\text{inv}^{ps}_{\text{SIM}}(\text{m})$ of SIM shown in Fig. 9. It consists of three parts: First, $\text{SI}_{\text{Rec}}(\text{m})$ links the points-to and domain predicates $\ell \mapsto \text{v}$ and $\ell \rightarrowtail_\circ D$ to the current memory m. Second, $\text{LowAuth}(L)$ tracks the authoritative set L of all shared blocks. Concretely, it gives rise to the $\text{b} \mapsto ?$ predicate that states that the block id b is low. $\text{b} \mapsto ?$ is used to define $\text{low}(\ell)$:

$$\text{low}(\ell) \triangleq \ell.\text{id} \mapsto ? \qquad \text{b} \mapsto ? * \text{LowAuth}(L) \vdash \text{b} \in L \qquad \text{b} \mapsto ? \vdash \text{b} \mapsto ? * \text{b} \mapsto ?$$

Finally, $\text{inv}^{ps}_{\text{SIM}}(\text{m})$ contains the points-to predicates for all low blocks (*i.e.*, , all blocks in L) and ensures that all low locations contain low values. The initial ownership $\text{init}^{\text{m}_0}_{\text{SIM}}$ is parameterized by an initial memory m_0 and contains the points-to predicates for all locations in m_0.

Now all that is left to define SIM is to provide its pre- and postcondition as recProp relations, shown in Fig. 9: The precondition $\text{pre}^{\mathcal{F}_?}_{\text{SIM}} \xrightharpoonup{\text{e}}$ states the property one has to prove when calling (or returning to) the simulator with the event e. Dually, the postcondition $\text{post}^{\mathcal{F}_!}_{\text{SIM}} \xleftharpoondown{\text{e}}$ describes the property one obtains from the simulator when *it* performs a call or returns with event e.

Let us first focus on pre_{SIM}. There are two cases to consider: Either, one can call SIM by invoking one of the functions in $\mathcal{F}_?$—*i.e.*, e is Call?. Then the number of arguments $|\overline{\text{v}}|$ must match the expected number of arguments n_{f}. Or, one can return to SIM after a previous call from SIM—*i.e.*, e is Return?. In both cases, all values (*i.e.*, arguments or return value) must be low, matching the precondition of wp-call-un, and the invariant $\text{inv}^{ps}_{\text{SIM}}(\text{m})$ must hold for the memory m. post_{SIM} is analogous to pre_{SIM} except that the simulator can only call functions in $\mathcal{F}_!$. post_{SIM} also ensures that all values provided by SIM are low, again following wp-call-un.

In summary, we end up with the definition of $\text{SIM}^{\mathcal{F}_?,\text{m}_0}_{\mathcal{F}_!}$ that is parameterized by the functions $\mathcal{F}_?$ where $\text{SIM}^{\mathcal{F}_?,\text{m}_0}_{\mathcal{F}_!}$ accepts calls, the functions $\mathcal{F}_!$ that $\text{SIM}^{\mathcal{F}_?,\text{m}_0}_{\mathcal{F}_!}$ can call, and the initial memory m_0.

6.2 Universal Contract for Cap

This section defines UNIV, a semantic Cap module that gives a universal contract satisfied by arbitrary untrusted Cap code. The intuition for UNIV is similar to SIM, except that it is for Cap instead of Rec. This has three main consequences: First, instead of recProp, UNIV is based on the separation logic capProp with points-to predicates for the Cap heap and stack. Second, we need to provide a definition for $\mathbf{low}(\mathbf{w})$ that works on Cap words $\mathbf{w}$ (instead of $\text{low}(\text{v})$ used by SIM). Third, we cannot base UNIV on wp-call-un, but instead take inspiration for similar call rules for capability machines in prior work [55, 56, 20, 21]. Let us now expand on the second and third points.

Defining low. The low predicate, describing when a Cap word is safe to share with the untrusted code, is analogous to low of Rec. The definition of low is given in Fig. 10. Since there are four different kinds of words including different kinds of capabilities, we need to distinguish four cases: integers (z), enter capabilities (**E**), heap capabilities (**p**), and stack capabilities (**Stk**).

First, integers are trivially **low**, as in Rec.

The second case concerns enter capabilities **E** (introduced in §4). They do not exist in Rec, but it is crucial to limit the **E**-capabilities that untrusted code has access because it should only be able to jump to the correct entry points of trusted functions. We therefore (implicitly) parameterize the definition over a set $\mathcal{E}_!$ of **E**-capabilities that untrusted code is allowed to have access to (playing a similar role to $\mathcal{F}_!$ in the definition of SIM). If the address a of the enter capability is outside of **UNIV**'s instruction region $\mathcal{A}_?$, it must be in the set $\mathcal{E}_!$. If the address a is in $\mathcal{A}_?$, it is treated like a **RX** capability (discussed in the next paragraph).

The third case is for other capabilities that give read or write access to the heap. These are modeled similarly to low locations in SIM: We introduce a resource **sb** $\mapsto$? in **capProp** that tracks what memory ranges[10] **sb** have been shared with **UNIV**. **LowAuth**(L) is the authoritative view that the set of all shared memory ranges is L. We then define that a capability is **low** if an enclosing memory range is shared with **UNIV**. One interesting special case are **R(W)X**-capabilities: The untrusted code can derive arbitrary **E**-capabilities from the **RX** and **RWX**-capabilities it has (since **E** is below **RX** in the permission order in Fig. 5). Thus, **low** must ensure that the ranges for these permissions are always $\mathcal{A}_?$. This prevents the untrusted code from creating new **E**-capabilities that point outside of its instruction addresses.

The fourth and last case are stack capabilities $\mathbf{Stk}(d, \mathbf{p}, b, e, a)$, which are a little more subtle because they give access to some memory range $[b, e)$ on the dth stack frame while it is alive. Since the dth stack frame might get popped and reused later, we cannot give **UNIV** persistent access to the dth stack frame. We therefore give stack frames unique shadow ids ι that are mapped to the physical stack frame number d by a shadow stack **shd**. A stack capability is $\mathbf{low}^{\mathbf{shd}}$ if an enclosing memory range tagged with the current shadow id ι of the stack frame d is shared with **UNIV**.

Defining UNIV. To see how we define **UNIV**, let us take a step back: Recall that the definition of SIM very closely mirrors the wp-call-un rule for calling untrusted code at the Rec level. Unfortunately, we cannot directly use wp-call-un for **UNIV** since **UNIV** is defined at the **Cap** level instead of Rec. But we are in luck: there is a growing body of prior work on program logics for capability machines that provide rules for calling untrusted code analogous to wp-call-un [55, 56, 20, 21]. These rules follow the same high-level structure as wp-call-un: **low** values in and **low** values out. However, the details are significantly more involved due to the handling of the stack and well-bracketed control flow (in particular, requiring subtle future world relations to control how the stack may evolve). Our **UNIV** takes these ideas and encodes them into a semantic DimSum module. For this, we define an invariant $\mathsf{inv}_{\mathbf{UNIV}}$ analogous to $\mathsf{inv}_{\mathsf{SIM}}$ that integrates **LowAuth**(L) analogous to $\mathsf{LowAuth}(L)$, in particular ensuring that low memory regions only contain low values. $\mathsf{pre}_{\mathbf{UNIV}}$ and $\mathsf{post}_{\mathbf{UNIV}}$ follow the structure of $\mathsf{pre}_{\mathsf{SIM}}$ and $\mathsf{post}_{\mathsf{SIM}}$, but include a future world relation for enforcing well-bracketedness of the stack. The full definition is given in the appendix [43] and formalized in the Rocq development—we omit the details here since they are not important for the rest of the discussion and follow prior work.

We show that our definition of **UNIV** is indeed refined by arbitrary untrusted code **un**:

Theorem 6.1 (Universal Contract). *For any set of entry points $\mathcal{E}_!$ allowed to be shared, we have*

$$\forall \mathbf{un}.\ [\![\mathbf{un}]\!]_{\mathbf{c}} \preceq \mathbf{UNIV}_{\mathcal{E}_!}^{\mathsf{dom}(\mathbf{un})}$$

[10]We track this on the level of memory ranges instead of individual addresses to be closer to Rec which has memory blocks and to simplify the semantic back-translation proof.

$$z \Leftrightarrow i \triangleq z = i \qquad \ell \Leftrightarrow (\mathbf{E}, b, e, a) \triangleq \text{if } a \in \mathcal{A}_? \text{ then } [b, e) \subseteq \mathcal{A}_? \text{ else } (\mathbf{E}, b, e, a) \in \mathcal{E}_!$$
$$\ell \Leftrightarrow (\mathbf{p}, b, e, a) \triangleq \exists b' \leq b, e \leq e'.\, \ell.\mathsf{id} \leftrightarrow (b', e') * (\mathbf{RX} \preccurlyeq \mathbf{p} \rightarrow [b, e) \subseteq \mathcal{A}_?)$$
$$\ell \Leftrightarrow \mathbf{Stk}(d, \mathbf{p}, b, e, a) \triangleq \mathbf{p} \preccurlyeq \mathbf{RW} * \exists \iota.\, \mathbf{shd}[d] = \iota * \exists b' \leq b, e \leq e'.\, \ell.\mathsf{id} \leftrightarrow (\iota, b', e')$$

Fig. 11. Value relation of $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$ parameterized over $\mathcal{A}_?$, $\mathcal{E}_!$, and **shd** also used in the definition of UNIV where $\mathsf{id} \leftrightarrow \mathbf{sb}$ is a resource to persistently track which source memory blocks are translated to which target memory ranges.

Proof. The proof considers all instructions of Cap and proves that they maintain the invariant $\mathsf{inv}_{\mathrm{UNIV}}$, only generate **low** outputs from **low** inputs and maintain well-bracketed control flow. □

6.3 Semantic Back-Translation

We now turn to the core of RobustDimSum: the *semantic back-translation* $\mathrm{UNIV} \preceq \lceil \mathsf{SIM} \rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$. It allows us to lift the reasoning about the untrusted code from Cap to Rec. It is also where all the work from the prior sections pays off: Since UNIV and SIM are defined purely semantically based on separation logic, we do not need to manipulate any program syntax in this proof!

Let us now state the semantic back-translation theorem. Although it contains several preconditions, none of them should be that surprising. Importantly, the statement restricts the initial memory of untrusted Cap code to not contain any capability that it should not have access to.

Theorem 6.2 (Semantic Back-translation). *Let $\mathcal{A}_?$ be a contiguous non-empty address space. Let $\mathcal{F}_?$ correspond to $\mathcal{A}_?$ and $\mathcal{F}_!$ corresponded to $\mathcal{E}_!$ according to the linking table tbl. Let $\mathbf{m_0}$ be an arbitrary initial memory with* $\mathrm{dom}(\mathbf{m_0}) = \mathcal{A}_?$ *that only contains* **E***-capabilities within $\mathcal{E}_!$ and all other capabilities within $\mathcal{A}_?$. Let m_0 contain values corresponding to the words in $\mathbf{m_0}$. Then we have:*

$$\mathrm{UNIV}^{\mathcal{A}_?}_{\mathcal{E}_!} \preceq 🔓 \lceil \mathsf{SIM}^{\mathcal{F}_?, \mathsf{m}_0}_{\mathcal{F}_!} \rceil^{\mathcal{A}_?, \mathcal{F}_?, tbl, \mathbf{m_0}, \mathsf{m}_0, \mathcal{A}_?, \mathcal{E}_!}_{\mathsf{r}\rightleftharpoons\mathbf{c}}$$

More about $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$. To discuss Theorem 6.2 and its proof in more detail, we first need to give some more details about the wrapper $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$: It is defined using separation logic pre- and postconditions $\mathsf{pre}_{\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}}/\mathsf{post}_{\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}}$, similar to SIM and UNIV. These are defined in the separation logic r2cProp that features the points-to predicates from both recProp and **capProp**. The key predicate of r2cProp is the *value relation* $\mathsf{v} \Leftrightarrow \mathbf{w}$ shown in Fig. 11, which relates Rec values v that are passed across the wrapper to Cap words **w** and vice versa. We will discuss the value relation more in a moment when we talk about relating low to **low**.

Relating Theorem 6.2 to logical relations. As discussed in §6.1 and §6.2, SIM and UNIV can be seen as encodings of (unary) logical relations for reasoning about untrusted code into DimSum modules. Also $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$ can be seen as a DimSum version of a binary logical relation for relating Rec and Cap with value relation $\mathsf{v} \Leftrightarrow \mathbf{w}$. However, usually logical relations cannot be related to each other—*e.g.*, it is unclear how one would relate the logical relation from OCPL [64] that SIM is based on to the logical relation of Cerise [18] that UNIV is based on. By encoding everything in DimSum, we can in fact state and prove Theorem 6.2 and thus lift the reasoning from Cap to Rec.

Proving Theorem 6.2. Let us now give an intuition for the proof of Theorem 6.2. At the high-level, we need to show two things: On the one hand, given an incoming Rec-event and a corresponding Cap-event we can assume $\mathsf{pre}_{\mathsf{SIM}}$ and $\mathsf{pre}_{\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}}$ and need to show $\mathsf{pre}_{\mathrm{UNIV}}$. On the other hand, for an outgoing Cap-event we can assume $\mathsf{post}_{\mathrm{UNIV}}$ and need to back-translate the event to a Rec-event and show $\mathsf{post}_{\mathsf{SIM}}$ and $\mathsf{post}_{\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}}$.

The main challenge of these proofs is handling arbitrary dynamic pointer sharing. Concretely, this challenge shows up when we need to convert the low of SIM to the $\mathbf{low}$ of UNIV and vice versa for the arguments and return values (and all shared values in memory). Intuitively, we need a lemma like the following:[11]

$$\exists \mathsf{v}.\ \mathsf{low}(\mathsf{v}) * \mathsf{v} \Leftrightarrow \mathbf{w} \dashv\vdash \mathbf{low}(\mathbf{w}) \tag{14}$$

When reasoning about the preconditions, Eq. (14) allows us to convert the $\mathsf{low}(\mathsf{v})$ we obtain from $\mathsf{pre}_{\mathsf{SIM}}$ and the $\mathsf{v} \Leftrightarrow \mathbf{w}$ we obtain from $\mathsf{pre}_{\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}}$ to the $\mathbf{low}(\mathbf{w})$ we need for $\mathsf{pre}_{\mathbf{UNIV}}$. For the other direction, we obtain $\mathbf{low}(\mathbf{w})$ from $\mathsf{post}_{\mathbf{UNIV}}$ and use Eq. (14) to split it into $\mathsf{v} \Leftrightarrow \mathbf{w}$ for $\mathsf{post}_{\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}}$ and $\mathsf{low}(\mathsf{v})$ for $\mathsf{post}_{\mathsf{SIM}}$. (Recall that we have UNIV on one side of the refinement vs. SIM and $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$ on the other.) To see why Eq. (14) holds, it is instructive to compare the definition of $\mathsf{v} \Leftrightarrow \mathbf{w}$ (Fig. 11) with the definition of $\mathsf{low}(\mathsf{v})$ (§6.1) and $\mathbf{low}(\mathbf{w})$ (Fig. 10): The structure of $\mathsf{v} \Leftrightarrow \mathbf{w}$ and $\mathbf{low}(\mathbf{w})$ match exactly, with the main difference being that $\mathbf{low}(\mathbf{w})$ uses $\mathbf{sb} \mapsto ?$ to track shared memory ranges while $\mathsf{v} \Leftrightarrow \mathbf{w}$ uses a bijection $\mathsf{b} \leftrightarrow \mathbf{sb}$ that maps a shared memory range $\mathbf{sb}$ to a Rec block identifier b. Thus, proving Eq. (14) reduces to maintaining the following invariant:

$$\exists \mathsf{b}.\ \mathsf{b} \mapsto ? * \mathsf{b} \leftrightarrow \mathbf{sb} \dashv\vdash \mathbf{sb} \mapsto ? \tag{15}$$

The proof of Theorem 6.2 is thus based around maintaining Eq. (15). Toward this end, whenever one of UNIV, SIM, or $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathbf{c}}$ shares a new location or capability, the ghost state of the others must be updated to share it as well. The full proof can be found in the accompanying Rocq development [43]. It is worth noting that, although there were clearly many subtle details that had to be worked out in order to formalize our semantic back-translation result, the proof itself (once all definitions had been set up) was completed within under two person weeks.

7 Putting It All Together

We now have all the pieces we need to obtain our desired robust safety result, and in this section we finally show how to put them together. First, §7.1 presents the soundness statement of Reckon w.r.t. SIM. Then, finally, §7.2 presents the final robust safety result about the compiled code.

7.1 Soundness of Reckon w.r.t. SIM

We prove the following soundness theorem for our program logic Reckon (§2.2) w.r.t. SIM:[12]

Theorem 7.1 (Soundness of Reckon w.r.t. SIM). *Given a program* P *that we want to prove robustly safe, primitives* $\mathsf{prim}_{\mathsf{spec}}$ *(e.g.,* $\mathsf{to\_int}_{\mathsf{spec}}$ *from §3), preconditions* $\mathit{pre}_{\mathsf{f}}$ *and postconditions* $\mathit{post}_{\mathsf{f}}$ *for the functions* f *in* P *and* $\mathsf{prim}_{\mathsf{spec}}$*, a set of predicates* Cond*, a set of functions* $\mathcal{F}_?$ *that belong to the untrusted code, a subset of functions* $\mathcal{F}_! \subseteq$ P *that the untrusted code can call, and an initial memory* m_0 *of the untrusted code, satisfying the following constraints:*

(1) For all $\mathsf{fn}\ \mathsf{f}(\overline{x}) \triangleq \mathsf{e} \in \mathsf{P}$*, assuming the triple* $\forall \overline{\mathsf{v}}.\ \{\mathit{pre}_{\mathsf{f}_1}(\overline{\mathsf{v}})\}\ \mathsf{f}_1(\overline{\mathsf{v}})\ \{\mathit{post}_{\mathsf{f}_1}\}^{\mathsf{Cond}}$ *for all functions* f_1 *in* P *and* $\mathsf{prim}_{\mathsf{spec}}$*, and assuming* $\mathsf{UnFn}(\mathsf{f}_2, n_{\mathsf{f}_2})$ *for all functions* f_2 *with* n_{f_2} *arguments in* $\mathcal{F}_?$*, one has proven the triple*[13]

$$\forall \overline{\mathsf{v}}.\ \{\mathit{pre}_{\mathsf{f}}(\overline{\mathsf{v}})\}\ \mathsf{e}[\overline{\mathsf{v}}/\overline{x}]\ \{\mathit{post}_{\mathsf{f}}\}^{\mathsf{Cond}}$$

(2) For all functions f *in* $\mathsf{prim}_{\mathsf{spec}}$*, one has proven the triple*

$$\forall \overline{\mathsf{v}}.\ \{\mathit{pre}_{\mathsf{f}}(\overline{\mathsf{v}})\}\ \mathsf{f}(\overline{\mathsf{v}})\ \{\mathit{post}_{\mathsf{f}}\}^{\mathsf{Cond}}$$

[11]This statement serves as an intuition, the actual lemma is more involved. Among other, we need to embed recProp, capProp, and r2cProp into a unified separation logic such that we can talk about $\mathsf{low}(\mathsf{v})$, $\mathsf{v} \Leftrightarrow \mathbf{w}$, and $\mathbf{low}(\mathbf{w})$ in the same separation logic.

[12]We omit disjointness constraints on function names.

[13]We omit the handling of local and static variables for simplicity.

(3) For all functions f *with* n_{f} *arguments in* $\mathcal{F}_!$, *we have*

$$\forall \overline{\mathsf{v}}.\ |\overline{\mathsf{v}}| = n_{\mathsf{f}} * \mathop{\Huge\ast}_{\mathsf{v}\in\overline{\mathsf{v}}} \mathsf{low}(\mathsf{v}) \vdash \mathit{pre}_{\mathsf{f}}(\overline{\mathsf{v}}) \quad \textit{and} \quad \forall \mathsf{v}.\ \mathit{post}_{\mathsf{f}}(\mathsf{v}) \vdash \mathsf{low}(\mathsf{v})$$

(4) $\mathtt{main} \in \mathsf{P}$ *with* $\mathit{pre}_{\mathtt{main}}(\overline{\mathsf{v}}) \triangleq (\overline{\mathsf{v}} = [])$ *and* $\mathit{post}_{\mathtt{main}}(\mathsf{v}) \triangleq (\mathsf{v} = 0)$
(5) $\vdash \exists \Phi.\ \mathsf{Cond}(\Phi) * \Phi$
Then we have

$$[\![\mathsf{P}]\!]_{\mathsf{r}} \oplus_{\mathsf{r}} \mathsf{prim}_{\mathsf{spec}} \oplus_{\mathsf{r}} \mathsf{SIM}_{\mathcal{F}_!}^{\mathcal{F}_?,\mathsf{m}_0} \preceq \mathsf{safe}_{\mathsf{r}}^{\mathsf{m}_0}$$

Theorem 7.1 proves that P, $\mathsf{prim}_{\mathsf{spec}}$ and $\mathsf{SIM}_{\mathcal{F}_!}^{\mathcal{F}_?,\mathsf{m}_0}$ form a safe, closed program. This is represented by a refinement to the following Spec program:[14]

$$\mathsf{safe}_{\mathsf{r}}^{\mathsf{m}_0} \triangleq \exists \mathsf{f}, \overline{\mathsf{v}}, \mathsf{m};\mathsf{vis}(\mathsf{Call?}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}));\mathsf{assume}(\mathsf{f} = \mathtt{main} \wedge \overline{\mathsf{v}} = [] \wedge \mathsf{m}_0 \subseteq \mathsf{m}); \exists \mathsf{m}';\mathsf{vis}(\mathsf{Return!}(0, \mathsf{m}'));$$

$\mathsf{safe}_{\mathsf{r}}^{\mathsf{m}_0}$ states that if the `main` function is called without arguments in an initial heap containing m_0, it will return 0. Implicit in this spec is the property that `main` will *not* perform any calls to undefined functions as those would result in unresolved call events. In particular, this implies `main` does not execute any failed assertions since they would invoke an undefined `assert_failed` function.

To apply Theorem 7.1, we need to verify the functions in P using Reckon (1). We can assume Hoare triples for all functions in P, $\mathsf{prim}_{\mathsf{spec}}$ and SIM, allowing arbitrary (mutual) recursion. We also need to prove the Hoare triples for $\mathsf{prim}_{\mathsf{spec}}$ (2), which can be done by unfolding the definition of Hoare triples. Additionally, we need to show that all functions exposed to the untrusted code (*i.e.*, the functions in $\mathcal{F}_!$), are safe to call when passed low arguments and then return a low result (3). Also, P needs to contain a `main` function with trivial precondition and a postcondition that ensures that it returns 0 (4). Finally, we need to provide the initial Φ that fulfills $\mathsf{Cond}(\Phi)$ to kick off the well-bracketedness reasoning described in §2.2 (5).

Proving Theorem 7.1. To prove Theorem 7.1, we essentially need an adequate model of Reckon's weakest-pre assertions $\mathsf{wp}^{ps}\ \mathsf{e}\ \{\Phi\}$. This is slightly challenging in that the behavior of Rec programs depends integrally on their environment, so the standard "context-free" model of weakest-pre in Iris does not apply. Instead, we apply the standard technique of $\top\top$-closure (aka biorthogonality) [13]. Concretely, we first define $\mathsf{wp}^{ps}\ \mathsf{e}$, a version of $\mathsf{wp}^{ps}\ \mathsf{e}\ \{\Phi\}$ *without* the postcondition:[15]

$$\mathsf{wp}^{ps}\ \mathsf{e} \triangleq \forall \mathsf{m}.\ \mathsf{inv}_{\mathsf{SIM}}^{ps}(\mathsf{m}) \twoheadrightarrow \Big(\mathop{\ast}_{\mathsf{b}\in ps} \exists D.\ \mathsf{b} \rightarrowtail_{\circ} D\Big) \twoheadrightarrow \mathsf{wsat} \twoheadrightarrow$$
$$[\![(\mathsf{e}, \mathsf{m}) \mathbin{@} \mathsf{P}]\!]_{\mathsf{r}} \oplus_{\mathsf{r}} \mathsf{prim}_{\mathsf{spec}} \oplus_{\mathsf{r}} \mathsf{SIM}_{\mathcal{F}_!}^{\mathcal{F}_?,\mathsf{m}_0} \preceq \mathsf{safe}_{\mathsf{r}}^{\mathsf{m}_0}$$

That is, given a memory m satisfying $\mathsf{inv}_{\mathsf{SIM}}^{ps}(\mathsf{m})$ and the world-satisfaction wsat [32] that tracks ownership of the invariants, we have the refinement at the conclusion of Theorem 7.1, except that we currently execute the expression e with memory m in P. (This is denoted by $(\mathsf{e}, \mathsf{m}) \mathbin{@} \mathsf{P}$.) Using $\top\top$-closure, we then define $\mathsf{wp}^{ps}\ \mathsf{e}\ \{\Phi\}$ to mean that e behaves safely under all safely-behaving evaluation contexts K:

$$\mathsf{wp}^{ps}\ \mathsf{e}\ \{\Phi\} \triangleq \forall \mathsf{K}.\ (\forall \mathsf{v}.\ \Phi(\mathsf{v}) \twoheadrightarrow \mathsf{wp}^{[]}\ \mathsf{K}[\mathsf{v}]) \twoheadrightarrow \mathsf{wp}^{ps}\ \mathsf{K}[\mathsf{e}]$$

It is straightforward to prove all the rules in §2.2 under this definition—most rules just execute e according to its Rec operational semantics and do not require any reasoning about DimSum linking. The only exception is WP-CALL-UN where control switches to SIM. However, proving this rule is also straightforward since SIM was *defined* to directly match WP-CALL-UN.

[14]vis means that the program emits a visible event. Here, first for receiving an incoming call, then an outgoing return.
[15]For technical reasons pertaining to how DimSum encodes SIM into a state transition system, the refinement does not use the usual embedding of pure propositions into separation logic, but instead $\mathsf{sat}\sharp(P)$, which is the right adjoint of the satisfiable relation $\mathsf{sat}(Q)$ [59, p. 108], *i.e.*, $(Q \vdash \mathsf{sat}\sharp(P))$ is equivalent to $(\mathsf{sat}(Q) \to P)$.

The rest of the proof of Theorem 7.1 is straightforward. We define $\mathsf{UnFn}(f, n_f)$ as the Hoare triple in the conclusion of WP-CALL-UN. Then we prove two nested inductions: one to resolve mutual recursion between functions in P, and another to resolve mutual recursion between P and SIM.

7.2 Robust Safety

Before we can compose all the refinements (as shown in Fig. 4 from §3), we need to prove one last missing refinement: lifting the safety result safe_r from Rec to Cap. For this, we define $\mathbf{safe}_c^{\mathbf{m}_0}$, as a specification for safe, closed Cap programs, analogous to safe_r, assuming a fixed entry capability **main** for the main function. Like $\mathsf{safe}_r^{m_0}$, $\mathbf{safe}_c^{\mathbf{m}_0}$ is parameterized by the initial memory $\mathbf{m}_0$.

$$\mathbf{safe}_c^{\mathbf{m}_0} \triangleq \exists \mathbf{r}, \mathbf{m}, \mathbf{s}; \mathsf{vis}(\mathbf{Jump?}(\mathbf{r}, \mathbf{m}, \mathbf{s})); \mathsf{assume}(\mathbf{LangInv}(\mathbf{r}, \mathbf{m}, \mathbf{s}) \wedge \mathbf{r}(\mathbf{pc}) = \mathbf{main} \wedge \mathbf{m}_0 \subseteq \mathbf{m});$$
$$\exists \mathbf{r}', \mathbf{m}', \mathbf{s}'; \mathsf{assert}(\mathbf{r}'(\mathbf{pc}) = \mathbf{s}.\mathit{cont} \wedge \mathbf{r}'(\mathbf{x0}) = 0); \mathsf{vis}(\mathbf{Jump!}(\mathbf{r}', \mathbf{m}', \mathbf{s}'));$$

The predicate $\mathbf{LangInv}(\mathbf{r}, \mathbf{m}, \mathbf{s})$ encodes that the registers $\mathbf{r}$, heap $\mathbf{m}$, and stack $\mathbf{s}$ in the initial state satisfy the well-formedness invariants of Cap: there are no dangling pointers, the stack is directed (*i.e.*, stack capabilities point to itself or lower stack frames), and the heap contains no stack capabilities. It is straightforward to prove that $\mathsf{safe}_r^{m_0}$ refines $\mathbf{safe}_c^{\mathbf{m}_0}$. (In the rest of this section, we fix a mapping from function names to entry points *tbl*, the domain of the untrusted code $\mathcal{A}_?$, and the enter capabilities shared with the untrusted code $\mathcal{E}_!$. We omit these parameters from $\lceil \cdot \rceil_{r \rightleftharpoons \mathbf{c}}$.)

Theorem 7.2. *Assuming tbl maps* `main` *to* **main**, *we have*

$$🔓\lceil \mathsf{safe}_r^{m_0} \rceil_{r \rightleftharpoons \mathbf{c}}^{\mathbf{A}, A, \mathbf{m}_0, m_0} \preceq \mathbf{safe}_c^{\mathbf{m}_0}$$

Now we can sketch the final robust safety result:

Theorem 7.3 (Robust Safety). *Assume*
(1) P *has been verified using the program logic and we have the result of* Theorem 7.1, i.e.,

$$[\![\mathsf{P}]\!]_r \oplus_r \mathsf{prim}_{\mathsf{spec}} \oplus_r \mathsf{SIM}_{\mathcal{F}_!}^{\mathcal{F}_?, m_0} \preceq \mathsf{safe}_r^{m_0}$$

(2) a correct implementation **prim** *of the primitives in* $\mathsf{prim}_{\mathsf{spec}}$ *with instruction range* $\mathbf{A_x}$ *and function names* A_x *encoded in* $\mathbf{m_x}$, i.e.,

$$[\![\mathbf{prim}]\!]_c \preceq 🔒\lceil \mathsf{prim}_{\mathsf{spec}} \rceil_{r \rightleftharpoons \mathbf{c}}^{\mathbf{A_x}, A_x, \mathbf{m_x}, \emptyset}$$

(3) arbitrary untrusted code **un** *represented in an arbitrary initial memory* $\mathbf{m_u}$ *with* $\mathsf{dom}(\mathbf{m_u}) = \mathcal{A}_?$ *that only contains* **E***-capabilities within* $\mathcal{E}_!$ *and all other capabilities within* $\mathcal{A}_?$,
(4) and a bunch of disjointness conditions on address spaces, memory ranges and function names.
We define the initial memory $\mathbf{m}_0 \triangleq \mathbf{m_p} \uplus \mathbf{m_x} \uplus \mathbf{m_u}$ *where* $\mathbf{m_p}$ *is the initial memory of the compiled code* (§5). *Then we have*

$$[\![\downarrow \mathsf{P} \cup_{\mathbf{c}} \mathbf{prim} \cup_{\mathbf{c}} \mathbf{un}]\!]_c \preceq \mathbf{safe}_c^{\mathbf{m}_0}$$

Proof. The theorem follows from the theorems of the previous sections, following the outline in Fig. 4 from §3. □

Applying Theorem 7.3 to `password_check`. We can apply Theorem 7.3 to the `password_check` example from §1 by combining it with the `hash` function from §3 and a simple implementation of `main` and `read_hash_from_db`.

$$\mathsf{Pwd} \triangleq \mathtt{main} \cup_r \mathtt{password\_check} \cup_r \mathtt{hash} \cup_r \mathtt{read\_hash\_from\_db}$$

We obtain the following final robust safety result:

$$\forall \mathbf{un}.\ [\![\downarrow \mathsf{Pwd} \cup_{\mathbf{c}} \mathbf{to\_int} \cup_{\mathbf{c}} \mathbf{un}]\!]_c \preceq \mathbf{safe}_c^{\mathbf{m}_0}$$

8 Related Work

Compartmentalizing compilation. Prior work on compartmentalizing compilation studies the compilation and security of C-like unsafe source languages that have built-in compartments, with the scope of undefined behavior restricted to individual compartments [30, 5, 65]. The compilers they consider preserve the compartment abstraction using hardware features like capabilities, and the compiler property they prove—a variant of robust safety preservation called RSCC—states that if a compiled program linked to an untrusted target context incurs a sequence of safe and (compartment-specific) unsafe observable events, then the corresponding source program linked to *some* source context exhibits the same sequence.

Both our work and compartmentalizing compilation are frameworks for proving the security of programs compiled from unsafe languages. However, the two differ in fundamental ways. First, we work with vanilla unsafe languages and do not assume the existence of compartments in the source language. To define robust safety in the source, we introduce a different abstraction—the semantic source context, SIM—which can simulate the behavior of all target-language contexts. Second, compartmentalizing compilation uses *syntactic* back-translation for proofs, which works only if source contexts can express all target-language behavior in syntax, while we use *semantic* back-translation, which does not have this limitation as long as the source semantic domain can express all target-language behavior (our Rec–**Cap** instance demonstrates this). Third, work on compartmentalizing compilation does not support pointer passing, while we do. Finally, we are not aware of any program logic that leverages RSCC for proving robust safety of source programs, whereas we provide the program logic Reckon to establish robust safety of compiled Rec programs.

Full abstraction and robust property preservation. A large body of work studies compiler security using full abstraction [1, 3, 48, 15, 62, 6, 47, 12, 46] or the preservation of a class of robust properties (*e.g.*, robust safety or robust hypersafety) [49, 50, 14, 4, 30, 5, 65, 8, 9] as compiler properties. With the exception of RSCC above, none of this work considers unsafe source languages or hardening compilers. We believe that this choice is largely due to limitations of the proof methods that have been used thus far, namely, strong types and syntactic back-translations. We avoid these limitations by moving to a semantic proof technique based on DimSum. An interesting line of future work is to extend RobustDimSum to establish a wider variety of compiler properties that have been considered in the literature.

Robust safety and universal contracts. Our program logic Reckon (§2.2) is inspired by prior program logics for proving robust safety [64, 53, 18, 25]. Unlike Reckon, these prior logics apply to safe languages, which precludes the need for a hardening compiler to obtain robust safety. On the other hand, prior logics build directly on Iris and support Iris features like higher-order ghost state and nested invariants, which Reckon does not support as DimSum lacks step-indexing.

Program logics establish robust safety against some form of universal contract that overapproximates untrusted code behavior. Our universal contract, **UNIV** (§6.2), is inspired by prior work on universal contracts for capability machines [11, 28, 55]. The difference is that prior work defined these contracts as logical relations, whereas we define them as a semantic module in DimSum.

Capability machines as a target for verified compilers. Since the introduction of the CHERI architecture [71], capability machines have been used extensively as targets of verified compilers in prior work. Capabilities have been used for enforcing fine-grained memory safety (as we have done here) [15, 14], implementing specific calling conventions including well-bracketed control-flow (see below), and implementing compartments as in RSCC.

Our work assumes an idealized calling convention for **Cap**. Numerous authors have shown how such a calling convention can be realized using hardware capabilities. Skorstengaard et al.

[55] show how CHERI's *local capabilities* [70] can be used to enforce well-bracketed control flow and local state encapsulation by clearing the call stack before and after function calls. While this particular strategy is expensive when the stack is large, it is feasible on embedded devices with limited memory space, and has thus been adapted to the CHERIoT switcher [7]. Various extensions of capabilities—linear capabilities [56], temporal capabilities [67], uninitialized capabilities [20] and directed capabilities [21]—have been proposed to support secure and efficient calling conventions on general-purpose architectures. Each design comes with formal proofs that the enabled calling convention enforces local state encapsulation and well-bracketed control flow. Notably, some work relies on so-called *overlay semantics*—where the machine operational semantics is augmented with an idealized and well-behaved call-stack—which is then proven to be fully abstract with respect to the machine semantics. The idealized semantics we use in this work is closely related to the overlay semantics presented in [21]. An interesting line of future work is to implement the idealized stack of **Cap** using one of these techniques, and prove it correct within RobustDimSum.

Real-world compilers to capability machines. Existing compilers from C to CHERI implement varying degrees of safety. CHERI C/C++ [69] is a dialect of C which compiles all pointers to capabilities with tight bounds. However, as far as we know, the CHERI C/C++ compiler does not clear non-argument registers, nor does it implement a secure call-stack. As such it is not a hardening compiler in the way that Rec2Cap is. Rec2Cap resembles the Clang compiler implemented on top of CherIoT [7] more closely. CherIoT is an adaptation of CHERI for embedded systems, whose goal is to provide "full inter-compartment memory safety" [7] for embedded systems. It does so by implementing a secure calling convention between compartments called the switcher (which can be used to implement compilers that target CherIoT). The switcher adheres to the principle of least privilege (a core design principle of CherIoT) by clearing non-argument registers, and by clearing the call-stack upon function calls and returns. While verifying a full Clang compiler is still out of reach, we believe that adding a more elaborate backend targeting the CherIoT switcher to a compiler like Rec2Cap could be an interesting, real-world application of the ideas presented in this paper.

Multi-language semantics. Our work builds on DimSum [54]. Our additions to DimSum are the encodings of **UNIV** and SIM, the semantic back-translation proof, and the hardening Rec-to-**Cap** compiler Rec2Cap. While there is a large body of work on multi-language semantics, it is unclear whether one could base RobustDimSum on any of the other approaches. Work by Hur and others [10, 26, 27, 45] requires all target language programs to be representable as syntactic source programs, which rules out our Rec–**Cap** setup. Work on syntactic multi-languages [41, 52, 51, 40] focuses on safe, typed languages, and also cannot be applied to Rec and **Cap**, which are unsafe and untyped. Specifically, it seems infeasible to represent SIM syntactically. Compositional variants of the CompCert compiler [61, 24, 57, 34] require the source, target and intermediate languages to share the same memory model, but the memory models of Rec and **Cap** differ significantly.

Data Availability Statement

The Rocq development and appendix can be found in the supplementary material [43].

References

[1] Martín Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming (LNCS, Vol. 1603)*. Springer, 19–34. doi:10.1007/3-540-48749-2_2

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. A Theory of Secure Control Flow. In *ICFEM (LNCS, Vol. 3785)*. Springer, 111–124. doi:10.1007/11576280_9

[3] Martín Abadi, Cédric Fournet, and Georges Gonthier. 2000. Authentication Primitives and Their Compilation. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, Mark N. Wegman and Thomas W. Reps (Eds.). ACM, 302–315. doi:10.1145/325694.325734

[4] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hrițcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *CSF*. IEEE, 256–271. doi:10.1109/CSF.2019.00025

[5] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Cătălin Hrițcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *CCS*. ACM, 1351–1368. doi:10.1145/3243734.3243745

[6] Amal Ahmed and Matthias Blume. 2011. An equivalence-preserving CPS translation via multi-language semantics. In *ICFP*. ACM, 431–444. doi:10.1145/2034773.2034830

[7] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert M. Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *MICRO*. ACM, 641–653. doi:10.1145/3613424.3614266

[8] Cezar-Constantin Andrici, Ştefan Ciobâcă, Cătălin Hrițcu, Guido Martínez, Exequiel Rivas, Éric Tanter, and Théo Winterhalter. 2024. Securing Verified IO Programs Against Unverified Code in F*. *Proc. ACM Program. Lang.* 8, POPL (2024), 2226–2259. doi:10.1145/3632916

[9] Cezar-Constantin Andrici, Danel Ahman, Cătălin Hrițcu, Ruxandra Icleanu, Guido Martínez, Exequiel Rivas, and Théo Winterhalter. 2025. SecRef*: Securely Sharing Mutable References Between Verified and Unverified Code in F*. *Proc. ACM Program. Lang.* 9, ICFP (2025). To appear.

[10] Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *ICFP*. ACM, 97–108. doi:10.1145/1596550.1596567

[11] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *EuroS&P*. IEEE, 147–162. doi:10.1109/EUROSP.2016.22

[12] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017. Modular, Fully-abstract Compilation by Approximate Back-translation. *Log. Methods Comput. Sci.* 13, 4 (2017). doi:10.23638/LMCS-13(4:2)2017

[13] Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.* 22, 4&5 (2012), 477–528.

[14] Akram El-Korashy, Roberto Blanco, Jérémy Thibault, Adrien Durier, Deepak Garg, and Cătălin Hrițcu. 2022. SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation. In *CSF*. IEEE, 64–79. doi:10.1109/CSF54842.2022.9919680

[15] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2021. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. In *CSF*. IEEE, 1–16. doi:10.1109/CSF51468.2021.00036

[16] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2007. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 25. doi:10.1145/1275497.1275500

[17] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. doi:10.1145/3498689

[18] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71, 1 (2024), 3:1–3:59. doi:10.1145/3623510

[19] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71, 1 (2024), 3:1–3:59. doi:10.1145/3623510

[20] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. doi:10.1145/3434287

[21] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30.

[22] Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *J. Comput. Secur.* 11, 4 (2003), 451–520. doi:10.3233/JCS-2003-11402

[23] Orna Grumberg and David E. Long. 1994. Model Checking and Modular Verification. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 843–871. doi:10.1145/177492.177725

[24] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. ACM, 595–608. doi:10.1145/2676726.2676975

[25] Angus Hammond, Ricardo Almeida, Thomas Bauereiss, Brian Campbell, Ian Stark, and Peter Sewell. 2025. Morello-Cerise: A Proof of Strong Encapsulation for the Arm Morello Capability Hardware Architecture. *Proc. ACM Program. Lang.* 9, PLDI, Article 226 (June 2025), 23 pages. doi:10.1145/3729329

[26] Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. In *POPL*. ACM, 133–146. doi:10.1145/1926385.1926402

[27] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *POPL*. ACM, 59–72. doi:10.1145/2103656.2103666

[28] Sander Huyghebaert, Steven Keuchel, Coen De Roover, and Dominique Devriese. 2023. Formalizing, Verifying and Applying ISA Security Guarantees as Universal Contracts. In *CCS*. ACM, 2083–2097. doi:10.1145/3576915.3616602

[29] James H. Morris Jr. 1973. Protection in Programming Languages. *Commun. ACM* 16, 1 (1973), 15–21. doi:10.1145/361932.361937

[30] Yannis Juglaret, Cătălin Hrițcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *CSF*. IEEE Computer Society, 45–60. doi:10.1109/CSF.2016.11

[31] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. doi:10.1145/2951913.2951943

[32] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[33] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[34] Jérémie Koenig and Zhong Shao. 2021. CompCertO: compiling certified open C components. In *PLDI*. ACM, 1095–1109. doi:10.1145/3453483.3454097

[35] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[36] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 147–163.

[37] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *PLOS@SOSP*. ACM, 51–57. doi:10.1145/3144555.3144562

[38] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2*. Technical Report RR-7987. Inria.

[39] Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR* 41, 1 (2008), 1–31. doi:10.1007/s10817-008-9099-0

[40] Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *PPDP*. ACM, 16:1–16:15. doi:10.1145/3354166.3354181

[41] Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *POPL*. ACM, 3–10. doi:10.1145/1190216.1190220

[42] Mark S. Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-Based Financial Instruments. In *Financial Cryptography (LNCS, Vol. 1962)*. Springer, 349–378. doi:10.1007/3-540-45472-1_24

[43] Niklas Mück, Aïna Linn Georges, Derek Dreyer, Deepak Garg, and Michael Sammler. 2025. Artifact of "Endangered by the Language But Saved by the Compiler: Robust Safety via Semantic Back-Translation". Zenodo. doi:10.5281/zenodo.17285727 Also available on GitLab: https://gitlab.mpi-sws.org/FP/robustdimsum/-/tree/POPL26.

[44] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. doi:10.1145/1542476.1542504

[45] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *ICFP*. ACM, 166–178. doi:10.1145/2784731.2784764

[46] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 103–116. doi:10.1145/2951913.2951941

[47] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 6:1–6:50. doi:10.1145/2699503

[48] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6 (2019), 125:1–125:36. doi:10.1145/3280984

[49] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperty Preservation. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017.* IEEE Computer Society, 392–404. doi:10.1109/CSF.2017.13

[50] Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 1:1–1:41. doi:10.1145/3436809

[51] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: reasonably mixing a functional language with assembly. In *PLDI.* ACM, 495–509. doi:10.1145/3062341.3062347

[52] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP (LNCS, Vol. 8410).* Springer, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

[53] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.* 4, POPL (2020), 32:1–32:32. doi:10.1145/3371100

[54] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 775–805. doi:10.1145/3571220

[55] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2020. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst.* 42, 1 (2020), 5:1–5:53. doi:10.1145/3363519

[56] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2021. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *J. Funct. Program.* 31 (2021), e9. doi:10.1017/S095679682100006X

[57] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 23:1–23:31. doi:10.1145/3371091

[58] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. In *POPL.* ACM. https://doi.org/10.1145/3571232

[59] Simon Spies. 2025. *Shaking up the foundations of modern separation logic.* Ph. D. Dissertation. Saarland University, Germany. doi:10.22028/D291-46080

[60] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI.* ACM, 80–95. doi:10.1145/3453483.3454031

[61] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL.* ACM, 275–287. doi:10.1145/2676726.2676985

[62] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2021. Linear capabilities for fully abstract compilation of separation-logic-verified code. *J. Funct. Program.* 31 (2021), e6. doi:10.1017/S0956796821000022

[63] Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP (LNCS, Vol. 8410).* 149–168. doi:10.1007/978-3-642-54833-8_9

[64] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. doi:10.1145/3133913

[65] Jérémy Thibault, Roberto Blanco, Dongjae Lee, Sven Argo, Arthur Azevedo de Amorim, Aïna Linn Georges, Cătălin Hrițcu, and Andrew Tolmach. 2024. SECOMP: Formally Secure Compilation of Compartmentalized C Programs. In *CCS.* ACM, 1061–1075. doi:10.1145/3658644.3670288

[66] Amin Timany, Armaël Guéneau, and Lars Birkedal. 2024. The Logical Essence of Well-Bracketed Control Flow. *Proc. ACM Program. Lang.* 8, POPL (2024), 575–603. doi:10.1145/3632862

[67] Stelios Tsampas, Dominique Devriese, and Frank Piessens. 2019. Temporal Safety for Stack Allocated Memory on Capability Machines. In *CSF.* IEEE, 243–255. doi:10.1109/CSF.2019.00024

[68] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, Andrew P. Black and Barbara Liskov (Eds.). ACM, 203–216. doi:10.1145/168619.168635

[69] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. 2020. *CHERI C/C++ Programming Guide.* Technical Report UCAM-CL-TR-947. University of Cambridge, Computer Laboratory. doi:10.48456/tr-947

[70] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 20–37. doi:10.1109/SP.2015.9

[71] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA*. IEEE Computer Society, 457–468. doi:10.1109/ISCA.2014.6853201

[72] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. doi:10.1145/3371119

[73] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *SP*. IEEE Computer Society, 79–93. doi:10.1109/SP.2009.25

Received 2025-07-10; accepted 2025-11-06