Composing Secure Compilers

Matthias Kruse
CISPA Helmholtz Center for Information Security
Germany
matthias.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 inter-operating 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 $\{\bullet\}_{S}^{S\rightarrow T}$ 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 $\Pi$ and classes of hyperproperties (i.e., sets of $\Pi$) as $\mathcal{C}$. A program $p$ robustly satisfies class $\mathcal{C}$ (written $p \models_{R} \mathcal{C}$) if its behaviour is included in an element of $\mathcal{C}$ when linked with an arbitrary program context. Similarly, for some $\Pi \in \mathcal{C}$, we write $p \models_{R} \Pi$ whenever $p$ robustly satisfies $\Pi$.

Definition 2.1 (Robust Compilation). For a given class $\mathcal{C}$, a compiler from languages $S$ to $T$ robustly preserves $\mathcal{C}$ ($\{\bullet\}_{S}^{S\rightarrow T}$ : $\mathcal{C}$) iff

$$\forall \Pi \in \mathcal{C}, \forall p \in \mathcal{P}, p \models_{R} \Pi \implies [p]_{S\rightarrow T} \models_{R} \Pi$$

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 $\{\bullet\}_{S}^{S\rightarrow T}$ and $\{\bullet\}_{T}^{T\rightarrow U}$, their sequential composition is $\{\bullet\}_{T}^{S\rightarrow U} = \{\bullet\}_{S}^{S\rightarrow T} \circ \{\bullet\}_{T}^{T\rightarrow U}$.

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

$$\{\bullet\}_{S}^{S\rightarrow T} \cap \{\bullet\}_{T}^{T\rightarrow U} = \{\bullet\}_{S}^{S\rightarrow T} \circ \{\bullet\}_{T}^{T\rightarrow U}$$

Lemma 2.3 (Sequential Composition with RC). Given $\{\bullet\}_{S}^{S\rightarrow T}$ : $\mathcal{C}_1$ and $\{\bullet\}_{T}^{T\rightarrow U}$ : $\mathcal{C}_2$, then $\{\bullet\}_{S}^{S\rightarrow U} : \mathcal{C}_1 \cap \mathcal{C}_2$.

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 $\{\bullet\}_{S}^{S\rightarrow T}$ and $\{\bullet\}_{T}^{T\rightarrow U}$, their upper composition is $\{\bullet\}_{S}^{S\rightarrow U} = \lambda p. \begin{cases} [p]_{S\rightarrow T} & \text{if } p \in \mathcal{P} \\ [\hat{\Pi}]_{S\rightarrow T} & \text{if } p \in \hat{\mathcal{P}} \end{cases}$
We can derive a similar result to Lemma 2.3 here, too:

**Lemma 2.5 (Upper Composition with RC).** Given \( \vdash [\bullet]^{S \rightarrow T} : C_1 \) and \( \vdash [\bullet]^{T \rightarrow I} : C_2 \), then \( \vdash [\bullet]^{S \rightarrow I} : C_1 \cap C_2 \).

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 \([\bullet]^{S \rightarrow T} \) and \([\bullet]^{T \rightarrow I} \), their lower composition is \([\bullet]^{S \rightarrow I} \).

**Lemma 2.7 (Lower Composition with RC).** Given \( \vdash [\bullet]^{S \rightarrow T} : C_1 \) and \( \vdash [\bullet]^{T \rightarrow I} : C_2 \), then \( \vdash [\bullet]^{S \rightarrow I} : C_1 \cap C_2 \).

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 \( \vdash [\bullet]^{T \rightarrow I} \) : \( C_1 \) and \( \vdash [\bullet]^{T \rightarrow I} : C_2 \), then \( \vdash [\bullet]^{T \rightarrow I} : C_1 \cap C_2 \).

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:

```c
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:

```c
    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 \( C \)).** A secure instrumentation with respect to some class \( C \) is a pass that enforces hyperproperties described by some other class \( C' \) without violating \( C \)-satisfying programs. We denote such a secure instrumentation as: \([\bullet]^{S \rightarrow I} \uparrow C C'\).

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 \( C_1, C_2 \) (resp. no property and memory safety, in our Rust to CheckedC example) and compilers \([\bullet]^{S \rightarrow T} \), \([\bullet]^{T \rightarrow I} \), if:

- \( \vdash [\bullet]^{S \rightarrow I} : C_1 \)
- \( \vdash [\bullet]^{T \rightarrow I} >_{C_1} C_2 \)

Then, \( \vdash [\bullet]^{S \rightarrow I} : C_1 \cup C_2 \).

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 \( C_1, C_2 \) and compilers \([\bullet]^{S \rightarrow I} \), \([\bullet]^{T \rightarrow I} \), if:

- \( \vdash [\bullet]^{T \rightarrow I} >_{C_1} C_2 \)
- \( \vdash [\bullet]^{S \rightarrow I} >_{C_2} C_1 \)

Then, \( \vdash [\bullet]^{S \rightarrow I} : C_1 \).

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].
Composing Secure Compilers

References


