SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation

Akram El-Korashy[1] Roberto Blanco[2] Jérémy Thibault[2] Adrien Durier[2] Deepak Garg[1] Cătălin Hriţcu[2]

[1]Max Planck Institute for Software Systems (MPI-SWS) [2]Max Planck Institute for Security and Privacy (MPI-SP)

***Abstract*—Proving secure compilation of partial programs typically requires back-translating a target attack against the compiled program to an attack against the source program. To prove this back-translation step, one can syntactically translate the target attacker to a source one—i.e., syntax-directed back-translation—or show that the interaction traces of the target attacker can also be produced by source attackers—i.e., trace-directed back-translation.**

Syntax-directed back-translation is not suitable when the target attacker uses unstructured control flow that the source language cannot directly represent. Trace-directed back-translation works with such syntactic dissimilarity because only the external interactions of the target attacker have to be mimicked in the source, not its internal control flow. Revealing only external interactions is, however, inconvenient when sharing memory via unforgeable pointers, since information about stashed pointers to shared memory gets lost. This made prior proofs complex, since the generated attacker had to stash all reachable pointers.

In this work, we introduce more informative data-flow traces, which allow us to combine the best of syntax-directed and trace-directed back-translation. Our data-flow back-translation is simple, handles both syntactic dissimilarity and memory sharing well, and we have proved it correct in Coq.

We, moreover, develop a novel turn-taking simulation relation and use it to prove a recomposition lemma, which is key to reusing compiler correctness in such secure compilation proofs. We are the first to mechanize such a recomposition lemma in a proof assistant in the presence of memory sharing.

We put these two key innovations to use in a secure compilation proof for a code generation compiler pass between a safe source language with pointers and components, and a target language with unstructured control flow.

1 Introduction

Compiler correctness or semantics preservation is the current gold standard for formally verified compilers [24, 27, 30, 41]. However, compiler correctness alone is insufficient for reasoning about security of compiled partial programs linked with arbitrary target contexts (e.g., components such as libraries) because compiler correctness shows that the compiled program simulates the source program only under the *assumption* that the target context obeys all restrictions of the *source language* semantics, i.e., it does not perform any low-level attacks that the source language would disallow. This assumption is usually false in practice: compiled programs are routinely linked with arbitrary, unverified target-language code that may be buggy, compromised, or outright malicious in contravention of source semantics. In these cases, compiler correctness (even in its "compositional" form [23, 35, 47, 49]), establishes no security guarantees for compiled partial programs.

This problem can be addressed by *secure compilation* [4, 40], wherein one shows that any violation of a security property of a compiled program in some target context also appears for the source program in some source context. Formally, this requires proving the existence of a property-violating source context given a target-level violation and the corresponding violating target context. This proof step, often called *back-translation*, is crucial for establishing that a vulnerable compiled program only arises from a vulnerable source program, thus preserving the security of partial source programs even against adversarial target contexts. Although there is a long line of work on proving secure compilation for prototype compilation chains that differ in the specific security properties preserved and the way security is practically enforced [3, 4, 6, 7, 13, 15, 16, 18, 36, 39, 40, 40, 46, 51, 53], back-translation is a common, large element of such secure compilation proofs.

Back-translation can be done in two different ways: *syntax-directed* and *trace-based*. Syntax-directed back-translation defines a function from the violating target context (a piece of syntax) to a source context. While this approach is easy to use in some situations [6, 7, 15, 16, 36, 40, 46, 51, 53], it has a significant limitation: it cannot be used if some constructs of the target language cannot be easily mimicked in the source language. For example, it is not well suited when the source language only has structured control flow, while the target language has unstructured control flow (goto or jump), as representing unstructured control flow in the source would require complex transformations or rely on heuristics that may not always work [32, 59]. Yet this kind of a difference between source and target languages is commonplace, e.g., when compiling any block-structured language to assembly.

In contrast, trace-based back-translation works by defining a target-language *interaction trace semantics* that represents all the interactions between the compiled program and its context (e.g., cross-component calls and returns) and constructing the violating source context from the violating target trace instead of the violating target context [3, 18, 37, 39]. This has the advantage of not having to mimic the internal behavior of the target context in the source language. So in contrast to the syntax-directed method, this trace-directed method works well even when some target language construct cannot be easily mimicked in the source language, as long as the construct's effect does not cross linking (program-context) boundaries.

Although very powerful in principle, trace-based back-translation is rather understudied for settings where the program and its context can *share memory* by passing pointers or references to each other, something that is common in practical languages like C, Java, and ML. There is a good reason for this relative paucity of work: memory sharing is a source of interesting interaction between the program and its context, so allowing it makes the definition of traces [18, 26], the back-translation, and the proof of secure compilation significantly more complex. Moreover, as we explain below, memory sharing changes parts of the proof conceptually and needs fundamentally new techniques. This is precisely the gap that this paper fills: it significantly advances secure compilation proofs *in the presence of memory sharing* by introducing two new proof techniques: (1) *data-flow back-translation*, which is a simpler form of back-translation, and (2) *turn-taking simulation*, which simplifies the remaining secure compilation proof. Next, we briefly describe why these new techniques are needed and what they do.

Data-flow back-translation To understand the need for this technique, consider a situation where source language has memory safety and prevents pointer forging (as in Java, Rust or ML). Suppose the compiled program has shared some pointer to its private memory with the co-linked target context in the past, and the context has stored this pointer somewhere in its private memory. At some later point in the execution, the context may use a *chain of memory dereferences within its private memory* to recover this pointer and write to the memory to which it points. Since this write changes shared memory, it must be recorded on the trace and must be mimicked by the source context constructed by back-translation. To mimic this write in the source language, the back-translated source context cannot just forge a pointer to the memory. Instead, it must follow a similar chain of dereferences (to the one used by the target context) in the source. However, the chain of memory dereferences leading to this pointer is in the target context's *private* memory and interaction traces omit these private dereferences by design!

Consequently, information needed to reconstruct *how* to access the shared pointer is missing from interaction traces, which makes the back-translation extremely difficult. Prior work that has even considered this situation [18, 38] relied on extensive bookkeeping to reconstruct this missing information, which is unwieldy and complex. For instance, the source context generated by El-Korashy et al. [18] had to fetch all reachable pointers every time it got control and store them in its internal state. This required complex simulation invariants, on top of the usual invariants between the states of the target and source contexts.

This is where our new idea of data-flow back-translation comes in. To perform the back-translation, we first enrich the standard interaction traces with information about data-flows *within* the context. This considerably simplifies the back-translation definition by providing precisely the missing chain of private memory dereferences in the trace itself. We see data-flow back-translation as a sweet spot between standard trace-based back-translation, which abstracts away all internal behavior of the context, and syntax-directed back-translation, which mimics the internal behavior of the context in detail.

Turn-taking simulations Our turn-taking simulations are useful when one tries to *reuse* compiler correctness as a lemma in the secure compilation proof to avoid duplicating large amounts of work. Specifically, after (trace-based) back-translation has been defined, one still has to prove that the source program and the back-translated source context actually reproduce the given (property-violating) trace of the compiled program and the target context. This is a difficult simulation proof over reduction steps, but many of the source and target steps are executed by the source program and its compilation and these two are already related by the statement of the compiler correctness theorem. Having to reprove the simulation for these steps would be tantamount to duplicating an involved proof [27]. This duplication can in fact be avoided by proving a simpler *recomposition* lemma in the target language [3]. Intuitively, recomposition says that if a program P_1 linked with a context C_1, and a program P_2 linked with a context C_2 both produce the same trace, then one may *recompose*—link P_1 with C_2—to obtain again the same trace.

The proof of recomposition is a ternary simulation between the runs of $P_1 \cup C_1$, $P_2 \cup C_2$, and the recomposed program $P_1 \cup C_2$. The question that becomes nuanced with memory sharing is how the memory of the recomposed program should be related to those of the given programs in this simulation. In the *absence of memory sharing*, this is straightforward: at any point in the simulation, the projection of P_1's memory in the recomposed run of $P_1 \cup C_2$ will equal the projection of P_1's memory from the run $P_1 \cup C_1$ (and dually for C_2's memory). However, with memory sharing, this simplistic relation does not work because C_2 may change parts of P_1's shared memory in ways that C_1 does not. Specifically, while control is not in P_1, the projections of P_1's memories in the two runs mentioned above will not match.

This is where our turn-taking simulations come in. They relate the memory of P_1 from the run of $P_1 \cup C_2$ to that from the run of $P_1 \cup C_1$ only while control is in P_1. When control shifts to the contexts (C_2 or C_1), this relation is limited to P_1's private memory (which is not shared with the context). The picture for C_2's memory is exactly dual. Overall, the relation takes "turns", alternating between two memory relations depending on where the control is. This non-trivial replacement for the juxtaposition-based relation allows us to prove recomposition and therefore reuse a standard compiler correctness proof even with memory sharing.

Concrete setting We illustrate our two new proof techniques by extending an existing mechanized secure compilation proof by Abate et al. [3] to cover dynamic memory sharing. The original proof was done for a compilation pass from an imperative source language with structured control flow (e.g., calls and returns, if-then-else) to an assembly-like target language with unstructured jumps. Both languages had components and pointers, and for the purpose of this compilation pass, pointers in both languages were assumed to be safe—i.e., out of bound accesses are errors that stop execution.[1] In both languages, the program and the context had their own private memories, and pointers to these memories could not be shared with other components. The program and the context interacted only by calling each other's functions, and passing only primitive values via call arguments and return values.

We extend both languages by allowing safe pointers to be passed to and dereferenced by other components, thus introducing dynamic memory sharing. We then prove again that this extended compilation step is secure with respect to a criterion called "robust safety preservation" [4, 5, 37]. For this, we apply our two new techniques, data-flow back-translation and turn-taking simulations. Since the parts of the proof using these new techniques are fairly involved and non-trivial, we also mechanize them in the Coq proof assistant.

Summary of contributions:

- We introduce data-flow back-translation and turn-taking simulations, two new techniques for scaling secure compilation proofs to both syntactic dissimilarity between the source and target languages and (dynamic) memory sharing.
- We apply these conceptual techniques to prove secure compilation for a code generation pass between a source language with structured control flow and a target language with unstructured control flow. In both languages, memory can be dynamically shared by passing safe pointers between components.
- We mechanize the parts of our proofs centered around these two key proof techniques in the Coq proof assistant.

Structure The rest of the paper is organized as follows: In §2 we motivate robust safety preservation and outline a previous proof [3] that did not support memory sharing. In §3 we explain the challenges of memory sharing, introduce data-flow back-translation and turn-taking simulation, and show how they fit into the existing proof outline. In §4 we show the source and target languages to which we apply these techniques. §5 provides additional details of applying data-flow back-translation to our setting (and §B in the appendix does the same for turn-taking simulation). Finally, we discuss related (§6) and future (§7) work.

Mechanized proofs in Coq The Coq proofs of back-translation and recomposition for the compilation pass outlined above are available at
`https://github.com/secure-compilation/when-good-components-go-bad/tree/memory-sharing`
The size of these two proof steps is 2.4k lines of specifications and 23k lines of proof. For comparison, in the Coq development without memory sharing on which we are building, these two proof steps were 2.7k lines in total [3], so an order of magnitude smaller. We believe that a significant part of this increase in proof size can be attributed to the increase in the conceptual difficulty of the two proofs in the presence of shared memory.

[1]While this is orthogonal to our current work on proof techniques, such safe pointers can be efficiently implemented using, for instance, hardware capabilities [55, 58] or programmable tagged architectures [14, 17].

Our mechanized proofs currently assume not only standard axioms (excluded middle, functional extensionality, etc.) but also some low-level specifications about the data structure we use for memory maps, as well as about allocation, reachability, and well-formedness of trace events (these are all documented in the included `README.md`). We believe that with a bit of extra effort these low-level specifications used transitively in our proofs can be proved as well. Even in the current state though, our proofs are done in much greater detail than previous paper proofs of secure compilation with memory sharing [18, 37], which gives us much higher confidence. We found that the use of a proof assistant was vital in getting the invariants right and checking the thousands of lines of proof of all the relevant simulation lemmas, some of which appear later in this paper.

2 Background

We start by giving some background: a motivating example explaining the broad setting we work in, the formal secure compilation property we prove (called robust safety preservation) and a proof strategy from prior work on which we build.

2.1 Motivating Example and Setting

We are interested in the common scenario where a *part* of a program is written in a *safe source language*, compiled to a target language and then linked against other target-language program parts, possibly untrusted or prone to be compromised, to finally obtain an executable target program. By "program part", we mean a collection of components (modules), each of which contains a set of functions. These functions may call other functions, both within this part and those in other parts. We use the terms "program" and "program part" to refer to the program part we wrote and compiled, and "context" to refer to the remaining, co-linked program part that we didn't write.

As an example, consider the following source program part, a single component, which implements a server-side function set_ads_image that prepares a page to be shown to the end-user. The function calls a helper function populate_partner_ads which is implemented by a third-party library from an advertising company.

```
extern int populate_partner_ads(char* img);
static char ads_image[65536];
static long long int user_balance_usd;

void set_ads_image () {
  populate_partner_ads(ads_image);
}
```

Suppose that the source language is memory safe and that the program part above is compiled using a *correct* compiler to some lower-level language, then linked to a context that implements populate_partner_ads, and the resulting program is executed. Our goal is to ensure a safety property **nowrite**—that populate_partner_ads never modifies the variable user_balance_usd (which is high integrity). Note that it is okay for populate_partner_ads to modify the array ads_image, which is passed as a parameter. The concern really is that a low-enough implementation of populate_partner_ads may overflow the array ads_image to overwrite user_balance_usd.

The invariant **nowrite** can be attained in at least two different ways, which we call **Setting 1** and **Setting 2**. In **Setting 1**, we allow the compilation of the program part above to be linked *only* to target-language contexts that were obtained by compiling program parts written in the same source language. Since the source language is safe, there is no way for any source function to cause a buffer overflow and a correct compiler will transfer this restriction to the target language so, in particular, the compilation of populate_partner_ads cannot overwrite user_balance_usd, thus ensuring **nowrite**. This kind of restriction on linking—and the verification of compilers under such restrictions—has been studied extensively in so-called *compositional compiler correctness* [23, 35, 47, 49]. Although useful, this offers no guarantees when the context containing populate_partner_ads is arbitrary target code.

This restriction on linking is lifted in **Setting 2**, where the context is arbitrary target code (written directly in the target language, or maybe compiled from another less safe language). Now, **nowrite** does not follow from source memory safety and the correctness of the compiler. Instead, we must rely on the compilation chain satisfying a secure compilation criterion that defends against malicious target-level contexts. It is this second setting that interests us here and, more broadly, a large part of the literature on secure compilation.

2.2 Robust Safety Preservation (*RSP*$^{\sim}$)

The next question is what security criterion the compilation chain must satisfy to ensure that **nowrite** or, more generally, any property of interest, holds in **Setting 2**. The literature on secure compilation has proposed many such criteria (see Abate et al. [4], Patrignani et al. [40]). Here we describe and adopt one of the simplest criteria that ensures **nowrite**, namely, *robust safety preservation* or ***RSP***$^{\sim}$ [5]:

Definition 2.1 (Compilation chain has ***RSP***$^{\sim}$ [5])**.**

$$\begin{aligned} \mathsf{RSP}^{\sim} \overset{\text{def}}{=}\ & \forall \mathsf{P}^{2}\ \mathbf{C_t}\ t.\\ & (\mathbf{C_t} \cup \mathsf{P}\!\downarrow) \rightsquigarrow^{*} t \implies \\ & \exists \mathsf{C_s}\ t'.\ (\mathsf{C_s} \cup \mathsf{P}) \rightsquigarrow^{*} t' \ \wedge\ t' \sim t \end{aligned}$$

This definition states the following: Consider any source program part[3] P and its compilation P↓. If P↓ linked with some (arbitrarily chosen) target context $\mathbf{C_t}$ emits a finite trace prefix t, then there must exist a source context $\mathsf{C_s}$ that when linked P is able to cause P to emit a related trace prefix t'.

$$\begin{tikzcd}
 & t_1\uparrow = (\mathsf{C_s} \cup \mathsf{P}') \rightsquigarrow^{*} t_{backtr} \arrow[dr, "\textit{II. Forward Compiler Correctness}"] & & (\mathsf{C_s} \cup \mathsf{P}) \rightsquigarrow^{*} t_{QED} \\
(\mathbf{C_t} \cup \mathsf{P}\!\downarrow) \rightsquigarrow^{*} t_1 \arrow[ur, "\textit{I. Back-translation}"] \arrow[rrr, bend right, "\textit{III. Recomposition}"'] & & (\mathsf{C_s}\!\downarrow \cup \mathsf{P}'\!\downarrow) \rightsquigarrow^{*} t_2 \arrow[r] & (\mathsf{C_s}\!\downarrow \cup \mathsf{P}\!\downarrow) \rightsquigarrow^{*} t_{1,2} \arrow[u, "\textit{IV. Backward Compiler Correctness}"']
\end{tikzcd}$$

Fig. 2.1: Generic proof technique [3] for ***RSP***$^{\sim}$. The traces t_1, t_{backtr}, t_2, $t_{1,2}$, and t_{QED} are pairwise related by relation $\sim$.

To understand why this definition captures secure compilation in **Setting 2**, consider the case where t is a trace witnessing the violation of a safety property of interest. Then, if the compiler has ***RSP***$^{\sim}$, there must be a source context, which causes a similar violation entirely *in the source language*. In other words, an attack from some target-level context can only arise if the source program is vulnerable to a similar attack from some source-level context. In our particular example, since there clearly is no source context violating **nowrite**, no target context can violate **nowrite** either.

Practically, a compilation chain attains ***RSP***$^{\sim}$ (or any other criterion for secure compilation of partial programs) by enforcing source language abstractions like memory safety against arbitrary target contexts. For this, the compiler and its associated runtime may rely on hardware support for memory isolation [50], or bounds checking (e.g., hardware capabilities [55, 58]), or programmable tagged architectures [14, 17], or it may combine control-flow integrity [2], software fault isolation [54], and software bounds checking [33].

Our goal in this paper is to explain that proving ***RSP***$^{\sim}$ in the presence of memory sharing is difficult and to develop proof techniques for doing this. The specific target language we use is memory safe, which simplifies the compilation chain's enforcement aspect substantially.[4] However, even in this setting, the difficulties in *proving* ***RSP***$^{\sim}$ with memory sharing show up prominently.

Finally, the definition of ***RSP***$^{\sim}$ is indexed by a relation $\sim$ between source and target traces. The concrete instantiation of this relation determines how safety properties transform from source to target [5]. In our setting, $\sim$ is a bijective renaming relation on memory addresses, which we describe later (Definition 3.10).

2.3 A Proof Strategy for Robust Safety Preservation

RSP$^{\sim}$ can be proved in various ways [3, 4, 38]. Here, we adapt a proof strategy by Abate et al. [3], since it *reuses* the proof of compiler correctness, thus avoiding duplication of work. Figure 2.1 summarizes the proof strategy.[5]

[2]As a notational convention, we use different fonts and colors for source language elements and **target language elements**. Common elements are written in normal black font. We also use the symbol ↓ for the compiler's translation function.

[3]We use the notation uppercase P for a program, partial or whole. But only whole programs can execute. Whole-program execution is denoted $P \rightsquigarrow^{*} t$ or $P \xrightarrow{t} s$ where s is a state reached after emitting a trace prefix t.

[4]Note that our source and target languages differ significantly in their control flow constructs, so the compiler itself is nontrivial; only its security enforcement aspect is rather straightforward.

[5]Abate et al. [3] instantiate the strategy mostly for $\sim$ set to equality, while we use a nontrivial $\sim$ everywhere, but this difference is less important here.

Overall, Abate et al. [3]'s proof of $\boldsymbol{RSP}^{\sim}$ consists of four steps, two of which are immediate from compiler correctness. $\boldsymbol{RSP}^{\sim}$ requires starting from $(\mathbf{C_t} \cup \mathsf{P}{\downarrow}) \rightsquigarrow^* t_1$ to demonstrate the existence of a $\mathsf{C_s}$ such that $(\mathsf{C_s} \cup \mathsf{P}) \rightsquigarrow^* t_{QED}$. The first proof step uses *back-translation* (Lemma 2.2) to show from $(\mathbf{C_t} \cup \mathsf{P}{\downarrow}) \rightsquigarrow^* t_1$ that there exist $\mathsf{C_s}$ and P' such that $(\mathsf{C_s} \cup \mathsf{P}') \rightsquigarrow^* t_{backtr}$ with $t_1 \sim t_{backtr}$. Note that the back-translation produces both a new context and a new program part, and that P' may be completely different from P. The second step directly uses a form of compiler correctness called forward compiler correctness (Assumption 2.3), to conclude that the compilation of this new source program, $(\mathsf{C_s} \cup \mathsf{P}'){\downarrow} = \mathsf{C_s}{\downarrow} \cup \mathsf{P}'{\downarrow}$, produces t_2, related to t_1. At this point, we have two target programs – $\mathbf{C_t} \cup \mathsf{P}{\downarrow}$ and $\mathsf{C_s}{\downarrow} \cup \mathsf{P}'{\downarrow}$ – that produce related traces t_1 and t_2. The third step uses an innovative target-language lemma, *recomposition* (Lemma 2.4), to show that a third program $\mathsf{C_s}{\downarrow} \cup \mathsf{P}{\downarrow}$, which takes $\mathsf{P}{\downarrow}$ from the first program and $\mathsf{C_s}{\downarrow}$ from the second, also produces a related trace $t_{1,2}$. The final, fourth step uses another form of compiler correctness, called backward compiler correctness (Assumption 2.5), to conclude from this that the corresponding source, $\mathsf{C_s} \cup \mathsf{P}$ produces a related trace t_{QED}. This concludes the proof.

Lemma 2.2 (Whole-Program Back-translation [3])**.**

$$\forall \mathbf{P}\ t.\ \mathbf{P} \rightsquigarrow^* t \implies \exists \mathsf{P}\ t'.\ \mathsf{P} \rightsquigarrow^* t' \ \wedge\ t' \sim t$$

Assumption 2.3 (Whole-Program Forward Compiler Correctness)**.**

$$\forall \mathsf{P}\ t.\ \mathsf{P} \rightsquigarrow^* t \implies \exists t'.\ \mathsf{P}{\downarrow} \rightsquigarrow^* t' \ \wedge\ t' \sim t$$

Lemma 2.4 (Recomposition [3])**.**

$$\begin{aligned}&\forall \mathbf{P_1}\ \mathbf{C_1}\ \mathbf{P_2}\ \mathbf{C_2}\ t_1 t_2.\\ &\quad (\mathbf{P_1} \cup \mathbf{C_1}) \rightsquigarrow^* t_1 \implies (\mathbf{P_2} \cup \mathbf{C_2}) \rightsquigarrow^* t_2 \implies\\ &\quad t_1 \sim t_2 \implies \exists t_{1,2}.\ (\mathbf{P_1} \cup \mathbf{C_2}) \rightsquigarrow^* t_{1,2} \ \wedge\ t_{1,2} \sim t_1\end{aligned}$$

Assumption 2.5 (Whole-Program Backward Compiler Correctness)**.**

$$\forall \mathsf{P}\ t.\ \mathsf{P}{\downarrow} \rightsquigarrow^* t \implies \exists t'.\ \mathsf{P} \rightsquigarrow^* t' \ \wedge\ t' \sim t$$

By following this proof strategy, Abate et al. [3] are able to reuse compiler correctness and reduce the entire proof of $\boldsymbol{RSP}^{\sim}$ to two key lemmas: back-translation (Lemma 2.2) and recomposition (Lemma 2.4). However, Abate et al. execute this strategy for languages without any memory sharing between components. Their components—both source and target—communicate only through function call arguments and return values. As such, our earlier example cannot even be expressed in their setting. In the rest of this paper, we adapt their proof strategy for $\boldsymbol{RSP}^{\sim}$ to the setting where memory sharing is allowed. We show that memory sharing significantly complicates the proofs of both back-translation and recomposition, and requires new proof techniques. However, before explaining these, we briefly show what traces actually look like.

Interaction traces A trace or, more precisely, an interaction trace, is a modeling and proof artifact that arises from an augmented reduction semantics of a language, wherein certain steps are labeled with descriptors called *events*. The sequence of events along a reduction sequence forms a trace, denoted t. In prior work on secure compilation, only steps involving cross-component interactions or external communication (input-output) have been labeled with events. In contrast, internal steps within a component have not been labeled with events. For example, in Abate et al. [3]'s setting without shared memory, cross-component interaction happens through cross-component calls and returns only (information crosses components via function arguments and return values only). Hence, their events are only cross-component calls and returns. We denote these events e_{no_shr} where the subscript *no_shr* stands for "no memory sharing".

$$e_{no\_shr} ::= \mathtt{Call}\ c_{caller}\ c_{callee}.f(v) \mid \mathtt{Ret}\ c_{prev}\ c_{next}\ v$$

The event $\mathtt{Call}\ c_{caller}\ c_{callee}.f(v)$ represents a call from component c_{caller} to the function f of component c_{callee} with argument v. The dual event $\mathtt{Ret}\ c_{prev}\ c_{next}\ v$ represents a return from component c_{prev} to component c_{next} with return value v. Along a trace, calls and returns are always well-bracketed (the semantics of both the source and target languages enforce this well bracketing).

In our setting, memory shared between components is another medium of interaction, so reads and writes to it must be represented on interaction traces. However, our languages are sequential (only one component executes at a time), so writes to shared memory made by a component become visible to another component only when the writing component transfers control to the other component. As such, to capture interactions between components, it suffices to record the state of the shared memory when control transfers from one component to another, i.e., at cross-component calls and returns. For this, we modify call and return events to also record the state of the memory shared up to the time of the event (the shared part of memory grows along an execution as more pointers are passed across components). The new events, denoted e, are defined below. The shared memory on each event, written *Mem*, is underlined for emphasis only. Technically, *Mem* is a just a partial map from locations l to values v, which themselves can be pointers to locations.

Definition 2.6 (Interaction-trace events e with memory sharing)**.**

$$e ::= \mathtt{Call}\ \underline{Mem}\ c_{caller}\ c_{callee}.f(v) \mid \mathtt{Ret}\ \underline{Mem}\ c_{prev}\ c_{next}\ v$$

Interaction traces serve two broad purposes. First, they are used to express safety properties of interest, such as the **nowrite** property in our earlier example. (Appendix A shows how **nowrite** can be expressed as a predicate on interaction traces.) Second, as we explain in §3, interaction traces are essential to the proof of back-translation, Lemma 2.2. One of our key insights is that, with memory sharing, enriching interaction traces with selective information about data flows *within* a component can simplify the proof of back-translation considerably.

3 Key Technical Ideas

Next, we describe why the proofs of Lemma 2.2 and Lemma 2.4 become substantially more complex in the presence of memory sharing, and our new techniques—data-flow back-translations and turn-taking simulations—that offset some of the extra complexity.

3.1 Data-Flow Back-translation

In proving Lemma 2.2, we are given a target language whole program **P** and an interaction trace t that it produces, and we have to construct a source language whole program P that produces a related interaction trace t'. This process of constructing the source program P is often called *back-translation* (hence, the name of the lemma). Obviously, we can construct P from either **P** or t. Prior work has considered both approaches.

Construction of P from **P**, which we call *syntax-directed back-translation*, typically works by simulating **P** in the source language [6, 7, 15, 16, 36, 40, 46, 51, 53]. This is tractable when every construct of the target language can be simulated easily in the source. However, for many pairs of languages, including our source and target languages (§4), this is not the case.

The alternative then is to construct P from the given target trace t [3, 18, 37, 39]. This alternative, which we call *trace-directed back-translation*, should be easier in principle since the interaction trace only records cross-component interactions, so there is no need to simulate every language construct in the source; instead, only constructs that can influence cross-component interactions need to be simulated.

Indeed, trace-directed back-translation is fairly straightforward when there is no memory sharing [3, 39] or when memory references (pointers) can be constructed from primitive data like integers in the source language (the latter is true in unsafe languages like C). However, with memory sharing and unforgeable memory references in the source—something that is common in safe source languages like Java, Rust, Go and ML—trace-directed back-translation is really difficult. To understand this, consider the following simple example.

Example 3.1. Suppose we want to back-translate the following interaction trace with four events:

$$\begin{aligned} &\texttt{Call}\ \mathit{Mem}\ c_1\ c_2.f(l_1)\ ::\ \texttt{Ret}\ \mathit{Mem}\ c_2\ c_1\ 0 \\ ::\ &\texttt{Call}\ \mathit{Mem}'\ c_1\ c_2.g()\ ::\ \texttt{Ret}\ \mathit{Mem}'\ c_2\ c_1\ l_2 \end{aligned}$$

where l_1 and l_2 are distinct memory locations, $\mathit{Mem} = [l_1 \mapsto l_2, l_2 \mapsto 0]$ and $\mathit{Mem}' = [l_1 \mapsto 100, l_2 \mapsto 0]$.[6]

In this example, the module c_1 calls c_2 twice – first it calls $c_2.f()$ and then it calls $c_2.g()$. Assume that prior to these calls, c_1 dynamically allocated locations l_1 and l_2. In the first call, c_1 passes the location l_1 to c_2 as the function call argument. At this point l_1 happens to contain l_2. As a result, the first call shares *both* l_1 and l_2 with c_2 – it shares l_1 directly, and shares l_2 indirectly via l_1. $c_2.f()$ returns 0 to c_1 without changing the shared memory. Later, c_1 overwrites l_1's contents with the value 100 (to get a new shared memory Mem'), and calls $c_2.g()$. This time c_2 returns l_2 to c_1.

Note that at the time of second call, l_2 is actually not reachable from the shared memory Mem'. However, c_2 could have stashed l_2 somewhere in its private memory during the first call and retrieved it from there during the second call to return it.

The question is how we can back-translate this sequence of interactions into a source program. If pointers were forgeable in the source language, this would be quite easy: l_2, being forgeable, could simply be hardcoded in the body of the simulating source function $c_2.g()$.

However, when locations cannot be forged in the source language, this is not straightforward. Now, we cannot hardcode l_2 into the back-translated $c_2.g()$'s body since l_2 is dynamically allocated by c_1 during execution! Consequently, the back-translated component c_2 *must* store l_2 in an indexed data structure during the first call (to $c_2.f()$), and then somehow retrieve it from that data structure in the second call.

The problem is actually more difficult than this example shows: The back-translated context must fetch and store in its indexed data structure *all* pointers that become accessible to it, directly or indirectly, since these pointers may show up on the trace later. Although some prior work has used such a bookkeeping data structure, this is an immensely difficult construction [18, 37], because additional, complex invariants about this data structure must be proved.

The new idea This is where our new idea of *data-flow back-translation* comes in. We enrich interaction traces of the target language – only for the purposes of the back-translation proof – with information about *all* data-flows, even those *within* a single component (within its private state). We call these enriched traces *data-flow traces*. From the target language's reduction semantics, we can easily prove that every interaction trace as described above can be enriched to a data-flow trace (Lemma 3.4 below). And, given such a data-flow trace, we can easily back-translate to a simulating source program, since we know exactly how pointers flow. In the example above, the enriched trace would tell us exactly what $c_2.f()$ did to store l_2 and how $c_2.g()$ retrieved it later. We can then mimic this in the constructed source program (see Example 3.3 below).

Concretely, we define a new class of data-flow traces, denoted T, whose events, $\mathcal{E}$, extend those of interaction traces to capture all possible data flows in the target language. In the following, we show the events for our target language (§4), which is a memory-safe assembly-like language with registers and memory. The events `dfCall` and `dfRet` are just the `Call` and `Ret` events of interaction traces (Definition 2.6). The remaining events correspond to target language instructions

[6]Technically, in our languages, function calls and returns and, hence, interaction traces carry *pointers* to locations, not locations themselves. However, in this section, we blur this distinction.

that cause data flows: loading a constant to a register (`Const`), copying from a register to another (`Mov`), binary operations (`BinOp`), copying from a register to memory or vice-versa (`Store`, `Load`) and allocating a fresh location (`Alloc`). Importantly, in a data-flow trace, every event records the entire state – both shared state and state private to individual components. Accordingly, in the events below, *Mem* also includes locations that are private to a component, and *Reg* is the state of the register file.

Definition 3.2 (Events of data-flow traces)**.**

$$\begin{aligned}\mathcal{E} ::= {} & \texttt{dfCall}\ \mathit{Mem}\ \mathit{Reg}\ c_{caller}\ c_{callee}.\mathit{proc}(v)\\ \mid {} & \texttt{dfRet}\ \mathit{Mem}\ \mathit{Reg}\ c_{prev}\ c_{next}\ v\\ \mid {} & \texttt{Const}\ \mathit{Mem}\ \mathit{Reg}\ c_{cur}\ v\ r_{dest}\\ \mid {} & \texttt{Mov}\ \mathit{Mem}\ \mathit{Reg}\ c_{cur}\ r_{src}\ r_{dest}\\ \mid {} & \texttt{BinOp}\ \mathit{Mem}\ \mathit{Reg}\ c_{cur}\ op\ r_{src1}\ r_{src2}\ r_{dest}\\ \mid {} & \texttt{Load}\ \mathit{Mem}\ \mathit{Reg}\ c_{cur}\ r_{addr}\ r_{dest}\\ \mid {} & \texttt{Store}\ \mathit{Mem}\ \mathit{Reg}\ c_{cur}\ r_{addr}\ r_{src}\\ \mid {} & \texttt{Alloc}\ \mathit{Mem}\ \mathit{Reg}\ c_{cur}\ r_{ptr}\ r_{size}\end{aligned}$$

Example 3.3. Consider the following data-flow trace, which enriches a part of Example 3.1's interaction trace – the part that covers the call and return to $c_2.f()$ only. Here, l is a fixed, hardcodable location that can always be accessed by c_2, r_{COM} is a special register used to pass arguments and return values, and Mem_1 and Reg_1 are some initial states of memory and registers, respectively.

$$\begin{aligned}& \texttt{dfCall}\ \mathit{Mem}_1\ (\mathit{Reg}_1[r_{COM} \mapsto l_1])\ c_1\ c_2.f(l_1)\\ ::\ & \texttt{Const}\ \mathit{Mem}_1\ (\mathit{Reg}_1[r_{COM} \mapsto l_1, r_1 \mapsto l])\ c_2\ l\ r_1\\ ::\ & \texttt{Load}\ \mathit{Mem}_1\ (\mathit{Reg}_1[r_{COM} \mapsto l_1, r_1 \mapsto l, r_2 \mapsto l_2])\ r_{COM}\ r_2\\ ::\ & \texttt{Store}\ (\mathit{Mem}_1[l \mapsto l_2])\ (\mathit{Reg}_1[r_{COM} \mapsto l_1, \ldots])\ r_1\ r_2\\ ::\ & \texttt{dfRet}\ (\mathit{Mem}_1[l \mapsto l_2])\ (\mathit{Reg}_1[r_{COM} \mapsto 0, \ldots])\ c_2\ c_1\ 0\end{aligned}$$

This data-flow trace shows clearly how $c_2.f()$ stashed away l_2: It copied l_2 to the register r_2 and then to the location l. The rest of the data-flow trace (not shown) will also show precisely how $c_2.g()$ later retrieved l_2. It is not difficult to construct a source program that mimics these data flows step-by-step, by using source memory locations to mimic the target's register file and memory (see §5 for further details).

Outline of data-flow back-translation proof Data-flow traces simplify the proof of back-translation (Lemma 2.2) by splitting it into two key lemmas: Enriching interaction traces to data-flow traces (Lemma 3.4) and back-translation of data-flow traces (Lemma 3.5), both of which are shown below and are much easier to prove that standard trace-based backtranslation. Recall that T denotes a data-flow trace. $\texttt{remove\_df}(T)$ denotes the interaction trace obtained by removing all internal data-flow events from T, i.e., by retaining only `Call` and `Return` events.

Lemma 3.4 (Enrichment)**.**

$$\forall \mathbf{P}\ t.\ \mathbf{P} \rightsquigarrow^* t \implies \exists T.\ \mathbf{P} \rightsquigarrow^* T\ \wedge\ t = \texttt{remove\_df}(T)$$

Proof. Immediate from the definition of the target-language semantics. □

Lemma 3.5 (Data-flow back-translation)**.**

$$\forall \mathbf{P}\ T.\ \mathbf{P} \rightsquigarrow^* T \implies \exists \mathsf{P}\ t.\ \mathsf{P} \rightsquigarrow^* t\ \wedge\ t \sim \texttt{remove\_df}(T)$$

Proof sketch. By constructing a P that simulates the data flows in T, thus keeping its state in lock-step with the state in T's events. See §5 for further details. □

Composing these 2 lemmas immediately yields Lemma 2.2.

3.2 Turn-Taking Simulation for Recomposition

Next, we turn to recomposition (Lemma 2.4). This lemma states that if two programs $\mathbf{P_1} \cup \mathbf{C_1}$ and $\mathbf{P_2} \cup \mathbf{C_2}$ produce two related interaction traces, then the program $\mathbf{P_1} \cup \mathbf{C_2}$ can also produce an interaction trace related to both those traces.[7] We refer to $\mathbf{P_1} \cup \mathbf{C_1}$ and $\mathbf{P_2} \cup \mathbf{C_2}$ as *base* programs, and to $\mathbf{P_1} \cup \mathbf{C_2}$ as the *recomposed* program. We say that the partial programs $\mathbf{P_1}$ and $\mathbf{C_2}$ are *retained* by the recomposition, and that $\mathbf{P_2}$ and $\mathbf{C_1}$ are *discarded*.

The proof of recomposition is a ternary simulation over executions of the three programs. For this, we need a ternary relation between a pair of states $\mathbf{s_1}$ and $\mathbf{s_2}$ of the base programs and a state $\mathbf{s_{1,2}}$ of the recomposed program. The question is how we can relate the *memories* in $\mathbf{s_1}$ and $\mathbf{s_2}$ to that in $\mathbf{s_{1,2}}$.

In the absence of memory sharing, as in Abate et al. [3], this is fairly straightfoward: We simply project $\mathbf{P_1}$'s memory from $\mathbf{s_1}$, $\mathbf{C_2}$'s memory from $\mathbf{s_2}$, put them together (take a disjoint union), and this yields the memory of $\mathbf{s_{1,2}}$ (up to location renaming, which we capture with a relation $\sim_{\texttt{ren}}$ that is formally defined later).

Definition 3.6 (Memory relation of Abate et al. [3])**.**

$$\texttt{mem\_rel}(\mathbf{s_1}, \mathbf{s_2}, \mathbf{s_{1,2}}) \overset{\text{def}}{=}$$
$$\mathbf{s_{1,2}}.\mathit{Mem} \sim_{\texttt{ren}} \texttt{proj}_{\mathbf{P_1}}(\mathbf{s_1}.\mathit{Mem})\ \uplus\ \texttt{proj}_{\mathbf{C_2}}(\mathbf{s_2}.\mathit{Mem})$$

However, with memory sharing, this definition no longer works.

Example 3.7. Consider the following three target-language components $\mathbf{C_1}$, $\mathbf{C_2}$ and $\mathbf{P_1}$, represented in C-like syntax for simplicity. The fourth component $\mathbf{P_2}$ is irrelevant for this explanation, hence not shown.

```
component C1 {
  int* ptr_to_P1 = malloc();
  void store(int* arg) {
    ptr_to_P1 = arg;
    int val_to_revert = *ptr_to_P1;
    *ptr_to_P1 = 42;
    ...
    *ptr_to_P1 = val_to_revert;
  }
}
```

[7] *Note.* Traces in this section refer to the interaction traces of Definition 2.6. Data-flow traces are used only for back-translation, not for recomposition.

```
component C₂ {
  int* ptr_to_P1_or_P2 = malloc();
  void store(int* arg) {
    ptr_to_P1_or_P2 = arg;
  }
}
```

```
component P₁ {
  int* priv_ptr = malloc();
  int* shared_ptr = malloc();
  void call_store() {
    store(shared_ptr);
  }
}
```

In the base program $P_1 \cup C_1$, P_1 shares `shared_ptr` with the function C_1.`store`(). This function *temporarily* updates `shared_ptr` but reverts it to its original value before returning. Somewhat differently, in the recomposed program $P_1 \cup C_2$, C_2.`store`() does *not* modify `shared_ptr` at all. Thus, even though the end-to-end interaction behavior of `store`() in both the runs is exactly the same, `shared_ptr` (which is actually in P_1's memory) has been temporarily modified in C_1.`store`() but not in C_2.`store`(). Consequently, *during* the execution of the context's function `store`(), the memory relation of Definition 3.6 does not hold.

More abstractly, the problem here is that P_1's *shared* memory in the recomposed program $P_1 \cup C_2$ can be related to that in the base program $P_1 \cup C_1$ *only while* control is in P_1. When control is in C_2, the contents of P_1's shared memory can change unrelated to the base runs. This naturally leads to the following program counter-aware memory relation.

Definition 3.8 (Our memory relation (first attempt))**.**

$$\texttt{mem\_rel\_pc}(s_1, s_2, s_{1,2}) \overset{\text{def}}{=}$$
$$\textit{if } s_{1,2} \textit{ is executing in } P_1 \textit{ then:}$$
$$\quad \texttt{proj}_{P_1}(s_{1,2}.\mathit{Mem}) \sim_{\texttt{ren}} \texttt{proj}_{P_1}(s_1.\mathit{Mem})$$
$$\textit{else: } (\textit{i.e.}, , s_{1,2} \textit{ is executing in } C_2)$$
$$\quad \texttt{proj}_{C_2}(s_{1,2}.\mathit{Mem}) \sim_{\texttt{ren}} \texttt{proj}_{C_2}(s_2.\mathit{Mem})$$

Although this definition relates *shared* memory correctly, it is inadequate for P_1's *private* memory – the memory P_1 has not shared with the context in the past, such as the pointer `priv_ptr` in Example 3.7. This private memory must remain related in the base program $P_1 \cup C_1$ and the recomposed program $P_1 \cup C_2$ independent of where the execution is. However, Definition 3.8 does not say this.

Accordingly, we revise our definition again. To determine which locations have been shared and which are still private, we rely on the interaction trace prefixes t_1, t_2 and $t_{1,2}$ that are emitted before reaching the states s_1, s_2 and $s_{1,2}$, respectively. For a memory *mem* and a trace t, we write $\texttt{shared}(mem, t)$ for the projection of *mem* on addresses that are shared on the trace t and $\texttt{private}(mem, t)$ for the projection of *mem* on all the other addresses. With this, we can finally define a *turn-taking relation* `mem_rel_tt` that accurately describes the memory $s_{1,2}.\mathit{Mem}$ of the recomposed program in terms of the memories $s_1.\mathit{Mem}$ and $s_2.\mathit{Mem}$ of the two base programs:

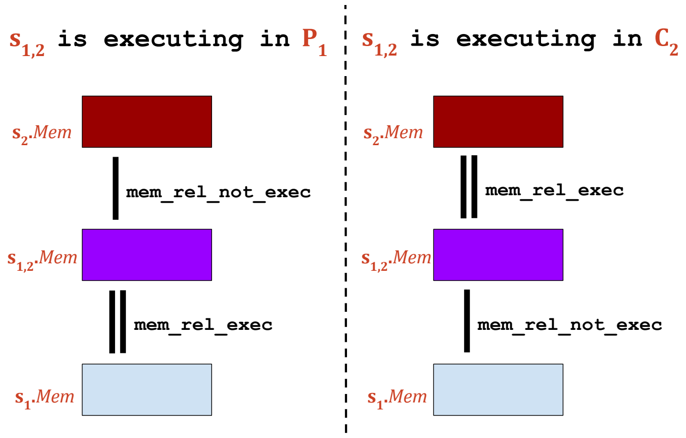


Fig. 3.1: The turn-taking memory relation, `mem_rel_tt`.

Definition 3.9 (Turn-Taking Memory Relation)**.**

$$\texttt{mem\_rel\_tt}(s_{1,2}, s_1, s_2, t_{1,2}, t_1, t_2) \overset{\text{def}}{=}$$
$$\textit{if } s_{1,2} \textit{ is executing in } P_1 \textit{ then:}$$
$$\quad \texttt{mem\_rel\_exec}(P_1, t_1, t_{1,2}, s_1.\mathit{Mem}, s_{1,2}.\mathit{Mem}) \land$$
$$\quad \texttt{mem\_rel\_not\_exec}(C_2, t_2, t_{1,2}, s_2.\mathit{Mem}, s_{1,2}.\mathit{Mem})$$
$$\textit{else: } (\textit{i.e.}, , s_{1,2} \textit{ is executing in } C_2)$$
$$\quad \texttt{mem\_rel\_exec}(C_2, t_2, t_{1,2}, s_2.\mathit{Mem}, s_{1,2}.\mathit{Mem}) \land$$
$$\quad \texttt{mem\_rel\_not\_exec}(P_1, t_1, t_{1,2}, s_1.\mathit{Mem}, s_{1,2}.\mathit{Mem})$$

where

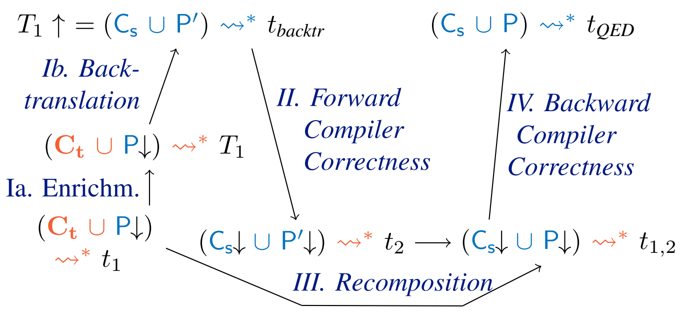
$$\texttt{mem\_rel\_exec}(\mathrm{part}, t, t_{1,2}, \mathrm{m_{base}}, \mathrm{m_{recomp}}) \overset{\text{def}}{=}$$
$$\texttt{proj}_{\mathrm{part}}(\mathrm{m_{recomp}}) \sim_{\texttt{ren}} \texttt{proj}_{\mathrm{part}}(\mathrm{m_{base}}) \land$$
$$\texttt{shared}(\mathrm{m_{recomp}}, t_{1,2}) \sim_{\texttt{ren}} \texttt{shared}(\mathrm{m_{base}}, t)$$

and

$$\texttt{mem\_rel\_not\_exec}(\mathrm{part}, t, t_{1,2}, \mathrm{m_{base}}, \mathrm{m_{recomp}}) \overset{\text{def}}{=}$$
$$\texttt{proj}_{\mathrm{part}}(\mathrm{m_{recomp}}) \cap \texttt{private}(\mathrm{m_{recomp}}, t_{1,2})$$
$$\sim_{\texttt{ren}} \texttt{proj}_{\mathrm{part}}(\mathrm{m_{base}}) \cap \texttt{private}(\mathrm{m_{base}}, t)$$

Intuitively, Definition 3.9 says the following about P_1's memory: (a) While P_1 executes, P_1's entire memory – both private and shared – relates in the runs of the base program $P_1 \cup C_1$ and the recomposed program $P_1 \cup C_2$. (b) While the contexts (C_1 and C_2) execute, only the private memory of P_1 in these two runs is related. For the context's memory, the dual relation holds. Figure 3.1 depicts this visually.

The memory relation $\sim_{\texttt{ren}}$. We explain the memory relation $\sim_{\texttt{ren}}$ that appears in the above definitions. This relation simply allows for a consistent renaming of memory locations up to a partial bijection. The need for this renaming arises because corresponding program parts may differ in the layouts of their



$T_1 \uparrow = (C_s \cup P') \rightsquigarrow^* t_{backtr}$
$(C_s \cup P) \rightsquigarrow^* t_{QED}$
Ib. Back-translation
II. Forward Compiler Correctness
IV. Backward Compiler Correctness
$(C_t \cup P\downarrow) \rightsquigarrow^* T_1$
Ia. Enrichm.
$(C_t \cup P\downarrow) \rightsquigarrow^* t_1$
$(C_s\downarrow \cup P'\downarrow) \rightsquigarrow^* t_2 \longrightarrow (C_s\downarrow \cup P\downarrow) \rightsquigarrow^* t_{1,2}$
III. Recomposition

Fig. 3.2: Our proof technique for ***RSP***$^\sim$ with memory sharing. The interaction traces t_1, $\texttt{remove\_df}(T_1)$, t_{backr}, t_2, $t_{1,2}$, and t_{QED} are pairwise related by the trace relation $\sim$.

private memories. In the example above, consider the case where the module $\mathbf{P_2}$, which we didn't show until now, is the same as $\mathbf{P_1}$, just without the private pointer `priv_ptr` and the corresponding malloc. In this case, the exact value of `shared_ptr` could differ across the base run $\mathbf{P_2} \cup \mathbf{C_2}$ and the recomposed run $\mathbf{P_1} \cup \mathbf{C_2}$.

Formally, `ren` denotes a partial bijection that may depend on $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{C_1}$ and $\mathbf{C_2}$, and $\sim_{\texttt{ren}}$ is renaming of memories (both locations and their contents) up to `ren`.

Proof of recomposition. In our Coq proof we show that the turn-taking memory relation of Definition 3.9 is an invariant (for the languages and compiler of §4). Using this, we are able to prove recomposition (Lemma 2.4). A key additional idea we use is *strengthening*, which allows us to strengthen `mem_rel_not_exec`, which relates only the private memory of a component, to `mem_rel_exec`, which relates private and shared memories of the same component, at cross-component calls and returns. Strengthening follows from the assumption that the two base runs produce related interaction traces. §B provides additional details.

3.3 Applying our ideas to an *RSP*$^\sim$ proof

Figure 3.2 summarizes our overall proof technique for proving ***RSP***$^\sim$ with dynamic memory sharing. $\uparrow$ denotes the data-flow back-translation function. Relative to Abate et al.'s [3] proof technique shown in Figure 2.1, the two key changes are that: (1) Step I (back-translation) has now been factored into two steps Ia and Ib to use data-flow traces. Steps Ia and Ib correspond to Lemma 3.4 and Lemma 3.5, respectively. (2) The proof of step III (recomposition) now relies on turn-taking simulations. Steps II and IV, which simply reuse compiler correctness, remain unchanged.

Trace relation $\sim$ We also define the trace relation $\sim$, which we referred to in §2 and §3. The relation says that two traces are related if corresponding events have the same kind (both call or both return, and between the same components), and there is a bijective renaming of locations `ren` such that the memories mentioned in corresponding events of the traces are related by $\sim_{\texttt{ren}}$ (§3.2), and so are the arguments of calls and returns.

Definition 3.10 (Relation on interaction traces)**.** *For address renaming relations* `ren`*, suppose* $\sim_{\texttt{ren}}$ *is the memory renaming relation described in §3.2.*

$$\begin{aligned} t_1 \sim t_2 \overset{\text{def}}{=} \ & \exists \texttt{ren}.\ \forall i. \\ & t_1[i].\mathit{Mem}\ \sim_{\texttt{ren}}\ t_2[i].\mathit{Mem} \\ & \wedge\ \texttt{match\_events}(t_1[i], t_2[i]) \\ & \wedge\ \texttt{valren}_{\texttt{ren}}(t_1[i].\mathit{arg}, t_2[i].\mathit{arg}) \end{aligned}$$

`match_events` *says that the kind of the two events* $t_1[i]$ *and* $t_2[i]$ *and the component ids (e.g., caller and callee) appearing on them are the same.*

$\texttt{valren}_{\texttt{ren}}$ *is a value renaming relation that just lifts the memory renaming relation* `ren` *to pointers.*

4 Concrete Languages and Compiler Pass

Next, we describe specific source and target languages – SafeP and Mach, respectively – and a specific compiler from the source to the target language. This specific setup is the testbed on which we have instantiated our new ideas from §3.

In both our languages, a program P consists of an interface, a set of named functions and a set of statically allocated data buffers. The interface divides the program into components (denoted c) and assigns every function to a component. It also defines which functions are imported and exported by each component.

Both languages are memory safe and share the same memory model, which is adapted from CompCert's block-memory model [28]. Briefly, memory consists of an unbounded number of finite and isolated *blocks* of values. A value v may be an integer, an (unforgeable) pointer, or a special undefined value. The memory can be seen as a collection of individual component memories because each memory block is initially accessible to only the allocating component. A *pointer* is a tuple $(perm, c, b, o)$ consisting of a permission $perm$ (used to distinguish code and data pointers), the allocating component c, a unique block identifier b, and an integer offset o within the block. A location, which we denoted by l so far, is a triple of a component, block and an offset, (c, b, o). A memory maps locations to values.

Pointers can be incremented or decremented (pointer arithmetic), but this only changes the offset o. The block identifier b cannot be changed by any language operation. Additional metadata not shown here tracks the size of each allocated block. Any dereference of a data pointer with an offset beyond the allocated size or any call/jump to a code pointer with a non-zero offset causes the program to get stuck. This immediately enforces memory safety.

Although these languages are strongly inspired by those of Abate et al. [3], unlike them, we allow a component to pass pointers to blocks it allocates to other components. The receiving components can dereference these pointers, possibly after changing offsets. However, no component can access a block which it did not allocate and to which it did not receive a pointer. Hence, our languages provide selective memory sharing at block granularity.

exp	::= v	values
	| arg	function argument
	| local	local static buffer
	| exp$_1$ $\otimes$ exp$_2$	binary operations
	| exp$_1$; exp$_2$	sequence
	| if exp$_1$ then exp$_2$ else exp$_3$	conditional
	| alloc exp	memory allocation
	| !exp	dereferencing
	| exp$_1$:= exp$_2$	assignment
	| c.func(exp)	function call
	| $*$[exp$_1$](exp$_2$)	call pointer
	| &func	function pointer
	| exit	terminate

Fig. 4.1: Syntax of source language expressions

```
instr ::= Const i -> r               | Bnz r L
        | Mov r_s -> r_d             | Jump r
        | BinOp r_1 ⊗ r_2 -> r_d     | JumpFunPtr r
        | Label L                    | Jal L
        | PtrOfLabel L -> r_d        | Call c func
        | Load *r_p -> r_d           | Return
        | Store *r_p <- r_s          | Nop
        | Alloc r_1 r_2              | Halt
```

Fig. 4.2: Instructions of the target language

The operational semantics of both languages produce interaction traces of events from Definition 2.6, recording cross-component calls and returns. Calls and returns are necessarily well-bracketed in the semantics. Despite these commonalities, the two languages differ significantly in the constructs allowed within the bodies of functions.

The source language The body of a SafeP function consists of a single expression, exp, whose syntax is shown in Figure 4.1 and is inspired from the source language of Abate et al. [3]. The construct arg evaluates to the argument of the current function, which is a value (which may be a pointer). There are constructs for if-then-else, dereferencing a pointer (!exp), assigning value to a pointer (exp$_1$:= exp$_2$), calling a function func in component c with argument exp (c.func(exp)), calling a function pointer exp$_1$ ($*$[exp$_1$](exp$_2$)), and taking the address of a function (&func). Additionally, every component has access to a separate statically allocated memory block, whose pointer is returned by the construct local.

Importantly, the source language has only *structured control flow*: Calls and returns are well-bracketed by the semantics, the only explicit branching construct is if-then-else, and indirect function calls with non-zero offsets beyond function entry points are stopped by the semantics.

In fact, we added function pointers in SafeP, not only because they are a natural feature, but also to make specific steps of the back-translation construction a little more convenient. Function pointers allow us e.g., to easily mimic a store of the program counter to memory, an operation that a target-language program routinely performs. Without function pointers in the source, our cross-language value relation may have needed to be a bit complex—complexity that would propagate to the trace relation (Definition 3.10) and to the top-level theorem (Definition 2.1).

The target language Mach is an assembly-like language inspired by RISC architectures, with two high-level features: the block-based memory model shared with SafeP and the component structure provided by interfaces. Its instructions are shown in Figure 4.2. Its state comprises a register file with a separate program counter and an abstract (protected) call stack for cross-component calls, which enforces well-bracketed cross-component Calls and Returns. A designated register $\mathbf{r_{COM}}$ is used for passing arguments and return values. At every cross-component call or return, all registers except $\mathbf{r_{COM}}$ are set to the undefined value.

Importantly, the target language has *unstructured control flow*: One may label statements (instruction **Label L**), jump to labeled statements (**Jump L**, **Bnz r L**), and call labeled statements (**Jal L**). Such unstructured jumps are limited to a single component; cross-component jumps are forbidden by the semantics. Nonetheless, the presence of unstructured control flow in the target language means that a syntax-directed back-translation to the source language, which has only structured control flow, is infeasible.

Like the source language, the target language has an interaction trace semantics with events of Definition 2.6. Following the idea in §3.1, we additionally equip the target language with enriched data-flow traces with events of Definition 3.2.

Compiler from SafeP to Mach Our compiler from SafeP to Mach is single-pass and quite simple. It implements SafeP's structured control flow with labels and direct jumps, intra-component calls using jump-and-link (**Jal**), and function pointer calls using indirect jumps (**Jump** and **JumpFunPtr**). Even this simple compiler suffices to bring out the difficulties in proving compiler security in the presence of memory sharing (as mentioned in §1, our proofs of back-translation and recomposition span 23k lines of Coq code). Using the ideas developed in §3, we can prove that this compiler provides ***RSP***$^{\sim}$ security.

Theorem 4.1. *Our SafeP to Mach compiler is* **RSP**$^{\sim}$ *(i.e., it satisfies Definition 2.1).*

5 Some Details of Mach's Data-Flow Back-Translation

We provide some details of how we back-translate Mach's data-flow traces to SafeP, i.e., how we prove Lemma 3.5. The back-translation function, written $\uparrow$, takes as input a data-flow trace T and outputs a SafeP whole program P

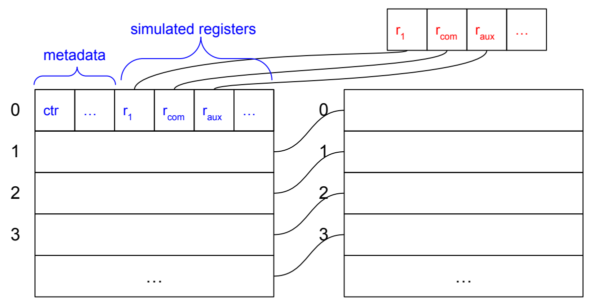


Fig. 5.1: Memory layout of a back-translated component (left) compared to a target component (right)

that produces the (standard) trace $\texttt{remove\_df}(T)$ in SafeP.[8] Similar to Abate et al. [3], each component in P maintains an *event counter* to keep track of which event of the trace P is currently mimicking. This counter, as well as a small amount of other metadata used by the back-translation, is stored inside the statically allocated buffers of each component of P, which are accessed using the local construct.

Mimicking register operations A technical difficulty in the back-translation is that, unlike Mach, SafeP does not have registers. In order to mimic data-flow events involving registers, P simulates these registers and operations on them within the static buffer of the active component. For instance, a Mov *Mem Reg* c_{cur} r_{src} r_{dest} event (which copies a value from register r_{src} to register r_{dest}) is simulated by the expression $(\mathsf{local} + \mathsf{OFFSET}(r_{dest})) := \;!(\mathsf{local} + \mathsf{OFFSET}(r_{src}))$, where $\mathsf{OFFSET}(r)$ is statically expanded to the offset corresponding of register r in the simulated register file.

Mimicking memory operations Because the source and target memory models coincide, we are able to back-translate memory events quite easily. That is, a Store event is back-translated using assignment ($:=$) and a Load event is back-translated using dereferencing (!). Since the static buffer (whose block number is 0 in our semantics) is already used by the back-translation to store metadata and simulated registers, the back-translated program's memory shifts by one block relative to the memory in the target: for each component, block b in the target corresponds to block $b+1$ in the source. The layout of the memory of the back-translated relative to the Mach program is shown in Figure 5.1.

P maintains the invariant that, after simulating an event in T, P's memory and its current component's simulated registers are synchronized with the target memory *Mem* and the target register file *Reg* mentioned in the simulated event (This is defined as part of a mimicking_state invariant — see Lemma 5.1).

[8]↑ also takes as input the *interface* of the given target-language program to be able to mimic the same interface in the source program, but we elide the details as they are not very insightful, and largely similar to those in Abate et al. [3]. P can then be split into a context $\mathsf{C_S}$ and a program part $\mathsf{P'}$ by cutting it along this interface.

Mimicking calls and returns Mach's semantics enforce a calling convention: calls and returns store the argument or return value in $\mathbf{r_{COM}}$, and set all other registers to the undefined value. Therefore, calls and returns in P need extra administrative steps to mimic this convention. For example, mimicking a call event requires two administrative steps: (1) In the caller, dereference the content of the location simulating $\mathbf{r_{COM}}$ to get the argument and pass it to the function, and (2) In the callee, assign the function argument arg to the location simulating $\mathbf{r_{COM}}$, and set all other registers to the undefined value. Similar administrative steps are needed for mimicking a return event.

Proof of back-translation To prove back-translation (Lemma 3.5), we use a simulation lemma that ensures a relation mimicking_state holds between the state of P and the prefix mimicked so far. Intuitively, "mimicking_state T_{pref} T_{suff} s" means that s is the state reached after mimicking all the data-flow events in T_{pref}, and that the starting state of the remaining trace T_{suff} matches s.

Lemma 5.1 (Trace-prefix mimicking)**.**

$$\begin{aligned}
&\forall \mathbf{P}\ T\ T_{pref}\ T_{suff}.\ \ \mathbf{P} \rightsquigarrow^{*} T \implies \\
&\qquad T = T_{pref} \mathbin{+\!+} T_{suff} \implies \\
&\qquad\quad \exists \mathsf{s}\ t'_{pref}.\ \ T\!\uparrow \xrightarrow{t'_{pref}}{}^{*} \mathsf{s} \ \land \\
&\qquad\qquad t'_{pref} \sim \texttt{remove\_df}(T_{pref}) \ \land \\
&\qquad\qquad \texttt{mimicking\_state}\ T_{pref}\ T_{suff}\ \mathsf{s}
\end{aligned}$$

Because Lemma 5.1 ensures the relation mimicking_state holds for every prefix, it effectively states that the memory of the back-translation is in lock-step with the *Mem* and *Reg* appearing in each data-flow event $\mathcal{E}$ from T. mimicking_state is also strong enough to ensure that the trace relation holds between the projection of the prefix mimicked so far $\texttt{remove\_df}(T_{pref})$ and the corresponding prefix t'_{pref} that the back-translation emits.

6 Related Work

Memory relations similar to turn-taking simulations (mem_rel_tt) El-Korashy et al. [18] and Stewart et al. [48] use memory relations that are similar to mem_rel_tt in that the shared memories of two related executions may mismatch and the memory relation guarantees that the context does not modify the *private* memory of the compiled program. However, there are notable differences. First, their relations are *binary*—between two runs that differ only in one component, not both—unlike ours, which is "ternary". This allows their relations to be strengthened whenever the compiled program is executing, while our relation can be strengthened (Definition A.2) only for single steps right after interaction events. Second, the applications are quite different. Stewart et al. [48]'s relation is used in a non-security proof about compositional compiler correctness where guarantees come from undischarged assumptions about the target context, while our guarantees come from runtime safety features of

Mach. El-Korashy et al. [18]'s memory relation is used to establish a different security criterion, full abstraction [1, 40].

Reuse of standard compiler correctness for secure compilation We are aware of only two prior work that reuse compiler correctness lemmas in a secure compilation proof. Abate et al. [3], which we directly build on, have goals similar to ours, but without memory sharing, which is really the focus of our paper. El-Korashy et al. [18] support memory sharing and proof reuse using a different proof technique they call *TrICL*. As explained in the paragraph on memory relations above, their memory relation (which is part of *TrICL*) is technically very different from our turn-taking simulations. Additionally, unlike our technique, their proof is not mechanically verified.

Other kinds of informative traces Using inspiration from fully abstract trace semantics [26], Patrignani and Garg [38] perform back-translation (with shared memory) for a compiler pass using traces that record the *whole* memory but still only emit it at just interaction events. Although more informative than traces that record only shared memory at interaction events [18, 26], these traces still do not eliminate the need for bookkeeping—unlike our data-flow traces that selectively expose *non-interaction events* to simplify back-translation.

Handling memory sharing as message passing Patrignani et al. [39] describe a completely different secure compilation of programs with memory sharing: Their compiled code implements shared memory in a trusted third party (realized as a hardware-protected module), and all reads and writes become explicit RPCs to this third party. Under the hood, the third party relies on dynamic sealing to hide memory addresses [31]. This effectively reduces memory sharing to message passing and elides most of the complications in proofs with true memory sharing, but also results in extremely inefficient code that requires heavyweight calls at every read/write to shared memory, thus largely defeating the purpose of sharing memory in the first place.

Secure linking To ensure safe interaction with low-level code, typed assembly languages [19] and multi-language semantics [29] have been used by Patterson et al. [42]. Their technique restricts the low-level language not with runtime enforcement of memory isolation like in some architectures [17, 50, 55, 56, 57, 58] and in our Mach model, but instead with a static type system. The type system and the accompanying logical relation allow reasoning about the equivalence of a "mixed-language" setting, which is what we called **Setting 2** in §2.1, but sometimes requires exposing low-level abstractions to high-level code. The secure compilation approach we follow has a chance of avoiding that. For example, by avoiding the need for directly reasoning in **Setting 2**, our secure compilation result beneficially hides from the programmer the fact that a Mach function can jump to non-entry points of other functions in the same component.

Robust safety preservation Robust safety preservation (RSP), the secure compilation criterion we use, was first described by Abate et al. [3, 4], Patrignani and Garg [38]. However, this initial work uses a trivial relation (equality) between source and target traces. With a general relation, as in our setting, RSP was first examined by Abate et al. [5]. RSP further traces lineage to the robust (context-independent) verification of safety properties of a given program (not a given compiler), which is often called "robust satisfaction of safety properties" [25]. Robust satisfaction is a well-developed concept, used in model checking [21], type systems [20], and program logics [22, 43, 52].

Secure compilation of information-flow guarantees A long line of work [8, 9, 10, 11, 12, 34, 44, 45] develops proof techniques and verified compilers to ensure that information flow properties like non-interference, the constant-time policy, or side-channel resistance are preserved by compilation. These techniques, however, are all concerned with whole-programs, unlike our work that starts with the premise that partial programs will interact with untrusted code.

7 Conclusion and Future Work

In this paper we introduced data-flow back-translation and turn-taking simulation, two new techniques for scaling secure compilation proofs to both syntactic dissimilarity between the source and target languages and dynamic memory sharing. In the future, we would like to apply these techniques to more realistic compilers and enforcement mechanisms, for instance based on capability machines [55, 58] or programmable tagged architectures [14, 17]. We also think our techniques can be extended to stronger secure compilation criteria, building on work by Abate et al. [4], who illustrate that the robust preservation of a large class of *relational* safety properties can be proved by trace-based back-translation. This is stronger than both RSP and a variant of full abstraction, but their multi-trace back-translation technique does not yet cope with mutable state.

Acknowledgements This work was in part supported by the European Research Council under ERC Starting Grant SECOMP (715753), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972.

Appendix

A Expressing nowrite using traces

The safety property **nowrite** can be defined formally as a predicate on traces (Definition 2.6) as follows.

Example A.1 (The safety spec **nowrite**). Suppose $l_{balance}$ is the memory location allocated for the variable user_balance_usd, and suppose c_{main} denotes the component that calls the function set_ads_image, which is implemented by c_{setter}.

$$\textbf{nowrite}(t) \overset{\text{def}}{=}$$
$$\forall t_1\ e_{call}\ t_2\ Mem_1\ Mem_2\ l.$$

$$t = t_1 \text{ ++ } [e_{call}] \text{ ++ } t_2 \implies$$
$$e_{call} = \texttt{Call}\ Mem_1\ c_{main}\ c_{setter}.set\_ads\_image() \implies$$
$$\texttt{find\_matching\_ret}(t_2, e_{call}) =$$
$$\quad \texttt{Ret}\ Mem_2\ c_{setter}\ c_{main}.void \implies$$
$$Mem_1(l_{balance}) = Mem_2(l_{balance})$$

The spec above makes sure that the value of the variable user_balance_usd before the call to the function set_ads_image is the same as its value at the point when the function returns.

B Proof of Recomposition for Mach

We use the turn-taking memory relation from §3.2 to prove recomposition (Lemma 2.4). To do that, we prove that Definition 3.9 of `mem_rel_tt` is an invariant. Definition 3.9 is part of a bigger invariant `state_rel_tt` on execution states that we elide here for space reasons. The Coq proof of Lemma 2.4 is, however, available in `Intermediate/RecombinationRel.v`, which in turn uses all of `RecombinationRelCommon.v`, `RecombinatioRelOptionSim.v`, `RecombinationRelLockStepSim.v` and `RecombinationRelStrengthening.v`[9].

A key insight of this proof is the need for a strengthening lemma that recovers a stronger invariant, `state_rel_border`, that holds at states that emit interaction events We show the memory part of `state_rel_border`:

Definition A.2 (Memory Relation At Interaction Events).

$$\texttt{mem\_rel\_border}(s_{1,2}, s_1, s_2, t_{1,2}, t_1, t_2) \overset{\text{def}}{=}$$
$$\texttt{mem\_rel\_exec}(P_1, t_1, t_{1,2}, s_1.Mem, s_{1,2}.Mem) \wedge$$
$$\texttt{mem\_rel\_exec}(C_2, t_2, t_{1,2}, s_2.Mem, s_{1,2}.Mem)$$

where `mem_rel_exec` *is exactly as in Definition 3.9.*

Among other things, `mem_rel_border` ensures that the shared memories of the three states (of the recomposed program and the two base programs) are all in sync. We are able to this strong invariant *only* at interaction events, because at these points we can use the assumption that the traces of the two base programs are related (last assumption of Lemma A.3), which implies that the shared memories of the base programs are related. This assumption can be combined with `mem_rel_tt` (which holds universally for every triple of corresponding states) to obtain `mem_rel_border`.

Lemma A.3 (Strengthening at interaction events).

$$\forall s_{1,2}\ s_1\ s_2\ t_{1,2}\ t_1\ t_2\ s_1'\ s_2'\ e_1\ e_2.$$
$$\texttt{state\_rel\_tt}(s_{1,2},\ s_1,\ s_2,\ t_{1,2},\ t_1,\ t_2) \implies$$
$$s_1 \xrightarrow{[e_1]} s_1' \implies$$
$$s_2 \xrightarrow{[e_2]} s_2' \implies$$
$$t_1 \text{ ++ } [e_1] \sim t_2 \text{ ++ } [e_2] \implies$$
$$\exists s_{1,2}'\ e_{1,2}.\ s_{1,2} \xrightarrow{[e_{1,2}]} s_{1,2}' \wedge$$
$$\texttt{state\_rel\_border}(s_{1,2}',\ s_1',\ s_2',$$
$$t_{1,2} \text{ ++ } [e_{1,2}],\ t_1 \text{ ++ } [e_1],\ t_2 \text{ ++ } [e_2])$$

[9]for a total of 850 lines of specification and 12.6k lines of proof

The relation `state_rel_tt` is a turn-taking simulation invariant. It ensures that the memory relation `mem_rel_tt` holds of the memories of the three related states. Similarly, the stronger state relation `state_rel_border` ensures that the memory relation `mem_rel_border` holds of the memories of the three related states.

The exact definition of the relation `state_rel_tt` is in `RecombinationRelCommon.v`. We show here two key lemmas:

Lemma A.4 (Option simulation w.r.t. the *non*-executing part).

$$\forall s_{1,2}\ s_1\ s_2\ t_{1,2}\ t_1\ t_2\ s_1'.$$
$$s_{1,2}\ \textit{is executing in}\ C_2\ (i.e., \text{not in } P_1) \implies$$
$$\texttt{state\_rel\_tt}(s_{1,2},\ s_1,\ s_2,\ t_{1,2},\ t_1,\ t_2) \implies$$
$$s_1 \xrightarrow{[]}{}^{*}\ s_1' \implies$$
$$\texttt{state\_rel\_tt}(s_{1,2},\ s_1',\ s_2,\ t_{1,2},\ t_1,\ t_2)$$

The last assumption ($s_1 \xrightarrow{[]}{}^{*} s_1'$) of the option simulation (Lemma A.4) says that state s_1 of the base program $P_1 \cup C_1$ takes some non-interaction steps. This base program contributes just P_1 to the recomposed program ($P_1 \cup C_2$), and we know by assumption "$s_{1,2}$ `is executing in` C_2" that the recomposed state $s_{1,2}$ is *not* executing in P_1. The invariant `state_rel_tt` ensures that $s_{1,2}$ executes in P_1 whenever s_1 executes in P_1. Thus, the steps that s_1 has made must be taken by the discarded part C_1, not the retained part P_1. As shown in Example 3.7, we know that steps taken by C_1 can cause a mismatch between the memory of the recomposed program and the memory of the base program $P_1 \cup C_1$. The option simulation lemma ensures that this mismatch is tolerated by the `state_rel_tt` invariant.

Lemma A.5 (Lock-step simulation w.r.t. the executing part).

$$\forall s_{1,2}\ s_1\ s_2\ t_{1,2}\ t_1\ t_2\ s_1'.$$
$$s_{1,2}\ \textit{is executing in}\ C_2\ (i.e., \text{not in } P_1) \implies$$
$$\texttt{state\_rel\_tt}(s_{1,2},\ s_1,\ s_2,\ t_{1,2},\ t_1,\ t_2) \implies$$
$$s_2 \xrightarrow{[]} s_2' \implies$$
$$\exists s_{1,2}'.\ s_{1,2} \xrightarrow{[]} s_{1,2}' \wedge$$
$$\texttt{state\_rel\_tt}(s_{1,2}',\ s_1,\ s_2',\ t_{1,2},\ t_1,\ t_2)$$

Lock-step simulation (Lemma A.5) ensures that the invariant `state_rel_tt` is strong enough to keep every non-interaction step of a retained part in sync between the recomposed program and the corresponding base program.

Although both Lemmas A.4 and A.5 hold only for the scenario when "$s_{1,2}$ `is executing in` C_2 (i.e., *not in* P_1)", they are still general enough because we can apply symmetry lemmas to our invariant `state_rel_tt` to reduce the other scenario "$\mathtt{s}_{1,2}$ `is executing in` $\mathbf{P}_1$" to the former scenario—thus avoiding lots of duplicate proof. The symmetry lemmas are proved in `RecombinationRelCommon.v`. In `RecombinationRel.v`, the reader can find the top-level proof of recomposition (Lemma 2.4) that uses these symmetry lemmas in addition to strengthening (Lemma A.3), option simulation (Lemma A.4), and lock-step simulation (Lemma A.5).

To summarize, the new idea of turn-taking simulations helped us complete the recomposition proof with memory sharing. Although we have elided many low-level details of the recomposition proof here, they are fully resolved in our Coq mechanization.

References

[1] M. Abadi. Protection in programming-language translations. In *International Colloquium on Automata, Languages, and Programming*. Springer, 1998.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[3] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hritcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018.

[4] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, 2019.

[5] C. Abate, R. Blanco, Ş. Ciobâcă, A. Durier, D. Garg, C. Hritcu, M. Patrignani, É. Tanter, and J. Thibault. Trace-relating compiler correctness and secure compilation. In P. Müller, editor, *29th European Symposium on Programming, ESOP*. 2020.

[6] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. *SIGPLAN Not.*, 43 (9), 2008.

[7] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not.*, 46(9), 2011.

[8] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*. 2004.

[9] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018.

[10] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving c compiler. *Proc. ACM Program. Lang.*, 4(POPL), 2019.

[11] G. Barthe, S. Blazy, R. Hutin, and D. Pichardie. Secure compilation of constant-resource programs. In *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 2021.

[12] F. Besson, A. Dang, and T. Jensen. Securing compilation against memory probing. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*. ACM, 2018.

[13] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. IEEE, 2020.

[14] A. A. De Amorim, M. Dénes, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015.

[15] D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016.

[16] D. Devriese, M. Patrignani, F. Piessens, and S. Keuchel. Modular, fully-abstract compilation by approximate back-translation. *Log. Methods Comput. Sci.*, 13(4), 2017.

[17] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. *SIGARCH Comput. Archit. News*, 43(1), 2015.

[18] A. El-Korashy, S. Tsampas, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. Capableptrs: Securely compiling partial programs using the pointers-as-capabilities principle. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, 2021.

[19] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1999.

[20] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols 1. *Journal of computer security*, 11 (4), 2003.

[21] O. Grumberg and D. E. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*. Springer, 1991.

[22] L. Jia, S. Sen, D. Garg, and A. Datta. A logic of programs with interface-confined code. In *2015 IEEE 28th Computer Security Foundations Symposium*, 2015.

[23] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. *SIG-*

PLAN Not., 51(1), 2016.

[24] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. *POPL*. 2014.

[25] O. Kupferman and M. Y. Vardi. Robust satisfaction. In J. C. M. Baeten and S. Mauw, editors, *CONCUR'99 Concurrency Theory*. 1999.

[26] J. Laird. *A fully abstract trace semantics for general references*, pages 667–679. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer Verlag, 2007.

[27] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4), 2009.

[28] X. Leroy and S. Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1), 2008.

[29] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2007.

[30] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7(3), 1972.

[31] J. H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16(1), 1973.

[32] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures: An application of decompilation into logic. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. 2008.

[33] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009.

[34] K. S. Namjoshi and L. M. Tabajara. Witnessing secure compilation. In D. Beyer and D. Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation*. 2020.

[35] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2015.

[36] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016.

[37] M. Patrignani and D. Garg. Robustly safe compilation. In L. Caires, editor, *28th European Symposium on Programming, ESOP*. 2019.

[38] M. Patrignani and D. Garg. Robustly safe compilation, an efficient form of secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(1), 2021.

[39] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2), 2015.

[40] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51 (6), 2019.

[41] D. Patterson and A. Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP), 2019.

[42] D. Patterson, J. Perconti, C. Dimoulas, and A. Ahmed. Funtal: Reasonably mixing a functional language with assembly. *SIGPLAN Not.*, 52(6), 2017.

[43] M. Sammler, D. Garg, D. Dreyer, and T. Litak. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.*, 4(POPL), 2019.

[44] R. Sison and T. Murray. Verifying That a Compiler Preserves Concurrent Value-Dependent Information-Flow Security. In J. Harrison, J. O'Leary, and A. Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*. 2019.

[45] R. SISON and T. MURRAY. Verified secure compilation for mixed-sensitivity concurrent programs. *Journal of Functional Programming*, 31, 2021.

[46] L. Skorstengaard, D. Devriese, and L. Birkedal. Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL), 2019.

[47] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C.-K. Hur. Compcertm: Compcert with c-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.*, 4(POPL), 2019.

[48] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional compcert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2015.

[49] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional compcert. *SIGPLAN Not.*, 50(1), 2015.

[50] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[51] T. V. Strydonck, F. Piessens, and D. Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, 3(ICFP), 2019.

[52] D. Swasey, D. Garg, and D. Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017.

[53] S. Tsampas, D. Devriese, and F. Piessens. Temporal safety for stack allocated memory on capability machines. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019.

[54] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP*, 1993.

[55] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka,

B. Laurie, et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015.

[56] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5), 2016.

[57] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, N. W. Filardo, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2019.

[58] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. *SIGARCH Comput. Archit. News*, 42(3), 2014.

[59] F. Zhang and E. D'Hollander. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering*, 30(4), 2004.