A Semantic Approach to Robust Property Preservation

Niklas Mück
MPI-SWS
mueck@mpi-sws.org

Michael Sammler
ETH Zurich
michael.sammler@inf.ethz.ch

Aïna Linn Georges
MPI-SWS
algeorges@mpi-sws.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Deepak Garg
MPI-SWS
dg@mpi-sws.org

Robust property preservation. When reasoning about partial-program security, a useful concept is that of robustness [Gordon and Jeffrey 2003; Grumberg and Long 1994]. A program module is said to possess a (hyper)property *robustly* if the (hyper)property holds when the module runs linked with an *arbitrary context*, which may model untrusted third-party libraries. Letting C, P, π, $\cup$ and $\mathcal{B}$ denote contexts, program modules, (hyper)properties, the language's module linking operator, and the mapping from whole programs to behaviors, respectively, we say that P has π robustly if $\mathcal{B}(C \cup P) \in \pi$.

In prior work, compiler security has been characterized as the *preservation* of a class of robust properties by the compiler's transformation [Abadi 1999; Abate et al. 2021, 2019, 2018; Devriese et al. 2018; New et al. 2016; Patrignani et al. 2015, 2019, 2016]. For example, if $\diamond$ denotes all safety properties, the subscripts $\mathtt{S}$ and $\mathtt{T}$ denote the source and target languages, and $\lceil\cdot\rceil_{\mathtt{S}\rightleftharpoons\mathtt{T}}$ is a mapping of source behaviors to corresponding target behaviors, then the compiler $\downarrow(\cdot)$ satisfies the security criterion *robust safety preservation* or RSP if

$$\forall \pi \in \diamond, \mathtt{P_S}.\ (\forall \mathtt{C_S}.\ \mathcal{B}(\mathtt{C_S} \cup_{\mathtt{S}} \mathtt{P_S}) \in \pi) \implies (\forall \mathtt{C_T}.\ \mathcal{B}(\mathtt{C_T} \cup_{\mathtt{T}} \downarrow \mathtt{P_S}) \in \lceil \pi \rceil_{\mathtt{S}\rightleftharpoons\mathtt{T}}) \tag{1}$$

In words, for any safety property π, if the source module $\mathtt{P_S}$ satisfies $\lceil\pi\rceil_{\mathtt{S}\rightleftharpoons\mathtt{T}}$ robustly, then the compiled module $\downarrow\mathtt{P_S}$ satisfies π robustly. Stated differently, a compiler satisfying RSP *transfers* any robust safety property of the source program to the compiled output program. Indeed, a typical setup would be to first prove a desired robust safety property of a source program using, say, a program logic, and to then compile the program with a RSP-satisfying compiler. [1]

Challenges. Applying RSP to a realistic compiler runs into two fundamental challenges that we seek to address.

(C1) If the source language has undefined behavior (UB), then a source program module can robustly possess a property π only if π includes UB (since the module's context can exhibit UB). However, most safety properties of interest exclude UB, so we must either weaken the RSP definition to account for source UB [Abate et al. 2018], or restrict source contexts to those that do not exhibit UB. Both approaches lead to robustness notions that are difficult to interpret.

(C2) A proof of RSP must show that if a target-language context $\mathtt{C_T}$ causes a compiled module $\downarrow\mathtt{P_S}$ to violate π, then some source context $\mathtt{C_S}$ causes $\mathtt{P_S}$ to violate π. This amounts to a simulation of target contexts in the source language – a proof step called *backtranslation* [Devriese et al. 2017; El-Korashy et al. 2022, 2021]. Backtranslation is tedious, especially when source modules have private state that they can share with the context dynamically as, in that case, the simulating source context must maintain a lookup data structure with every memory location that was shared directly or indirectly with the target context in the past. Doing this bookkeeping using source *syntax* is an exercise in tedium – a recent mechanized proof of backtranslation between two idealized languages with dynamic memory sharing spans 29k lines of Rocq proof, which is a 10x increase over an earlier setting that excluded dynamic memory sharing [El-Korashy et al. 2022].

Our proposal. Our insight is that both these problems can be solved by moving to denotational semantics and *defining RSP denotationally*. This approach is best suited to settings where robust safety properties of source programs are established using a program logic, $\mathcal{L}_{\mathcal{S}}$, and we wish to transfer these properties to compiled target programs.

Our design consists of three steps. 1) Pick suitable source and target denotation domains, $\mathcal{D}_{\mathtt{S}}$ and $\mathcal{D}_{\mathtt{T}}$; define denotational semantics of the source and target program modules, $[\![\mathtt{P_S}]\!]_{\mathtt{S}}$ and $[\![\mathtt{P_T}]\!]_{\mathtt{T}}$; define linking operators, $\oplus_{\mathtt{S}}$ and $\oplus_{\mathtt{T}}$ in the two domains. 2) Pick an object $\mathtt{UNIV} \in \mathcal{D}_{\mathtt{T}}$ that is *universal*, i.e., $\mathtt{UNIV}$ over-approximates the denotation of any target-language program. 3) Pick an object $\mathtt{SIM} \in \mathcal{D}_{\mathtt{S}}$ that satisfies two conditions: (a) $\mathtt{UNIV}$ can be simulated by $\mathtt{SIM}$ up to $\lceil\cdot\rceil_{\mathtt{S}\rightleftharpoons\mathtt{T}}$, i.e., the source semantic object $\mathtt{SIM}$ can simulate all *target* language behavior, and (b) $\mathtt{SIM}$ keeps $\mathcal{L}_{\mathcal{S}}$ sound in the following sense: If $\mathcal{L}_{\mathcal{S}}$ proves that $\mathtt{P_S}$ has property π robustly, then $([\![\mathtt{P_S}]\!]_{\mathtt{S}} \oplus_{\mathtt{S}} \mathtt{SIM}) \in \pi$. $\mathtt{SIM}$ does not have to over- or under-approximate source contexts and, in particular, it does not have to include UB.

We say that the compiler $\downarrow\cdot$ has *semantic robust safety preservation* or semantic RSP if

$$\forall \pi \in \diamond, \mathtt{P_S}.\ ([\![\mathtt{P_S}]\!]_{\mathtt{S}} \oplus_{\mathtt{S}} \mathtt{SIM}) \in \pi \implies ([\![\downarrow \mathtt{P_S}]\!]_{\mathtt{T}} \oplus_{\mathtt{T}} \mathtt{UNIV}) \in \lceil \pi \rceil_{\mathtt{S}\rightleftharpoons\mathtt{T}} \tag{2}$$

or, equivalently,

$$\forall \mathtt{P_S}.\ [\![\downarrow \mathtt{P_S}]\!]_{\mathtt{T}} \oplus_{\mathtt{T}} \mathtt{UNIV} \preceq \left\lceil [\![\mathtt{P_S}]\!]_{\mathtt{S}} \oplus_{\mathtt{S}} \mathtt{SIM} \right\rceil_{\mathtt{S}\rightleftharpoons\mathtt{T}} \tag{3}$$

where $\preceq$ denote *refinement* of finite behavior in $\mathcal{D}_{\mathtt{T}}$.

[1] We focus on RSP here, but we believe that our ideas will extend to other robustness based compiler security criteria, e.g., those based on hyperproperties.

Semantic RSP does not imply standard RSP (Eq. 1) because SIM is not an underapproximation of source contexts. However, semantic RSP is a suitable replacement because, like standard RSP, it transfers $\mathcal{L}_S$-provable robust source properties to the denotations of compiled programs (this is due to the conditions on SIM).

Semantic RSP addresses the two challenges of standard RSP by allowing us to pick a SIM that does not have UB, and by eliminating the bureaucracy of syntax. We explain later how we pick a suitable SIM for our concrete instantiation. Remarkably, our proof of semantic RSP factors into purely semantic reasoning for backtranslation, and a standard *compiler correctness property*, which is the only place where we have to reason with syntax.

Instantiation. We are working on an instantiation of semantic RSP in DimSum [Sammler et al. 2023], a foundational, denotational framework for reasoning modularly about multi-language programs. DimSum represents syntactic modules' behavior as *semantic modules* – labeled transition systems (LTSs) whose transition labels encode interactions with other modules (such as external calls and returns) as well as side conditions. The side conditions are assertions in a separation logic over all modules' private and shared state. For a transition to occur, its side conditions must hold.

DimSum provides generic libraries to define and reason with different semantic linking operators (denoted $\oplus_S, \oplus_T$), wrappers that translate between traces of different LTSs (denoted $\lceil\cdot\rceil_{S\rightleftharpoons T}$), and trace refinement (written $\preceq$). Refinement is congruent w.r.t. $\oplus$ and $\lceil\cdot\rceil_{S\rightleftharpoons T}$, and wrappers are homomorphic w.r.t. $\oplus$.

$$M_L \preceq M_L' \implies M_L \oplus_L M_L'' \preceq M_L' \oplus_L M_L'' \tag{4}$$

$$\lceil M_S \rceil_{S\rightleftharpoons T} \oplus_T \lceil M_S' \rceil_{S\rightleftharpoons T} \preceq \lceil M_S \oplus_S M_S' \rceil_{S\rightleftharpoons T} \tag{5}$$

Our source language, Rec, has modules modeled after C's compilation units, recursive functions, stack-local and module-local (static) variables, and pointers that can be passed at runtime to share local variables with other modules. Our target language, Cap, is an idealized assembly language with capabilities inspired by Cerise [Georges et al. 2024] and CHERI [Woodruff et al. 2014], and an in-built stack to protect return addresses (such a stack can be realized using capabilities [Georges et al. 2022]).

Our Rec-to-Cap compiler builds on a compiler in DimSum. The key additions are module-local variables, capabilities and the new semantic RSP property.

Proof strategy. Figure 1 shows an outline of our proof of semantic RSP in DimSum. The subscripts r and c represent the source and target languages, Rec and Cap, respectively.

Step (I) follows immediately from the congruence property (4) and standard compiler correctness, which is formalized in DimSum as $\forall P.\ [\![\downarrow P]\!]_c \preceq \lceil [\![P]\!]_r \rceil_{r\rightleftharpoons c}$. This is the only step in our proof that reasons with program syntax.

$$\begin{aligned}
&[\![\downarrow P]\!]_c \oplus_c \mathbf{UNIV} \\
\preceq\ &\lceil [\![P]\!]_r \rceil_{r\rightleftharpoons c} \oplus_c \mathbf{UNIV} && \text{Compiler correctness} && \text{(I)} \\
\preceq\ &\lceil [\![P]\!]_r \rceil_{r\rightleftharpoons c} \oplus_c \lceil \mathrm{SIM} \rceil_{r\rightleftharpoons c} && \text{Semantic back-translation} && \text{(II)} \\
\preceq\ &\lceil [\![P]\!]_r \oplus_r \mathrm{SIM} \rceil_{r\rightleftharpoons c} && \text{Compositionality} && \text{(III)}
\end{aligned}$$

Figure 1. Proof Outline

Step (II) follows from condition (a) on the choice of SIM, formally $\mathbf{UNIV} \preceq \lceil \mathrm{SIM} \rceil_{r\rightleftharpoons c}$, and property (4). This is the key "backtranslation" step. Here, its statement and proof are both semantic.

Step (III) is an instance of property (5).

UNIV and SIM as DimSum modules. The availability of separation logic in DimSum lets us pick suitable UNIV and SIM easily. Following prior work [Devriese et al. 2016, 2018; Georges et al. 2024; Huyghebaert et al. 2023; Strydonck et al. 2022; Swasey et al. 2017], we define UNIV as a *universal contract* for the target language in separation logic.

For the source language, we start with the program (separation) logic $\mathcal{L}_S$ that is used to establish robust safety properties of source programs. As in prior work [Swasey et al. 2017], such a logic features a separation logic predicate on locations, $\mathrm{low}(\ell)$, which semantically means that the module being verified has no invariants associated with the translated target location ℓ and with any heap location reachable from ℓ. Effectively, $\mathrm{low}(\ell)$ models locations that have been shared with the context. The logic features the following rule for calling into the context adv with a pointer argument ℓ.

$$\{\mathrm{low}(\ell)\}\ \mathrm{adv}(\ell)\ \{\mathrm{low}(\ell)\} \qquad \text{(Hoare-Adv)}$$

Given this setup, we define SIM as a LTS with two labels: An incoming label that models control transfer *into* the context and an outgoing label that models control transfer *from* the context. The incoming label's side condition states that any locations being shared with the context are low.

This definition of SIM precludes UB, and it is trivially compatible with $\mathcal{L}_S$ (in particular, with Hoare-Adv). Further, the proof of $\mathbf{UNIV} \preceq \lceil \mathrm{SIM} \rceil_{r\rightleftharpoons c}$ simplifies to the following: Let C be a triggered incoming transition in SIM, $\lceil C \rceil_{r\rightleftharpoons c}$ be a corresponding incoming transition in UNIV, and D' be an immediately following outgoing transition in UNIV. Then, there exists a SIM outgoing transition D such $\lceil D \rceil_{r\rightleftharpoons c} = D'$. To complete this proof, we must show that state updates made by UNIV prior to D' can be simulated by SIM. This step is easy because all state private to or shared with SIM is low from the linked module's perspective, i.e., it is under SIM's control. Hence, we can update the state nondeterministically to match D' without violating any constraints.

The same proof done *syntactically* would additionally require the construction of syntactic source statements to perform those updates, which would be extremely tedious.

References

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

Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-relating Compiler Correctness and Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 14:1–14:48. https://doi.org/10.1145/3460860

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

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

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 147–162.

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2018. Parametricity versus the universal type. *Proc. ACM Program. Lang.* 2, POPL (2018), 38:1–38:23. https://doi.org/10.1145/3158126

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). https://doi.org/10.23638/LMCS-13(4:2)2017

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

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. https://doi.org/10.1109/CSF51468.2021.00036

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. https://doi.org/10.1145/3623510

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. https://doi.org/10.1145/3527318

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

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

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. https://doi.org/10.1145/3576915.3616602

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. https://doi.org/10.1145/2951913.2951941

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. https://doi.org/10.1145/2699503

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. https://doi.org/10.1145/3280984

Marco Patrignani, Dominique Devriese, and Frank Piessens. 2016. On Modular and Fully-Abstract Compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 17–30. https://doi.org/10.1109/CSF.2016.9

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. https://doi.org/10.1145/3571220

Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. 2022. Proving full-system security properties under multiple attacker models on capability machines. In *CSF*. IEEE, 80–95. https://doi.org/10.1109/CSF54842.2022.9919645

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. https://doi.org/10.1145/3133913

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. https://doi.org/10.1109/ISCA.2014.6853201