Composing Secure Compilers

Matthis Kruse
CISPA Helmholtz Center for Information Security
Germany
matthis.kruse@cispa.de

Marco Patrignani
CISPA Helmholtz Center for Information Security
Germany
marco.patrignani@cispa.de

1 Introduction

Compilers translate programs from a source to a target programming language. A secure compiler preserves source level properties at the target level when interoperating with arbitrary program contexts (which are considered attackers). A recent theory of secure compilation is Robust Compilation (RC), which is a collection of criteria for secure compilers [1, 2, 13]. Informally, a compiler is RC if a source program and its compiled counterpart, linked with an arbitrary source and target context respectively, satisfy that property.

Even though there exist robust compilers, they are far from practical. Real-world compilers consist of several smaller compilers that are composed with each other in different ways. An example would be any compiler based on the LLVM toolchain [11], whose optimisation pipeline consists of many passes, which one can view as independent compilers composed with each others. Also, any lowering steps, such as from a frontend language to LLVM IR and subsequently to assembly, are compilers. To the best of our knowledge, current work on robust compilation does not discuss the preservation of source-level properties for compilers such as the ones above.

This paper investigates how different compiler compositions preserve different classes of hyperproperties, given that these compilers attain some form of RC. We examine whether these compositions preserve at least the set intersection of classes. We then show that the order of optimisations in a RC pipeline does not matter for property preservation. Finally, we conclude with a discussion on what happens if some compilers in the pipeline do not attain RC for some classes of interest.

2 Compositionality

In this work, programs p are elements of $\mathcal{P}$, the set of partial programs of a given programming language. A compiler is a partial function $\llbracket \bullet \rrbracket^{S \to T}$ from programs p of some source language S to programs p of some target language T. Compilers satisfying Definition 2.1 below attain RC [2], the intuition there is that if the programmer makes certain assumptions on what a program does, these assumptions also hold for the compiled program. In that definition, indicate hyperproperties [7] with Π and classes of hyperproperties (i.e., sets of Π) as $\mathbb{C}$. A program p robustly satisfies class $\mathbb{C}$ (written $p \vDash_R \mathbb{C}$) if its behaviour is included in an element of $\mathbb{C}$ when linked with an arbitrary program context. Similarly, for some $\Pi \in \mathbb{C}$, we write $p \vDash_R \Pi$ whenever p robustly satisfies Π.

Definition 2.1 (Robust Compilation)**.** For a given class $\mathbb{C}$, a compiler from languages S to T robustly preserves $\mathbb{C}$ ($\vdash \llbracket \bullet \rrbracket^{S \to T} : \mathbb{C}$) iff

$$\forall \Pi \in \mathbb{C}, \forall p \in \mathcal{P}, p \vDash_R \Pi \implies \llbracket p \rrbracket^{S \to T} \vDash_R \Pi$$

In practice, (robust) compilers are composed of numerous others. Therefore, we now investigate their compositionality.

2.1 Simple Compositionality

We first consider function composition, i.e., plugging the result of one compiler into another one. Such pipelines happen when optimising source code (so, at the level of a suitable intermediate representation), but also on a higher level: Consider as an example a typical TypeScript compilation pipeline. First, the compiler translates TypeScript code to *JavaScript*, which a part of V8 eventually compiles the code just-in-time to **assembly**.

Definition 2.2 (Sequential Composition of Compilers)**.** Given two compilers $\llbracket \bullet \rrbracket^{S \to I}$ and $\llbracket \bullet \rrbracket^{I \to T}$, their sequential composition is $\llbracket \bullet \rrbracket^{S \to T} = \llbracket \llbracket \bullet \rrbracket^{S \to I} \rrbracket^{I \to T}$.

Assuming that two compilers preserve certain classes, their sequential composition preserves the least upper bound, i.e., the set intersection of those classes:

Lemma 2.3 (Sequential Composition with RC)**.** *Given* $\vdash \llbracket \bullet \rrbracket^{S \to I} : \mathbb{C}_1$ *and* $\vdash \llbracket \bullet \rrbracket^{I \to T} : \mathbb{C}_2$*, then* $\vdash \llbracket \bullet \rrbracket^{S \to I \to T} : \mathbb{C}_1 \cap \mathbb{C}_2$.

Using an inductive argument, Lemma 2.3 generalises to n RC compilers, each preserving one of n classes. To do so, one has to generalise the composition of two RC compilers to a set of n ones. A real-world example for such deeply nested compositions is the TypeScript compilation mentioned above. When compiling *JavaScript*, V8 translates the code to **Ignition Bytecode**. At runtime, the Ignition interpreter does some performance measurements and particular parts of the code are eventually compiled to machine code.

We now consider a compiler that invokes two other compilers. Java and *Kotlin* are popular languages used in industry that are one example of such a composition and they both compile to **JVM Bytecode**.

Definition 2.4 (Upper Composition)**.** Given two compilers $\llbracket \bullet \rrbracket^{S \to T}$ and $\llbracket \bullet \rrbracket^{I \to T}$, their upper composition is

$$\llbracket \bullet \rrbracket^{S + I \to T} = \lambda p. \begin{cases} \llbracket p \rrbracket^{S \to T} & \text{if } p \in \mathcal{P} \\ \llbracket p \rrbracket^{I \to T} & \text{if } p \in \mathcal{P} \end{cases}$$

We can derive a similar result to Lemma 2.3 here, too:

Lemma 2.5 (Upper Composition with RC). *Given* $\vdash \llbracket\bullet\rrbracket^{S\to T} : \mathbb{C}_1$ *and* $\vdash \llbracket\bullet\rrbracket^{I\to T} : \mathbb{C}_2$, *then* $\vdash \llbracket\bullet\rrbracket^{S+I\to T} : \mathbb{C}_1 \cap \mathbb{C}_2$.

Lemma 2.5 also generalises inductively to a number of compilers and classes. A practical example of why that might be useful is the Java Virtual Machine with its **JVM Bytecode**, which has numerous frontends: Java, *Kotlin*, Scala, and *Clojure*, to list a few examples.

With the same idea, we define a dual composition that goes from a single source language to multiple target languages. dune is a build system which can be used to compile OCaml code to both *assembly* and **Caml Bytecode**.

Definition 2.6 (Lower Composition). Given two compilers $\llbracket\bullet\rrbracket^{S\to T}$ and $\llbracket\bullet\rrbracket^{S\to I}$, their lower composition is $\llbracket\bullet\rrbracket^{S\to I+T}$.

Lemma 2.7 (Lower Composition with RC). *Given* $\vdash \llbracket\bullet\rrbracket^{S\to T} : \mathbb{C}_1$ *and* $\vdash \llbracket\bullet\rrbracket^{S\to I} : \mathbb{C}_2$, *then* $\vdash \llbracket\bullet\rrbracket^{S\to I+T} : \mathbb{C}_1 \cap \mathbb{C}_2$.

As before, this can be generalized to an arbitrary number of compilers, which also has a connection to the real-world, given by the diverse set of assembly language dialects.

The following free theorem (Lemma 2.8) is a direct consequence of Lemma 2.3 where the involved compilers' input and output are both partial programs in the same language. Given that some compiler passes attain RC, they can be combined in an arbitrary order and the result preserves the same least upper bound. A compiler's pipeline ordering is difficult and often hand-tuned. The lemma allows us to not care about the particular order of optimisations regarding their robust property preservation. So, the compiler developer is free to swap passes around.

Lemma 2.8 (Swappable). *Given* $\vdash \llbracket\bullet\rrbracket^{T\to T}_{(1)} : \mathbb{C}_1$ *and* $\vdash \llbracket\bullet\rrbracket^{T\to T}_{(2)} : \mathbb{C}_2$, *then* $\vdash \llbracket\llbracket\bullet\rrbracket^{T\to T}_{(2)}\rrbracket^{T\to T}_{(1)} : \mathbb{C}_1 \cap \mathbb{C}_2$ *and* $\vdash \llbracket\llbracket\bullet\rrbracket^{T\to T}_{(1)}\rrbracket^{T\to T}_{(2)} : \mathbb{C}_1 \cap \mathbb{C}_2$.

However, in practice, compiler passes are not necessarily attaining RC. Consider any stereotypical compilation pipeline. Programmers want properties at the source level to be preserved at the target level. Thus, if source programs robustly satisfy some property, so should their compiled counterparts. Unfortunately, it might not be necessary for compilation passes from one intermediate representation to the other to preserve properties robustly. This also has a security justification since compiler intermediate representations are not where typical attackers reside (i.e., the target language). So, there might be some stronger property a pass has to satisfy in order to render the whole compilation pipeline secure: this is what we study next.

2.2 Advanced Compositionality

Consider the following C code snippet that performs an infinite loop if an invalid pointer is given:

```
int something(int* ptr) {
  while(!ptr);
  return *ptr;
}
```

Compiling such code with optimisations turned on by using the command g++ -O2 and the g++ compiler version 11.2 yields an x86-program where the potentially infinite loop has been removed:

```
something(int*):
  mov eax, DWORD PTR [rdi]
  ret
```

We now have an attack to violate memory safety: call the function with an invalid pointer and the program dereferences it.

To prevent such issues we can use instrumentation passes that *enforce* memory safety by adding dynamic checks to the program and crashing appropriately when a violation is detected. There exist several memory-safety instrumentations, both for target [8, 15–19] and source languages [3, 12, 14].

We now sketch how to extend our work with instrumentations, which enforce specific classes of hyperproperties.

Definition 2.9 (Secure Instrumentation for Preserving $\mathbb{C}$). A secure instrumentation with respect to some class $\mathbb{C}$ is a pass that enforces hyperproperties described by some other class $\mathbb{C}'$ without violating $\mathbb{C}$-satisfying programs. We denote such a secure instrumentation as: $\llbracket\bullet\rrbracket^{S\to T} \succ_{\mathbb{C}} \mathbb{C}'$.

Using this, we firstly want to inspect a compilation pipeline from memory-safe Rust to optimised, insecure *C*, to memory-safe **CheckedC**. Intuitively, we want to be able to state that this pipeline preserves memory safety, despite the fact that the pass to *C* does not.

Example 2.10 (Enforcement may preserve...). Given classes $\mathbb{C}_1, \mathbb{C}_2$ (resp. no property and memory safety, in our Rust to **CheckedC** example) and compilers $\llbracket\bullet\rrbracket^{S\to I}$, $\llbracket\bullet\rrbracket^{I\to T}$, if:

- $\vdash \llbracket\bullet\rrbracket^{S\to I} : \mathbb{C}_1$
- $\llbracket\bullet\rrbracket^{I\to T} \succ_{\mathbb{C}_1} \mathbb{C}_2$

Then, $\vdash \llbracket\bullet\rrbracket^{S\to I\to T} : \mathbb{C}_1 \cup \mathbb{C}_2$.

Dually, running a compiler that does not respect memory-safety after a memory-safety instrumentation nullifies its preservation:

Example 2.11 (...but, order matters!). Given classes $\mathbb{C}_1, \mathbb{C}_2$ and compilers $\llbracket\bullet\rrbracket^{S\to I}$, $\llbracket\bullet\rrbracket^{I\to T}$, if:

- $\llbracket\bullet\rrbracket^{S\to I} \succ_{\mathbb{C}_1} \mathbb{C}_2$
- $\vdash \llbracket\bullet\rrbracket^{I\to T} : \mathbb{C}_1$

Then, $\vdash \llbracket\bullet\rrbracket^{S\to I\to T} : \mathbb{C}_1$.

Beyond this general theory, we also intend to study the compositionality aspects of concrete hyperproperties, such as Speculative Non-Interference [10], memory safety [4, 5, 9], and cryptographic constant-time [6].

References

[1] Carmine Abate, Roberto Blanco, Ștefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hrițcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 1–28.

[2] 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 *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 256–25615. https://doi.org/10.1109/CSF.2019.00025

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada) *(SSYM'09)*. USENIX Association, USA, 51–66.

[4] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Cătălin Hrițcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *2015 IEEE Symposium on Security and Privacy (2015 IEEE Symposium on Security and Privacy)*. San Jose, United States, 813 – 830. https://doi.org/10.1109/SP.2015.55

[5] Arthur Azevedo de Amorim, Cătălin HriŢcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 79–105.

[6] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *CSF 2018 - 31st IEEE Computer Security Foundations Symposium*. Oxford, United Kingdom. https://hal.archives-ouvertes.fr/hal-01959560

[7] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. IEEE Computer Society, 51–65. https://doi.org/10.1109/CSF.2008.7

[8] Vítor Bujés Ubatuba De Araújo, Álvaro Freitas Moreira, and Rodrigo Machado. 2016. Týr: A Dependent Type System for Spatial Memory Safety in LLVM. *Electronic Notes in Theoretical Computer Science* 324 (2016), 3–13. https://doi.org/10.1016/j.entcs.2016.09.003 WEIT 2015, the Third Workshop-School on Theoretical Computer Science.

[9] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and Andre DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 487–502. https://doi.org/10.1145/2786763.2694383

[10] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2019. SPECTECTOR: Principled Detection of Speculative Information Flows. arXiv:1812.08639 [cs.CR]

[11] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[12] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

[13] 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. https://doi.org/10.1145/3436809

[14] Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. 2017. Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (Prague, Czech Republic) *(ManLang 2017)*. Association for Computing Machinery, New York, NY, USA, 35–47. https://doi.org/10.1145/3132190.3132204

[15] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

[16] David Tarditi, Archibald Samuel Elliott, Andrew Ruef, and Michael Hicks. 2018. Checked C: Making C Safe by Extension. In *IEEE Cybersecurity Development Conference 2018 (SecDev)*. IEEE, 53–60. https://www.microsoft.com/en-us/research/publication/checkedc-making-c-safe-by-extension/

[17] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-Free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 405–419. https://doi.org/10.1145/3064176.3064211

[18] Marco Vassena and Marco Patrignani. 2019. Memory Safety Preservation for WebAssembly. arXiv:1910.09586 [cs.PL]

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