CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle

Akram El-Korashy, MPI-SWS
Stelios Tsampas, KU Leuven
Marco Patrignani, CISPA
Dominique Devriese, Vrije Universiteit Brussel
Deepak Garg, MPI-SWS
Frank Piessens, KU Leuven

***Abstract*—Capability machines such as $CHERI$ provide memory capabilities that can be used by compilers to provide security benefits for compiled code (e.g., memory safety). The existing C to $CHERI$ compiler, for example, achieves memory safety by following a principle called “pointers as capabilities” (PAC). Informally, PAC says that a compiler should represent a source language pointer as a machine code capability. But the security properties of PAC compilers are not yet well understood. We show that memory safety is only one aspect, and that PAC compilers can provide significant additional security guarantees for *partial programs*: the compiler can provide security guarantees for a compilation unit, even if that compilation unit is later linked to attacker-provided machine code.**

As such, this paper is the first to study the security of PAC compilers for partial programs formally. We prove for a model of such a compiler that it is *fully abstract*. The proof uses a novel proof technique (dubbed $TrICL$, read *trickle*), which should be of broad interest because it reuses the whole-program compiler correctness relation for full abstraction, thus saving work. We also implement our scheme for C on $CHERI$, show that we can compile legacy C code with minimal changes, and show that the performance overhead of compiled code is roughly proportional to the number of cross-compilation-unit function calls.

***Index Terms*—Capability machines, full abstraction, secure compilation**

I. Introduction

In a conventional computer, memory is addressed using integers (*pointers*). In a capability machine, memory is addressed using *capabilities* [1, 2, 3, 4, 5, 6, 7], which carry more information than just a memory address—they also contain *bounds information*, indicating a range of memory that can be accessed using the capability, and possibly also other information such as access permissions. Load and store instructions take a capability (in a register), and the machine checks that the memory accessed is within the capability's bounds and that the operation is compliant with the capability's permissions. If not, the instruction fails with an exception. The hardware also ensures that integers and capabilities are not confused. One way of ensuring this is by tracking capabilities in memory and in registers by *tagging* memory locations and registers that contain capabilities. Hence, capability machines implement a more structured memory model where (somewhat simplified) memory is a collection of independent integer-indexed arrays containing integers or capabilities, and every capability gives access to a contiguous segment of one of those arrays. This more structured memory model can be used to implement *fine-grained memory protection*, and has the potential to provide protection against many software bugs.

A recent capability machine is $CHERI$ [4]. It has its own FreeBSD version and C compiler [8, 4, 9, 10, 11]. Many key design choices in $CHERI$ were made to facilitate the use of memory protection in existing, large code bases. Specifically, $CHERI$ supports the pointers-as-capabilities (PAC) principle, which intuitively dictates that a compiler should represent a source-level *pointer* as a target-level *capability*. To make this convenient, a $CHERI$ capability contains (among other things) `base` and `length` addresses, and an `offset` relative to the base address [11]. Such a capability represents a pointer pointing to the address `base`+`offset`, and that is valid only if (`base`+`offset`) $\in$ [`base`, `base`+`length`). Pointer arithmetic can be implemented by manipulating the offset. The following example illustrates how a compiler can map C pointers to such machine-level capabilities.[1]

```
extern void send_rcv(char* buffer);
static char iobuffer [512];
static int secret;

void f() {
  iobuffer[42] = 'X';
  send_rcv(iobuffer);
}
```

```
  csl $c1, $ddc, 512;
  li $r1, 'X';
  csw $r1, 42($c1);
  call send_rcv;
```

The C compilation unit (top) declares two module-scoped variables and defines a function `f()` using one of these variables. The assembly pseudocode (bottom) shows how a PAC compiler could translate the body of `f()`. The *default data capability* register **`$ddc`** contains a capability for the global data section. The compiler knows that the variable `iobuffer` occupies the first 512 bytes of that global data section. Hence, the first instruction (**`csl`**, set length of a capability) loads in register **`$c1`** a copy of **`$ddc`** but with the **`length`** field reduced to 512. The next two instructions implement the assignment instruction of `f()`. Note that an out-of-bounds access would be trapped by the hardware. The final **`call`** instruction implements the function call in `f()` (assuming a calling convention where the argument is passed in register **`$c1`**). All accesses to `iobuffer` performed in `send_rcv` will be bounds-checked, since the capability passed to `send_rcv` carries the bounds information.

[1] As a typesetting convention, we use a blue, sans-serif font for source language elements and an **orange**, **bold** font for **target** language ones. Elements common to both languages are typeset in a *black*, *italic* font.

Which security benefits does such a PAC compiler provide? First, the compiler provides *spatial memory safety*. Since the bounds meta-data for a pointer is stored together with the pointer address in a single capability value, it is natural to implement a bounds-checking compiler [12, Section 4.3]. For instance, an out-of-bounds access to `iobuffer` in our example will not access the `secret` variable, but just fail.[2]

However, this is not the full story regarding security properties. Consider the example again under the assumption that the external `send_rcv` function is implemented directly in assembly. Now, we lose the guarantee that `send_rcv` cannot access the `secret` variable because an assembly level implementation can directly access `$ddc`. Hence, upfront, memory safety is only guaranteed for *complete programs*: if *all* code in a program is compiled by the PAC compiler, then all out-of-bounds accesses will be trapped.

Nonetheless, a PAC compiler can provide a security guarantee even for *partial programs* by relying on capability unforgeability [19] and on a trusted control stack (that hides the capability of a program part from the context). The main contribution of this paper is to prove that a PAC compiler can, in fact, provide strong security guarantees for partial programs. For our example, the compiler we model in this paper provides the guarantee that `secret` is inaccessible, even if the function `send_rcv` is implemented in hand-crafted assembly.

To achieve non-trivial security guarantees for partial programs, the target capability machine needs to support a mechanism to define separate protection domains within a single process. For instance, $CHERI$ provides support for so-called *object capabilities* [10, 9]. This makes it possible to put different program parts in separate protection domains. $CHERI$ mainly uses object capabilities to *compartmentalize* programs: it offers an API to programmers to run parts of a program in a *sandbox*, a protection domain with reduced privileges. The current $CHERI$ compiler, however, does not make direct use of the object capability mechanism: it can only be used through the provided API by the *programmer* who has to define and set up sandboxes manually. The PAC compiler we propose in this paper *automatically* sets up a separate protection domain for every compilation unit. Doing so allows the compiler to provide strong security guarantees for partial programs.

Summary of our results: We study the security guarantees that a PAC compiler can provide for *partial programs*. Our setting is a PAC compiler from a simple imperative source language with pointers to a capability machine with memory capabilities and a very basic form of protection domains/object capabilities. Our overarching contribution is a very strong security theorem for this compiler, namely, *full abstraction (FA)* [20, 21], which intuitively means that the compilation preserves and reflects observational equivalence of partial programs. FA implies the preservation of many security properties like data confidentiality, even when target contexts are arbitrary target code (in our case, arbitrary assembly code) that may *not* respect the compiler's conventions.

In proving FA for PAC, we make two additional contributions. Our first contribution is a new trace-based proof technique for FA that can simultaneously handle *dynamic memory sharing* between modules and, importantly, *reuses whole-program compiler correctness as a black-box* to simplify the FA proof. FA proofs with dynamic memory sharing are difficult and (whole-program) compiler correctness is usually proved anyhow, so reusing it for proving FA reduces work. Technically, our proof is structured as a new 3-way simulation called $TrICL$ (read "trickle"). We expect $TrICL$ to be of interest beyond our PAC setting.

Second, to prove FA, we find it essential to reflect some structure of capabilities at the source-code level, forcing the programmer to take into account some of the machine-code-level expressiveness when reasoning about the source program. Interestingly, not doing this may also lead to subtle security vulnerabilities as the following example illustrates.

```
static bool secret=true;

int branch_on_secret(int* p, int* q) {
  if ((int) p != (int) q) return 0;
  if (secret) return p[1]; else return q[1];
}
```

Function `branch_on_secret()` first tests whether its two argument pointers are equal addresses. This equality implies that the dereference operations `p[1]` and `q[1]` both evaluate to the same value (or both fail). Hence, `branch_on_secret()` is not really meant to leak any information about `secret`. However, a machine-code level adversary can call `branch_on_secret()` with two capabilities that both point to the same address but that have different bounds information. In that case, accessing `p[1]` could fail, while accessing `q[1]` could succeed and return a value. Thus, the behavior of `branch_on_secret()` in that case *does* leak information about `secret`. More dangerously, it leaks this information in a way to which the source-level programmer is oblivious. The source-level programmer does not have a way to access bounds information through pointer operations, while this bounds information actually opens up an information channel. Hence, to prove our FA result (which includes the preservation of source-level contextual equivalence), we extend the source language to make pointers carry bounds information. This essentially makes explicit exactly what aspects of the target language programmers need to take into account to reason about the security of partial source programs.

A second property we require for FA is that machine code does not have direct access to the program counter capability. This is easily checked at link time. It guarantees that the target context cannot confuse a partial program by providing it a code capability where it expects a data capability (a behavior that does not exist in our source language).

Finally, as our last contribution, we implement our compiler for C by adding a "compartmentalizing compiler-pass"

[2]In principle, it is also possible to have the compiler provide *temporal memory safety*, but this is harder as it requires zeroing out all capabilities to a memory region when the region is freed. Efficient temporal memory safety for C is still an open problem and out of scope here. We refer the interested reader to prior work [13, 14, 15, 16, 17, 18].

on top of the existing C-to-$CHERI$ compiler (the non-compartmentalizing PAC compiler) [10, 12], and we evaluate the performance cost as well as the compatibility of the compartmentalizing PAC compiler with existing code.

Contributions summary: To summarize, we make the following contributions:

- We state and prove the security properties of a pointers-as-capabilities (PAC) compiler for partial programs for the first time. In doing so, we make several technical contributions:
  - the definition of a sound and complete trace semantics for a C-like language (Section III-A) and for a language with capabilities (Section III-B). Both languages feature a memory model that allows fine-grained dynamic memory sharing;
  - the definition of a compiler between the aforementioned languages that embodies the PAC principle (Section IV);
  - a proof that the said compiler is fully abstract, with a new trace-based technique, $TrICL$, that handles dynamic memory sharing and allows reuse of whole-program compiler correctness for *FA* (Section V);
- An implementation of our compiler on top of the existing C-to-$CHERI$ compiler and a measurement of its efficiency and compatibility with existing C code (Sections VI and VII).

We have simplified some aspects of the technical work for presentation. The full details as well as proofs are provided in a technical report (TR), available in a public repository [22]. The implementation of our compiler and the related benchmarks are also available in the same repository.

II. Full abstraction and a new proof technique

We briefly recap compiler full abstraction (*FA*), outline at a high-level how it is proved, explain why a new proof technique is needed and what our new technique ($TrICL$) does.

A. Preliminaries

Given execution states s, s' of a language, and a small-step reduction relation $\to$ (with a reflexive transitive closure $\to^*$), we denote by $s \to s'$ the judgment that state s *executes* and transitions into state s'. We call a state *stuck* if there is no s' such that $s \to s'$. We treat exceptions and silent divergence the same way we treat stuck states, so, with slight abuse of terminology, we use the term *diverging* for executions that get stuck, silently reduce forever or end in exceptions.

Programs, initial & terminal states: In our simple imperative setting, a (partial) program is a list of modules, and a module is a list of functions. A program is called *whole* if its functions refer only to other functions also defined within the program. Otherwise, it is called partial. Linking of a pair of programs is denoted $\ltimes$. We define linking noncommutatively because this simplifies some technicalities (see the TR for details). For the sake of exposition, we distinguish two parts $\mathfrak{C}$ and p of a linked program $\mathfrak{C} \ltimes p$ as the *context* and the *program*, suggesting that the latter is the *program part of interest* because it is the program that is translated by our compiler. Note that each of $\mathfrak{C}$ and p may themselves consist of more than one *module* (i.e., more than one compilation unit). As usual, only whole programs with a *main* entry-point function can execute. We also use the standard notation $\mathfrak{C}[p]$ for $\mathfrak{C} \ltimes p$.

The initial state of a program p is denoted $init(p)$. A state s is called *terminal* when it satisfies a special judgment $\vdash_t s$. If the execution of a program of interest p in a certain context $\mathfrak{C}$ reaches a terminal state, then we say the execution *converges* (or, that the *program* converges). We denote convergence by $\mathfrak{C}[p] \Downarrow$, which is shorthand for $\exists s.\ init(\mathfrak{C}[p]) \to^* s \ \wedge\ \vdash_t s$. Next, we define contextual equivalence of (partial) programs.

Definition 1 (Contextual equivalence)**.**

$$p_1 \simeq_{\text{ctx}} p_2 \overset{\text{def}}{=} \forall \mathfrak{C}.\ \mathfrak{C}[p_1] \Downarrow \iff \mathfrak{C}[p_2] \Downarrow$$

B. Compiler full abstraction

A compiler $[\![\cdot]\!]$ is fully abstract when it preserves and reflects contextual equivalence. The use of full abstraction to establish compiler security is standard [20, 23, 21].

Definition 2 (Full abstraction)**.** *The compiler $[\![\cdot]\!]$ is FA if for all $\mathsf{p_{s1}}, \mathsf{p_{s2}}$:*

(i) (Reflection) $[\![\mathsf{p_{s1}}]\!] \simeq_{\mathbf{ctx}} [\![\mathsf{p_{s2}}]\!] \implies \mathsf{p_{s1}} \simeq_{\text{ctx}} \mathsf{p_{s2}}$

(ii) (Preservation) $[\![\mathsf{p_{s1}}]\!] \simeq_{\mathbf{ctx}} [\![\mathsf{p_{s2}}]\!] \impliedby \mathsf{p_{s1}} \simeq_{\text{ctx}} \mathsf{p_{s2}}$

Condition (i) ensures that the compiler is non-trivial (a trivial compiler might compile semantically different programs to the same output program, which is forbidden by reflection). Reflection usually follows immediately from *backward simulation*, the standard formalization of the compiler's whole-program correctness [24, 25].

Condition (ii) is the "security-relevant" direction of *FA* as it ensures that no extra distinguishing power is gained by target contexts as compared to source contexts. It implies the preservation of any security property that can be formalized as program equivalence (a notable example being noninterference for confidentiality). It is usually proved in the contrapositive: Assume $[\![\mathsf{p_{s1}}]\!] \not\simeq_{\mathbf{ctx}} [\![\mathsf{p_{s2}}]\!]$, and show $\mathsf{p_{s1}} \not\simeq_{\text{ctx}} \mathsf{p_{s2}}$. From the assumption, there must be a target context that distinguishes $[\![\mathsf{p_{s1}}]\!]$ and $[\![\mathsf{p_{s2}}]\!]$ (causes one to diverge and the other to converge). From this, we need to construct a source context that distinguishes $\mathsf{p_{s1}}$ and $\mathsf{p_{s2}}$. This construction of the source context is called *back-translation* in the literature. There are two broad approaches to back-translation: *syntax-directed* and *trace-directed* [21]. Here, we follow the trace-directed approach as we find it technically more convenient (we discuss syntax-directed approaches in Section VIII).

The key idea of the trace-directed approach is to characterize the *observable* behavior of partial programs via a *labeled* transition system that produces *(finite) traces* describing how a partial program interacts with its environment.[3] This is done separately for source and target languages. We denote the set of traces of a partial program p by $Tr(p)$. Next, define *trace equivalence* $\overset{\text{T}}{=}$ of partial programs in source

[3] For the purpose of proving *FA*, finite traces suffice, so "trace" in this paper always refers to a finite trace.

and target languages separately: Two partial programs (both source or both target) p_1, p_2 are trace equivalent, $p_1 \stackrel{\mathrm{T}}{=} p_2$, if $Tr(p_1) = Tr(p_2)$. We then prove three lemmas.

Lemma 1 (Soundness of target trace equivalence)**.** $[\![\mathsf{p_{s1}}]\!] \not\simeq_{\mathbf{ctx}} [\![\mathsf{p_{s2}}]\!] \implies [\![\mathsf{p_{s1}}]\!] \stackrel{\mathbf{T}}{\neq} [\![\mathsf{p_{s2}}]\!]$

Lemma 2 (Compilation preserves trace equivalence)**.** $[\![\mathsf{p_{s1}}]\!] \stackrel{\mathbf{T}}{\neq} [\![\mathsf{p_{s2}}]\!] \implies \mathsf{p_{s1}} \stackrel{\mathsf{T}}{\neq} \mathsf{p_{s2}}$

Lemma 3 (Completeness of source trace-equivalence)**.** $\mathsf{p_{s1}} \stackrel{\mathsf{T}}{\neq} \mathsf{p_{s2}} \implies \mathsf{p_{s1}} \not\simeq_{\mathsf{ctx}} \mathsf{p_{s2}}$

The composition of these three lemmas immediately yields our goal, the contrapositive of condition (ii)! The important point is that only Lemma 2 bridges the two languages[4]. Lemma 2 follows immediately from the following two lemmas, which together say that the compiler preserves and reflects traces of partial programs.

Lemma 4 (No trace is omitted by compilation)**.** $\alpha \in \mathsf{Tr}(\mathsf{p_s}) \implies \alpha \in \mathbf{Tr}([\![\mathsf{p_s}]\!])$

Lemma 5 (No trace is added by compilation)**.** $\alpha \in \mathsf{Tr}(\mathsf{p_s}) \impliedby \alpha \in \mathbf{Tr}([\![\mathsf{p_s}]\!])$

Lemma 4 follows directly from compiler (forward) simulation, which is needed to prove the compiler correct anyhow. Hence, this lemma does not add additional proof effort. On the other hand, Lemma 5 is an additional (and the last remaining!) proof burden. The "difficulty" of this proof really depends on the complexity of the traces, i.e., the complexity of interaction between a program and its context.

C. The Why and What of TrICL

In prior work that uses trace-based back-translation, program modules cannot share memory or the shared memory is fixed upfront [26, 27, 21]. Traces are easy to define in this setting. In our *PAC* setting, however, the memory shared between modules can grow dynamically as the program shares more of its previously private memory with the context by passing it corresponding capabilities (or pointers in the source). The context can also pass capabilities back to the program but it *should* only pass back those capabilities that it started with or those that it received previously during the execution (since capabilities cannot be forged by design). Consequently, any trace on which the context passes back a capability that it didn't receive earlier should be removed from consideration as invalid. However, this notion of trace (in)validity is not straightforward (in either language) because capabilities can be passed not just directly via function arguments, but also *indirectly*, as something that's reachable in the heap at the point of a transition. This makes valid traces on *partial* programs rather hard to define. To avoid this problem, we use a different definition of traces.

[4]Although Lemma 1 mentions the compiler (i.e., appears to also bridge the two languages), we found that its proof does not rely on a cross-language simulation, which is what we really mean by bridging the two languages.

A new definition of traces: We start from *whole* programs, not partial programs. The trace is defined by decorating the small-step semantics of the whole program with information about interaction between its modules. We then define the traces of a partial program as the traces it can produce when linked with *some (existentially quantified) context*. The explicit presence of the context automatically ensures that all traces are valid, so we don't have to define validity separately. (In other words, validity of a trace now follows implicitly from easier-to-define invariants on the whole execution state.)

Definition 3 (Traces of a partial program)**.**

$$Tr(p) \stackrel{\mathsf{def}}{=} \{\alpha \mid \exists \mathfrak{C}, s, \varsigma.\, (init(\mathfrak{C}[p]), \emptyset) \xrightarrow{\alpha}_p (s, \varsigma)\}$$

$\emptyset$ and ς in this definition are auxiliary state components, that we describe in Section V. The reader can ignore them for now. $\xrightarrow{\alpha}_p$ represents label-decorated whole-program reduction, which we also define in Section V. To prove Lemma 5 with this definition of traces, we must show that, given a trace α of $[\![\mathsf{p_s}]\!]$, there is an *emulating* source context $\mathfrak{C}_{\mathsf{emu}}$ such that $\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]$ produces α. In other words, we must *back-translate* a trace α into a source context $\mathfrak{C}_{\mathsf{emu}}$ and show that $\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]$ can actually produce α. For this, we would like to set up a simulation between the target and source runs. However, this simulation can be very hard to set up because we constructed the emulating source context without considering the given target context (just from the trace prefix) and, hence, there can be differences between the *internal* behaviors of the emulating context and the given target context, e.g., in the specific order of updates to the shared memory, and in the internal function calls. These differences mean that the memory (and the call stack) do not remain in sync between the emulating source context and the given target context and our simulation relation needs to accommodate the big gap between the internal states of the target contexts and the emulating source contexts. We call this the "vertical gap".

TrICL: This is where our new *TrICL* technique comes in. *TrICL* introduces a third "mediator" run to the simulation, namely, that induced by the *compilation of the whole source program* containing both $\mathsf{p_s}$ and $\mathfrak{C}_{\mathsf{emu}}$. Overall, the three runs are (i) the given (target) run of $\mathfrak{C}[[\![\mathsf{p_s}]\!]]$, where $\mathfrak{C}$ is the target context that induces the trace α, (ii) the emulating source run of $\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]$, and (iii) the mediating target run of $[\![\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]]\!]$.

Introducing the mediator run simplifies the proof as follows. Because the mediator $[\![\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]]\!]$ is obtained by a whole program compilation of the emulator $\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]$, we can *reuse whole-program compiler correctness as a black box* to immediately reduce the problem of showing that the emulating (source) run emulates the given (target) run to that of showing that the mediating (target) run emulates the given (target) run. This reduced problem is now about *two runs in the same language* (target), which enables us to write a big part of our simulation invariants as same-language (instead of cross-language) invariants. In other words, this simplifies the "vertical gap".

```
1  Module id: Main
2  Import module: Networking
3
4  iobuffer [512];
5  secret;
6
7  main() {
8    Assign &iobuffer[42] 4242;
9    Call read_secret();
10   Call encrypt();
11   Call send_rcv(&iobuffer);
12   Call decrypt();
13 }
14
15 read_secret() { ... }
16 encrypt() { ... }
17 decrypt() { ... }
```

Listing III.1: ImpMod module.

Section V explains the *TrICL* simulation for our *PAC* compiler, but we note that the idea is general and should apply to other settings as well.

III. Source and target languages

Next, we introduce our source and target languages, ImpMod (Section III-A) and **CHERIExp** (Section III-B).

A. ImpMod: The source language

To keep our focus limited to the "pointers as capabilities" aspect of the translation, we design ImpMod and **CHERIExp** to be pretty close except in how they deal with memory. For example, ImpMod features only unstructured control flow in the form of a JumpIfZero instruction. But it still has functions and modules. In fact, *modules* are crucial. They are units of memory isolation, which makes programs interesting for security. Briefly, 1) every module gets its own *module-global variables*; every function inside the module can access these variables, whereas any function external to the module cannot access these variables directly by default, and 2) every module gets its own *data stack* on which it stores the frames of live (ongoing) calls to its functions.

For example, Listing III.1, shows a module (`Main`) with two module-global variables, `iobuffer` and `secret` (lines 4 & 5). The `main` function[5] and the secret-handling functions `read_secret`, `encrypt` and `decrypt` are defined in the same module, and thus can access the variables `iobuffer` and `secret`. The function `send_rcv`, in contrast, is external: it is defined in the `Networking` module which is presumably untrusted. On line 11, the `Networking` module gains access to `iobuffer`. The `Networking` module, however, does not gain access to `secret` through `&iobuffer`—so long as the other trusted functions `read_secret` and `encrypt` also make sure to not copy (pointers to) `secret` into the `iobuffer`. Any attempt to increment and access the pointer `&iobuffer` beyond the array bounds gets stuck. To understand how we model these bounds checks, we introduce the expression semantics and the memory model of ImpMod.

[5]Notice in the `main` function on line 8 the use of variable `iobuffer`; the syntax for l-values is more explicit than in C.

1) Expressions and memory model of ImpMod*:* Expressions are denoted e and do not update memory.

$$\mathsf{e} ::= \mathbb{Z} \mid \mathsf{VarID} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{e}[\mathsf{e}] \mid \&\mathsf{VarID} \mid \&\mathsf{e}[\mathsf{e}] \mid {}^{*}(\mathsf{e}) \mid \mathsf{start}(\mathsf{e}) \mid \mathsf{end}(\mathsf{e}) \mid \mathsf{offset}(\mathsf{e}) \mid \mathsf{capType}(\mathsf{e}) \mid \mathsf{limRange}(\mathsf{e},\mathsf{e},\mathsf{e})$$

$$\mathsf{V} ::= \mathbb{Z} \mid \mathsf{Cap}$$

Base expressions are integers $\mathbb{Z}$ and variable identifiers VarID (both local and global). *Binary arithmetic expressions* are generically denoted $\mathsf{e} \oplus \mathsf{e}$. *Pointer and array expressions* are: 1) the array-offset expression $\mathsf{e}[\mathsf{e}]$, 2) the ampersand (address-of) operator of the forms $\&\mathsf{VarID}$ and $\&\mathsf{e}[\mathsf{e}]$, and 3) the star (pointer dereference) operator ${}^{*}(\mathsf{e})$. The intuitive meanings of these expressions are as in C. Moreover, there are *low-level expressions* that are necessary for reflecting the target memory model in the source language (as mentioned in Section I): four getters: $\mathsf{start}(\mathsf{e})$, $\mathsf{end}(\mathsf{e})$, $\mathsf{offset}(\mathsf{e})$, and $\mathsf{capType}(\mathsf{e})$; and a setter: $\mathsf{limRange}(\mathsf{e},\mathsf{e},\mathsf{e})$. These low-level expressions operate on a capability-based representation of memory addresses that we explain next.

Addresses in ImpMod are represented as capabilities. Thus, built into the memory model is a type Cap for capabilities that is distinct from integers $\mathbb{Z}$. Hence, run-time values V are integers $\mathbb{Z}$ or capabilities Cap. A memory $\mathsf{Mem} : \mathbb{Z} \xrightarrow{\text{fin}} \mathsf{V}$ is a finite map from addresses to values. Note that the range of a memory may contain capability values. Concretely, we define the type Cap of capability values as $\mathsf{Cap} \stackrel{\text{def}}{=} \{\kappa, \delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. The first field of the capability value indicates its type: code (κ) or data (δ). The getter of this field is the expression $\mathsf{capType}$. The next three (integer) fields are respectively the start, end, and offset of the capability. The start and end identify the memory region on which this capability authorizes a code (κ) or data (δ) access operation. The offset designates the address at which an access operation is performed. The offset should be within range (checked by Rule Eval-star below).

The semantics of expressions are defined in Figure 1 by the judgment $\mathsf{e} \Downarrow \mathsf{v}$. Notice that expressions in ImpMod do not have side effects. The evaluation context for expressions consists of:

1) **syntactic** information about the program given by the function definitions Fd, the declarations of module-global variables MVar, and the layout and bounds β of all the program's variables;
2) **load-time** information about the program given by the per-module **d**ata-segment-location Δ, and the per-module local-**s**tack-location Σ; and
3) **execution-state** information given by the memory Mem, the stack pointers Φ of the module-local stacks and the program counter pc.

We make two observations on the rules of Figure 1. First, Rules Eval-amp-local-var and Eval-amp-module-var are the only two rules (of expression evaluation) where a capability value *originates*. In fact, this capability value originates out of thin air. This behavior (the minting of a capability value during the course of the program execution) is precisely why this source semantics is not really executable as is on a capability architecture. On a capability architecture, a program does not have any means to forge a capability value. Instead, it can only refer to capability registers[6]. (Contrast these two rules to Rules Eval-ddc and Eval-stc of Section III-B.) Replacing the mentions of program variables with expressions that mention capability registers is thus the key role of a *PAC* compiler.

Second, Rule Eval-star performs a bounds check before it loads a value from memory. Building-in this check is how we define ImpMod to be spatially safe by relying on the capability-based memory model. A similar bounds check is performed at *store* time as well (see the rule for the Assign command in the TR).

2) Commands and execution state of ImpMod*:* Commands of ImpMod are denoted Cmd. Unlike expressions, they can modify memory and other parts of the execution state.

$$\mathsf{Cmd} ::= \mathsf{Assign}\ \mathsf{e_l}\ \mathsf{e_r} \mid \mathsf{Alloc}\ \mathsf{e_l}\ \mathsf{e_{size}} \mid \mathsf{Call}\ \mathsf{fid}\ \overline{\mathsf{e}} \mid \mathsf{Return} \mid \mathsf{JumpIfZero}\ \mathsf{e_c}\ \mathsf{e_{off}} \mid \mathsf{Exit}$$

The commands should be self-explanatory (fid is a function name). An execution state $\mathsf{s} \stackrel{\text{def}}{=} \langle \mathsf{Mem}, \Phi, \mathsf{pc}, \mathsf{stk}, \mathsf{nalloc} \rangle$ of a program in ImpMod consists of the memory Mem, the stack pointers Φ of the module-local stacks, the program counter pc, a trusted *control* stack stk—which is shared among all modules of a program—and the memory-allocation status represented by the *next-free-address* nalloc. The space for dynamic memory allocation (i.e., the heap) is also shared by all the program modules (like the trusted control stack). The purpose of modeling a control stack that is trusted and hence separate from the accessible memory is that we want to rule out all ill-formed control sequences. No command in ImpMod can write directly to this control stack.

The semantics are small step and use the judgment $\rightarrow\ \subseteq \mathsf{s} \times \mathsf{s}$ that is indexed with an evaluation context $\mathsf{Fd}, \mathsf{MVar}, \beta, \Delta, \Sigma$ and an allocation limit ∇. The allocation limit (∇) is the maximum possible size of the shared heap. See the TR for details of the semantics.

B. CHERIExp*: The target language*

Our target language CHERIExp is, like ImpMod, an imperative language with modules. However, unlike ImpMod, it does not feature variables. Instead, it only features "capability registers" and integers as base expressions. The role of the compiler from ImpMod to CHERIExp is to implement operations on source pointers by using operations on capability registers. The memory model is like that of ImpMod (Section III-A1), i.e., it is capability based. Through this capability-based memory model and capability registers, CHERIExp models a slightly abstracted and simplified version of CHERI assembly [5].

[6]It can also get capabilities by calling the memory allocator, which in turn will place in a register (or in the program's memory) a new capability authorizing access to the allocated region.

1) Expressions and commands in CHERIExp*:*

$$\mathbf{e} ::= \mathbb{Z} \mid \mathbf{getddc} \mid \mathbf{getstc} \mid \mathbf{e} \oplus \mathbf{e} \mid \mathbf{inc}(\mathbf{e},\mathbf{e}) \mid \mathbf{deref}(\mathbf{e}) \mid \mathbf{start}(\mathbf{e}) \mid \mathbf{end}(\mathbf{e}) \mid \mathbf{offset}(\mathbf{e}) \mid \mathbf{capType}(\mathbf{e}) \mid \mathbf{limRange}(\mathbf{e},\mathbf{e},\mathbf{e})$$

Expressions are denoted $\mathbf{e}$. The language has three named capability registers: **stc**, **ddc** and **pcc**. The names of these registers hint at their recommended usage: **ddc** stands for default data capability while **stc** stands for stack capability. Our compiler uses **ddc** as a capability on the per-module data segment, and **stc** on the per-module stack. The third capability register, **pcc**, holds the program counter capability which points to the current command and allows the execution of commands. The expressions **getddc** and **getstc** immediately return the current value of the respective capability registers, **ddc** and **stc**. (See rule Eval-ddc in Figure 2.) However, **pcc** cannot be read in CHERIExp. This is a simple way of enforcing the restriction (mentioned in Section I) on linking with contexts that mention the **pcc** register. No expression allows the fabrication of an arbitrary capability, thus enforcing *capability unforgeability*.

In CHERIExp, one can increment (decrement) the offset of a capability by an arbitrary integer value (see rule Eval-inc), resulting in a new capability that nevertheless has the same bounds as the original. The check that the offset lies within bounds is only performed at use time (e.g., use by means of a **deref** expression). Observe that rule Eval-deref performs the same bounds-check on the capability value that rule Eval-star (of ImpMod) performs. $\mathbf{limRange}(\mathbf{e_1},\mathbf{e_2},\mathbf{e_3})$ *restricts* the lower- and upper-bounds of the capability $\mathbf{e_1}$ to the interval $[\mathbf{e_2},\mathbf{e_3})$, returning a new capability.

Commands in CHERIExp are the same as commands in ImpMod (modulo expressions). Briefly, the small-step command reduction is denoted with $\rightarrow\ \subseteq \mathbf{s} \times \mathbf{s}$ where a state $\mathbf{s}$ of a CHERIExp program, as in ImpMod, consists of a memory **Mem**, an allocation status **nalloc** and a trusted stack **stk**, but, unlike ImpMod, additionally consists of three capability registers: **ddc**, **stc** and **pcc**, and a map **mstc** holding a per-module capability authorizing access to the module's local *data* stack (the local data stack is part of the memory **Mem**).

Note that we model both a trusted stack (and hence a secure calling convention), and a separate map for the per-module data-stack capability in CHERIExp. This built-in segregation may sound too abstract for a *target* language that has low-level elements like capability registers. However, this modeling choice allows us to focus only on the *PAC* principle for program variables without worrying about the integrity of the stack pointer. (Prior work has already shown how compilers can enforce well-bracketed control flow and stack encapsulation using capabilities [16].)

IV. Our *PAC* compiler

Our ImpMod to CHERIExp compiler ($[\![\cdot]\!]$) translates pointers to capabilities (the *PAC* principle). In this section, we present its crucial bits, namely, the translation of ImpMod

$$\frac{(\mathsf{fid},\_) = \mathsf{pc} \quad \mathsf{vid} \in \mathtt{localIDs}\,(\mathsf{Fd}(\mathsf{fid})) \cup \mathtt{args}\,(\mathsf{Fd}(\mathsf{fid})) \quad \mathsf{mid} = \mathtt{moduleID}\,(\mathsf{Fd}(\mathsf{fid})) \quad \beta(\mathsf{vid},\mathsf{fid},\mathsf{mid}) = [\mathsf{st},\mathsf{end}) \quad \phi = \Sigma(\mathsf{mid}).1 + \Phi(\mathsf{mid})}{\&\mathsf{vid} \Downarrow (\delta, \phi + \mathsf{st}, \phi + \mathsf{end}, 0)}\ \text{(Eval-amp-local-var)}$$

$$\frac{(\mathsf{fid},\_) = \mathsf{pc} \quad \mathsf{vid} \notin \mathtt{localIDs}\,(\mathsf{Fd}(\mathsf{fid})) \cup \mathtt{args}\,(\mathsf{Fd}(\mathsf{fid})) \quad \mathsf{mid} = \mathtt{moduleID}\,(\mathsf{Fd}(\mathsf{fid})) \quad \mathsf{vid} \in \mathsf{MVar}(\mathsf{mid}) \quad \beta(\mathsf{vid},\bot,\mathsf{mid}) = [\mathsf{st},\mathsf{end})}{\&\mathsf{vid} \Downarrow (\delta, \Delta(\mathsf{mid}).1 + \mathsf{st}, \Delta(\mathsf{mid}).1 + \mathsf{end}, 0)}\ \text{(Eval-amp-module-var)}$$

$$\frac{\&\mathsf{e}_{\mathsf{arr}} \Downarrow (\delta,\mathsf{st},\mathsf{end},\mathsf{off}) \quad \mathsf{e}_{\mathsf{idx}} \Downarrow \mathsf{off}' \quad \mathsf{off}' \in \mathbb{Z}}{\&\mathsf{e}_{\mathsf{arr}}[\mathsf{e}_{\mathsf{idx}}] \Downarrow (\delta,\mathsf{st},\mathsf{end},\mathsf{off}+\mathsf{off}')}\ \text{(Eval-amp-arr)} \qquad \frac{\mathsf{e} \Downarrow (\delta,\mathsf{st},\mathsf{end},\mathsf{off}) \quad \mathsf{st} \le \mathsf{st}+\mathsf{off} < \mathsf{end}}{{}^{*}(\mathsf{e}) \Downarrow \mathsf{Mem}(\mathsf{st}+\mathsf{off})}\ \text{(Eval-star)}$$

Fig. 1: (Excerpt) Evaluation of expressions in ImpMod. The evaluation relation $\Downarrow$ is indexed with an evaluation context $\mathsf{Fd}, \mathsf{MVar}, \beta, \Delta, \Sigma, \mathsf{Mem}, \Phi, \mathsf{pc}$, which we elide for brevity.

$$\frac{}{\mathbf{Mem, ddc, stc, pcc \vdash getddc \Downarrow ddc}}\ \text{(Eval-ddc)} \qquad \frac{}{\mathbf{Mem, ddc, stc, pcc \vdash getstc \Downarrow stc}}\ \text{(Eval-stc)}$$

$$\frac{\mathbf{Mem, ddc, stc, pcc \vdash e \Downarrow (x, st, end, off)} \quad \mathbf{Mem, ddc, stc, pcc \vdash e_z \Downarrow v_z} \quad \mathbf{v_z \in \mathbb{Z}}}{\mathbf{Mem, ddc, stc, pcc \vdash inc(e, e_z) \Downarrow (x, st, end, off + v_z)}}\ \text{(Eval-inc)}$$

$$\frac{\mathbf{Mem, ddc, stc, pcc \vdash e \Downarrow (\delta, st, end, off)} \quad \mathbf{st \le st + off < end}}{\mathbf{Mem, ddc, stc, pcc \vdash deref(e) \Downarrow Mem(st + off)}}\ \text{(Eval-deref)}$$

$$\frac{\mathbf{Mem, ddc, stc, pcc \vdash e_1 \Downarrow (x, st, end, off)} \quad \mathbf{Mem, ddc, stc, pcc \vdash e_2 \Downarrow st'} \quad \mathbf{Mem, ddc, stc, pcc \vdash e_3 \Downarrow end'} \quad \mathbf{st' \in \mathbb{Z}} \quad \mathbf{end' \in \mathbb{Z}} \quad \mathbf{[st', end') \subseteq [st, end)}}{\mathbf{Mem, ddc, stc, pcc \vdash limRange(e_1, e_2, e_3) \Downarrow (x, st', end', 0)}}\ \text{(Eval-limRange)}$$

Fig. 2: (Excerpt) Evaluation of expressions in **CHERIExp**.

expressions to **CHERIExp** expressions. The translations of commands (denoted $(\!|\cdot|\!)$) and of modules ($[\![\cdot]\!]$) are rather trivial due to the similarity of the syntax of the source and target commands and modules, so we elide those.

The translation of expressions $\wr\cdot\int : \mathsf{e} \to \mathbf{e}$, whose excerpts are presented below, is indexed by the syntactic information $\mathsf{fid}, \mathsf{modID}$ (function id and module id) providing the scope of the expression being translated, and β, giving the layout and bounds of source variables.

$$\begin{aligned}
\wr \mathsf{z} \int_{_} &\stackrel{\text{def}}{=} \mathbf{z} \\
\wr \mathsf{e}_1 \oplus \mathsf{e}_2 \int_{\mathsf{fid},\mathsf{modID},\beta} &\stackrel{\text{def}}{=} \wr \mathsf{e}_1 \int_{\mathsf{fid},\mathsf{modID},\beta} \oplus \wr \mathsf{e}_2 \int_{\mathsf{fid},\mathsf{modID},\beta} \\
\wr \&\mathsf{vid} \int_{_,\mathsf{modID},\beta} &\stackrel{\text{def}}{=} \mathbf{limRange(getddc,} \\
&\qquad \mathbf{start(getddc)} + \mathsf{st}, \\
&\qquad \mathbf{start(getddc)} + \mathsf{end}) \\
&\quad \text{when } \beta(\mathsf{vid}, \bot, \mathsf{modID}) = (\mathsf{st}, \mathsf{end}) \\
\wr \&\mathsf{vid} \int_{\mathsf{fid},\mathsf{modID},\beta} &\stackrel{\text{def}}{=} \text{let } s = \mathbf{getstc} \text{ in} \\
&\qquad \text{let } so = \mathbf{start}(s) + \mathbf{offset}(s) \text{ in} \\
&\qquad \mathbf{limRange}(s,\ \mathsf{st} + so,\ \mathsf{end} + so) \\
&\quad \text{when } \beta(\mathsf{vid}, \mathsf{fid}, \mathsf{modID}) = (\mathsf{st}, \mathsf{end}) \\
\wr \mathsf{vid} \int_{\mathsf{fid},\mathsf{mid},\beta} &\stackrel{\text{def}}{=} \mathbf{deref}(\wr \&\mathsf{vid} \int_{\mathsf{fid},\mathsf{mid},\beta}) \\
\wr \&\mathsf{e}_{\mathsf{arr}}[\mathsf{e}_{\mathsf{off}}] \int_{\mathsf{fid},\mathsf{mid},\beta} &\stackrel{\text{def}}{=} \mathbf{inc}(\wr \&\mathsf{e}_{\mathsf{arr}} \int_{\mathsf{fid},\mathsf{mid},\beta}, \wr \mathsf{e}_{\mathsf{off}} \int_{\mathsf{fid},\mathsf{mid},\beta}) \\
\wr \mathsf{e}_{\mathsf{arr}}[\mathsf{e}_{\mathsf{off}}] \int_{\mathsf{fid},\mathsf{mid},\beta} &\stackrel{\text{def}}{=} \mathbf{deref}(\wr \&\mathsf{e}_{\mathsf{arr}}[\mathsf{e}_{\mathsf{off}}] \int_{\mathsf{fid},\mathsf{mid},\beta})
\end{aligned}$$

Translating expressions start, end, offset, capType, and limRange is straightforward and is similar to $\wr \mathsf{e}_1 \oplus \mathsf{e}_2 \int$. As an example, we show the translation of Line 11 of Listing III.1.

$$\begin{aligned}
&(\!|\mathsf{Call\ send_rcv}(\&\mathsf{iobuffer})|\!) \\
&= \mathbf{Call\ send_rcv\ (limRange(} \\
&\qquad \mathbf{ddc, start(ddc) + 0, start(ddc) + 512))}
\end{aligned}$$

Note that the compiler uses the bounds information that is given in the text of the program in the declaration of the array (Line 4) to introduce explicit curbing (using **limRange**) of the **ddc** capability so that the resulting capability is of the same size as the declared array size (**512**) and not bigger. This curbing prevents the external function `send_rcv` from offsetting the capability beyond the span of `iobuffer` and from accessing other variables like `secret`. This curbing is, in fact, essential for full abstraction since a similar out-of-bounds access is prohibited in the source semantics.

V. Proving the Compiler Fully Abstract

We prove our compiler fully abstract following the steps in Section II. We fill in details of two of the steps here: The definition of traces and the proof of Lemma 5 using *TrICL*.

(Return-to-program)

$$\frac{\begin{array}{c} \mathbf{s} \rightarrow \mathbf{s}' \qquad \mathbf{s.M_c(s.pcc)} = \mathbf{Return} \\ \mathbf{s.pcc} \not\subseteq \mathtt{dom}\,(\mathbf{p.M_c}) \qquad \mathbf{s'.pcc} \subseteq \mathtt{dom}\,(\mathbf{p.M_c}) \\ \varsigma' = \mathtt{reachable\_addresses\_closure}\,(\varsigma, \mathbf{s'.Mem}) \\ \mathbf{Mem_{shr}} = \mathbf{s'.Mem}|_{\varsigma'} \end{array}}{(\mathbf{s}, \varsigma) \xrightarrow{\mathtt{ret}?\ \mathbf{Mem_{shr}},\ \mathbf{s.nalloc}}_{\mathbf{p}} (\mathbf{s}', \varsigma')}$$

Fig. 3: Trace semantics of **CHERIExp** (Excerpt). The trace-step relation is indexed with a program $\mathbf{p}$.

A. *Traces*

As explained in Section II, we define traces by augmenting the small-step semantics of whole programs (Section III) with labels that capture information about the interaction between the program p of interest and its context. In our two languages, such an interaction happens only through shared memory or function arguments in only those steps that transfer control from the program to the context or vice-versa. Accordingly, a trace label arises only at such *border-crossing* control transfer steps, and the label records the shared memory, and the function arguments as in Laird [28]. In the following, we give a formal account of this development for the target language **CHERIExp**; the account for the source language ImpMod is similar.

The labeled trace-step relation for **CHERIExp** is written $\xrightarrow{\lambda}_{\mathbf{p}}$. The relation relates two *trace states* and a label λ. A trace state $(\mathbf{s}, \varsigma)$ extends the normal execution state $\mathbf{s}$ with auxiliary information ς, which is the set of memory addresses shared so far (i.e., from the initial state and up until execution state $\mathbf{s}$) between the program of interest $\mathbf{p}$ and the context. This auxiliary information is used to define an informative trace label λ. A trace α is a finite list $\overline{\lambda}$ of labels λ. Trace labels λ (of both our languages) have the following forms:

$$\lambda \ ::= \ \tau \mid \checkmark \mid \mathtt{ret}\ ?\ Mem, nalloc \mid \mathtt{ret}\ !\ Mem, nalloc \mid \mathtt{call}(fid)\ \overline{v}\ ?\ Mem, nalloc \mid \mathtt{call}(fid)\ \overline{v}\ !\ Mem, nalloc$$

- A silent label τ abstracts over any execution step that is *internal* to either the program $\mathbf{p}$ or the context.
- A termination label $\checkmark$ indicates that a terminal execution state was reached. (Once a $\checkmark$ appears, it re-appears in all subsequent trace labels.)
- An input call label $\mathtt{call}(fid)\ \overline{v}\ ?\ Mem, nalloc$ indicates that at an execution state where the shared memory was Mem, and the memory allocator state was $nalloc$, the context called the program's function fid with the list of values $\overline{v}$ as arguments.
- An output call label $\mathtt{call}(fid)\ \overline{v}\ !\ Mem, nalloc$ is similar to an input call label but the call goes in the opposite direction: the program called the context's function fid.
- An input return label $\mathtt{ret}\ ?\ Mem, nalloc$ indicates that at an execution state where the shared memory was Mem, and the allocator state was $nalloc$, the context returned to the program.
- An output return label $\mathtt{ret}\ !\ Mem, nalloc$ is similar except that the program returned control to the context.

Figure 3 shows the input return rule of $\xrightarrow{\lambda}_{\mathbf{p}}$. The third and fourth premises check that the program counter capability $\mathbf{pcc}$ belongs to $\mathbf{p}$'s code memory after the transition but not before, implying that this is a border crossing from the context into the program of interest $\mathbf{p}$. Similar checks exist in other rules (shown in the TR).

Next, given a reduction sequence of labeled steps, we drop all τ labels from it, and concatenate the non-τ labels into a *trace* α, writing $(s, \varsigma) \xrightarrow{\alpha}\!\!\!\!\!\to_p (s', \varsigma')$ for the resulting steps.[7] Then, we define the traces of a partial program p as in Definition 3. We note that all traces are *alternating* in "?" and "!".

Fact 1 (Traces are alternating)**.** $\alpha \in Tr(p) \implies \alpha \in \mathtt{Alt}\checkmark^*$ *where* $\mathtt{Alt} \overset{\text{def}}{=} (\dot{?}|\epsilon)\ (\dot{!}\dot{?})^*\ (\dot{!}|\epsilon)$ *and* $\dot{?}$ *is the set of* ?*-decorated labels, and similarly for* $\dot{!}$.

B. *Proof of Lemma 5 using TrICL*

Lemma 5 assumes a trace α for $[\![\mathsf{p_s}]\!]$ (in some target context, say, $\mathfrak{C}$) and requires constructing a source emulating context $\mathfrak{C}_{\mathsf{emu}}$ such that $\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]$ also has α. For this, we back-translate α to a source context. We illustrate the back-translation through an example. Figure 4 shows one trace that is emitted by the *compiled* version of the module Main of Listing III.1.

The compilation of the first three commands generate τ steps, which are dropped from traces. The next two non-τ labels (shown in the example) are interesting:

1) the function call send_rcv(&iobuffer) on Line 11 is border crossing, so its compilation emits an output call label which contains the callee function id (*send_rcv*), the argument to the call (the δ-capability representing the translation of the pointer &iobuffer), the direction of the call (! denoting output, i.e., program-to-context), a snapshot of the memory shared so far (namely, the contents of the array iobuffer), and the value -1 denoting the first heap address (the heap grows towards negative addresses in our semantics).
2) the target context (in which our compiled program executes) returns control to the compiled program, in this case, after *zeroing out the contents of the shared memory*. This emits an input return label.

Lemma 5 requires showing that exactly the same trace can be emitted by the source program in *some* source context. The proof of Lemma 5, therefore, requires us to construct such a source context, which we call the *emulating* context. The emulating context depends on the trace.[8] Listing V.1 shows an emulating context for our example trace.

```
Module id: Networking
Import module: HelperBackTranslation

current_trace_idx;

send_rcv(iob_ptr) {
  Call readAndIncrementTraceIdx(&current_trace_idx);
  Call saveArgs_send_rcv_1(iob_ptr);
  Call saveSnapshot_0();
```

[7] These technicalities, all worked out in our TR, are mostly inspired by process calculi [29].

[8] In principle, it could also depend on the target context, but this is usually not required. We also don't use the target context.

$$\texttt{call}\,\overbrace{(send\_rcv)}^{fid}\,\overbrace{[(\delta,\sigma,\sigma+512,0)]}^{\overline{v}}\ !\ \underbrace{[\sigma\mapsto 0,\dots,\sigma+42\mapsto 4242,\dots,\sigma+511\mapsto 0]}_{Mem},\ \underbrace{-1}_{nalloc}$$

$$::\ \texttt{ret}\ \ ?\ \overbrace{[\sigma\mapsto 0,\dots,\sigma+42\mapsto 0,\dots,\sigma+511\mapsto 0]},\ \overbrace{-1}$$

Fig. 4: Example trace of the compilation of the program in Listing III.1

```
10   Call doAllocations_1();
11   Call mimicMemory_1();
12   Return;
13 }
14 /*****************************************************/
15 Module id: HelperBackTranslation
16
17 current_trace_idx;
18 arg_store_0_send_rcv_0;
19 snapshot_0_σ;
20 ...
21 snapshot_0_σ + 511;
22
23 mimicMemory_1() {
24   Assign *(arg_store_0_send_rcv_0[0]) 0;
25   ...
26   Assign *(arg_store_0_send_rcv_0[511]) 0;
27   Return;
28 }
29 ...
```

Listing V.1: Example back-translation (simplified excerpt): An ImpMod context emulating the trace of Figure 4.

This emulating source context consists of two modules, Networking, which implements the API function send_rcv, and HelperBackTranslation, which implements helper functions and maintains metadata. We show just one example of such a helper function, namely mimicMemory_1(). mimicMemory_1() is called (on Line 11) by send_rcv(). It zeroes out the IO buffer (to mimic the shared memory in the second action of the given target trace). The IO buffer is accessed by mimicMemory_1() through the pointer stored in the global variable arg_store_0_send_rcv_0 (Line 18). This pointer is stored (not shown) by the function call saveArgs_send_rcv_1(iob_ptr) on Line 8.

We briefly explain what each helper function does. First, the context emulating a given trace defines a different set of helper functions for every position on the trace. The index of the corresponding trace label appears in the identifier of a helper function (for example, mimicMemory_1(), and saveSnapshot_0()). To explain the helper functions, we follow the body of send_rcv(iob_ptr) line by line. In the beginning, the call to readAndIncrementTraceIdx keeps track of the current position in the trace. This knowledge of the current position in the trace is not used in our toy example, but it would be used if the API function (send_rcv in this case) were called *at more than one position* in the given trace; at each such position, we would use this knowledge to call the corresponding helper functions (e.g., mimicMemory_3() instead of mimicMemory_1(), which would copy to the shared memory the values that appear in trace position 3 instead of 1).

Next, on Line 8, we store the pointer iob_ptr in a global variable by calling the HelperBackTranslation module because we may need it to simulate a *future* trace position, not just the current call to send_rcv. For the same reason, we save (on Line 9) a snapshot of the whole shared memory in global variables snapshot_0_σ to snapshot_0_$\sigma + 511$.

Next, on Lines 10 to 12, the actual emulation of the trace action at trace position 1 is done. In our example, doAllocations_1() would do nothing because *nalloc* in Figure 4 does not change. Importantly, mimicMemory_1() writes all the values (the zeros) to the shared memory before send_rcv eventually returns, thus mimicking the given target trace.

The difficult simulation: Returning back to the general picture of Lemma 5, after we have constructed the emulating source context from the given trace α, we must *prove* that the source program and this context emulate the given target trace. As explained in Section II, for this, we would like to set up a simulation between the target and source runs, but this simulation can be very hard because there can be differences between the *internal* behaviors of the emulating context and the given target context, e.g., in the specific order of updates to the shared memory, and in the internal function calls. Indeed, the target context likely does not use the kind of helper functions our emulating source context does! Our simulation needs to accommodate this "vertical gap".

To add to this difficulty, we do want to simulate through (the !-decorated) execution steps of the program of interest ($\mathsf{p_s}$ in Lemma 5). And since our compiler compiles it, this simulation would be very similar to what we would have to do anyhow just to prove our compiler correct, not fully abstract. So, we would like to "reuse" this part.

*TrICL **to the rescue:*** This is where our new *TrICL* simulation comes in. *TrICL* introduces a third "mediator" run to the simulation, namely, that induced by the *compilation of the whole source program* $[\![\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]]\!]$, containing both $\mathsf{p_s}$ and our emulating context, say, $\mathfrak{C}_{\mathsf{emu}}$. As explained in Section II, this simplifies the proof because we can "reuse" whole-program compiler correctness as a black box to immediately reduce the problem of showing that the *emulating* run emulates the given run to that of showing that the *mediating* run emulates the given run. The emulating and mediating runs are in the same language, so this reduces the "vertical gap".

Finally, to show that the mediating run emulates the given run in the target language, we rely on an *alternating simulation* in the target language that uses two different relations between the two runs – a *strong* relation $\approx_{[\![\mathsf{p_s}]\!]}$, which holds while the compilation of the program of interest executes, and a *weak* relation $\sim_{[\![\mathsf{p_s}]\!]}$, which holds while the contexts $\mathfrak{C}$ (which produced α) and $[\![\mathfrak{C}_{\mathsf{emu}}]\!]$ execute. The need for two relations will become clear shortly.

TrICL formally: Formally, *TrICL* is a relation between three trace states (a source emulating state $\mathsf{s_{emu}}$, a target mediating state $\mathbf{s_{med}}$ and a target given state $\mathbf{s_{given}}$ of the three runs explained above) that agree on the memory shared between the context and the program of interest. *TrICL* is indexed by a trace α that we are trying to emulate and a position i of that trace. The relation requires that (1) $\mathsf{s_{emu}}$ and $\mathbf{s_{med}}$ are related by the whole-program compiler correctness relation ($\cong_{\mathsf{p_s}}$), (2) the source state satisfies an *emulation invariant*, which basically captures that the construction of $\mathfrak{C}_{\mathsf{emu}}$ is indeed an emulation of the *input* steps of the trace α, e.g., that the functions mimicMemory_1() and doAllocations_1() indeed emulate the input trace action $\alpha(1)$, and (3) the two target states $\mathbf{s_{med}}$ and $\mathbf{s_{given}}$ are related by the strong relation $\approx_{[\![\mathsf{p_s}]\!]}$ when execution is in $[\![\mathsf{p_s}]\!]$ and by the weak relation $\sim_{[\![\mathsf{p_s}]\!]}$ when execution is in the contexts.

Definition 4 (Trace-Indexed Cross-Language (*TrICL*) alternating simulation relation)**.**

$$
\begin{aligned}
&\texttt{TrICL}\,(\mathsf{s_{emu}}, \mathbf{s_{med}}, \mathbf{s_{given}}, \varsigma)_{\alpha,i,\mathsf{p_s}} \overset{\mathsf{def}}{=} \\
&\quad \mathsf{s_{emu}} \cong_{\mathsf{p_s}} \mathbf{s_{med}} \;\wedge\; \texttt{emulate\_invariants}\,(\mathsf{s_{emu}})_{\alpha,i,\mathsf{p_s}} \\
&\quad \wedge\; (\alpha(i) \in \overset{\bullet}{!} \implies (\mathbf{s_{med}}, \varsigma) \approx_{[\![\mathsf{p_s}]\!]} (\mathbf{s_{given}}, \varsigma)) \\
&\quad \wedge\; (\alpha(i) \in \overset{\bullet}{?} \implies (\mathbf{s_{med}}, \varsigma) \sim_{[\![\mathsf{p_s}]\!]} (\mathbf{s_{given}}, \varsigma))
\end{aligned}
$$

Given this definition, we come to the key step of our proof, namely, that *TrICL* is an invariant.

Lemma 6 (*TrICL* step-wise alternating backward-simulation)**.**

$$
\begin{aligned}
&\alpha \in \texttt{Alt} \;\wedge\; \texttt{TrICL}\,(\mathsf{s_{emu}}, \mathbf{s_{med}}, \mathbf{s_{given}}, \varsigma)_{\alpha,i,\mathsf{p_s}} \;\wedge \\
&(\mathbf{s_{given}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{given}}, \varsigma') \\
\implies\; &\exists \mathsf{s'_{emu}}, \mathbf{s'_{med}}.\; (\mathsf{s_{emu}}, \varsigma) \xrightarrow{\alpha(i)}_{\mathsf{p_s}} (\mathsf{s'_{emu}}, \varsigma') \;\wedge \\
&(\mathbf{s_{med}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{med}}, \varsigma') \;\wedge \\
&\texttt{TrICL}\,(\mathsf{s'_{emu}}, \mathbf{s'_{med}}, \mathbf{s'_{given}}, \varsigma')_{\alpha,i+1,\mathsf{p_s}}
\end{aligned}
$$

Before sketching the proof of this key lemma, we explain how the proof reuses the backward- and forward- simulation lemmas that are anyhow needed for compiler correctness:

- When control is in the contexts (case $\alpha(i) \in \overset{\bullet}{?}$ of Definition 4), we know by the *emulation invariant* that the source context $\mathfrak{C}_{\mathsf{emu}}$ emulates the next action of the trace α (the emulation invariant holds of the construction shown in the example earlier), and we use *forward simulation* to argue that the mediating context, which is just the compilation of this source context, does the same.
- Dually, when control is in the program of interest (case $\alpha(i) \in \overset{\bullet}{!}$ of Definition 4), we know by the precondition of Lemma 6 that $[\![\mathsf{p_s}]\!]$ produces the next action of the *given* trace α, hence (by strong similarity that we explain below) $[\![\mathsf{p_s}]\!]$ also produces the same action in the *mediator* trace. But now from knowing that an action of the mediator trace was produced, we use *backward simulation* to reason that the source program $\mathsf{p_s}$ does the same action.

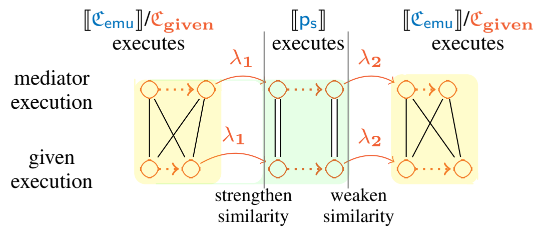

Fig. 5: Alternating strong and weak relations between the 'mediator' execution (top) and the 'given' execution (bottom).

This "reuse" of the (forward- and backward-) simulations that are anyhow needed for compiler correctness is the simplification that *TrICL* affords. The proof of the two interesting cases of Lemma 6 is given in Appendix A.[9]

The strong and weak relations: We now explain the strong relation $\approx_{[\![\mathsf{p_s}]\!]}$ and the weak relation $\sim_{[\![\mathsf{p_s}]\!]}$ between the target trace states of the mediating run (of $[\![\mathfrak{C}_{\mathsf{emu}}[\mathsf{p_s}]]\!]$) and the given run (of $\mathfrak{C}[[\![\mathsf{p_s}]\!]]$). Following Definition 4, the strong relation holds between trace states while control is in the compilation of the program of interest $[\![\mathsf{p_s}]\!]$ in the two runs. Since this program (part) is exactly the same in both runs, the strong relation is a lock-step simulation (Lemma 9), which maintains similarity of the call stacks, the private memory of the program of interest and the shared memory (memory accessible to both the context and the program of interest).

In contrast, the weak relation holds between trace states while control is in the contexts $[\![\mathfrak{C}_{\mathsf{emu}}]\!]$ and $\mathfrak{C}$. Since these two contexts can be very different, this relation is not lock-step. It allows one step on either side to be simulated by zero or more steps on the other side (Lemma 11) and allows a different sequence of internal function calls (and, hence, stack states). In terms of memory, it only enforces that the *private memory of the program of interest* remain in sync, but the shared memory and the private memory of the contexts may diverge arbitrarily.

In the proof of Lemma 6, these relations must be replaced by each other at border crossings. At a border crossing where control transfers from the program of interest to the contexts (case $\alpha(i) \in \overset{\bullet}{!}$), we must "weaken" the strong relation to the weak one. This is quite straightforward since the strong relation directly implies the weak relation.

In the other direction, at a border crossing where control transfers from the contexts to the program of interest (case $\alpha(i) \in \overset{\bullet}{?}$), we must "strengthen" the weak relation to the strong relation. For this, we must prove that the shared memory

[9]The proof relies on forward- and backward-simulation, and also on key properties of the strong and weak similarities and of the emulate invariants. All of these properties (Lemmas 7 to 14) are given in Appendix A, and are proved in the TR.

(recorded on $\alpha(i)$) is exactly the same in the two target runs at that point. This is not obvious and is proved as follows. First, we know from the *TrICL* invariants that the third trace (the emulating source trace) is mimicking the step $\alpha(i)$. We then use this conclusion to show that the trace label $\alpha(i)$ at the border crossing in question must be the same in the given and the mediator runs (the two target runs). Here is how: By applying compiler-correctness forward simulation to the emulating run so far, we conclude that the mediating trace must also take a step with the same label $\alpha(i)$. Since the shared memory is recorded in the label $\alpha(i)$, it immediately follows that the shared memory is exactly the same in the three runs, and in particular in the two target runs, which completes the strengthening proof.

Figure 5 depicts the alternating nature of the strong and weak relations, and the strengthening and weakening at border crossings. The two traces are depicted as two horizontal sequences of states/transitions. The black lines that connect states from opposite traces show the nature of the simulation condition: option simulation (Lemma 11) is possible for weak similarity (the single black line), while for strong similarity (the double black line), only lock-step simulation (Lemma 9) is possible.

VI. Implementation in a compiler from C to CHERI

We have implemented the key ideas of Section IV in a compiler from C to *CHERI*. *CHERI* comes with a Clang/LLVM compiler [10] that already implements *PAC*, but no isolation for modules, which ImpMod has. Our compiler enforces this isolation via a C-to-C source pre-processing transform. The compiler relies on `libcheri`, CheriBSD's library for building, invoking and loading *sandboxes* – isolated units of computation with their own code and memory, which is private until explicitly shared. `libcheri` relies on *CHERI*'s underlying support for object capabilities to provide sandboxes [30]. The important thing from our perspective is that `libcheri` requires the programmer to manually group functions into classes (the equivalent of modules), and to annotate functions to make them use object capabilities instead of the standard calling conventions (attribute `cheri_ccall`) and to specifically annotate functions that are exported from a class (attribute `cheri_ccallee`). Additionally, all initialization functions must be added to classes that call external functions.

Our source-to-source transform automates all this: It maps C modules (compilation units) to `libcheri`'s sandboxes to isolate them from each other and automatically inserts all the required annotations. Examples of programs output by our transform are shown in the TR. To initialize sandboxes, we had to make some significant changes to `libcheri` as well. Briefly, `libcheri` requires a second phase of runtime linking to resolve cross-sandbox references. We implement this through a new recursive initialization function, called once before `main`, which loads and initializes all sandboxes that `main` depends on transitively, creates relevant object capabilities, and links the modules.

Software	`zlib`	`LibYAML`	`GNU-barcode`	`libpng`
	Porting overhead summary			
Lines of code	11255	12762	4657	33029
Altered lines	130	114	164	51
Percentage	1.15%	0.89%	3.5%	0.15%
	No. of occurrences of each incompatible pattern			
Passed local ref	2	15	0	13
Extern global var	3	0	0	0
Pointer to ext fun	2	1	26	2
Other	0	1	1	4
Total	7	17	27	19

TABLE I: Porting overheads

Differences from the formalization: Our idealized source language ImpMod provides accessors for bounds information about pointers whereas native C, that our implementation supports, does not. *CHERI*'s C interface, however, provides pointers with such accessors. (These pointers wrap *CHERI* capabilities.) We rely on the programmer to use these *CHERI* pointers in place of native C pointers consistently. We have also not yet implemented the (straightforward) link-time check on target contexts to prevent them from accessing **pcc** directly, something that our ideal target language **CHERIExp** enforces.

VII. Experimental evaluation

We evaluate the overheads of our proof-of-concept isolation scheme using four large open-source C libraries that we chose carefully for heterogeneity and compiled with our compiler: `zlib` [31], `LibYAML` [32], `GNU-barcode` [33], and `libpng` [34].

Code changes: We had to make some manual code changes to address incompatibilities between our scheme and the existing toolchain. These incompatibilities are: (a) Moving local variables whose addresses are shared with other modules to the heap; this is a fundamental limitation of *CHERI*, not specific to our scheme. (b) Explicitly sharing pointers to exported global variables of a module; this can be automated with further work on the *CHERI* linker. (c) In the existing toolchain, a function pointer always compiles to an execute capability. Our isolating scheme requires object capabilities for cross-module calls. To compensate for this incompatibility, we had to make manual changes to cross-module calls using function pointers. Table I summarizes the magnitude of our changes on the various benchmarks. Overall, the changes are quite limited.

Performance overheads: The most significant performance overhead in our scheme comes from cross-module function calls, where we use the *CHERI* `ccall` instruction, which is used to implement object capabilities. This instruction is expensive as it performs a number of operations like saving and restoring callee-save registers, clearing unused argument registers and making certain checks on arguments.[10]

[10]Some of this overhead is *not* fundamental and the *CHERI* design has improved on `ccall` over time [9]. The overhead reported here is based on `ccall` as it exists in *CHERI* ISA version 5, and should be considered conservative.

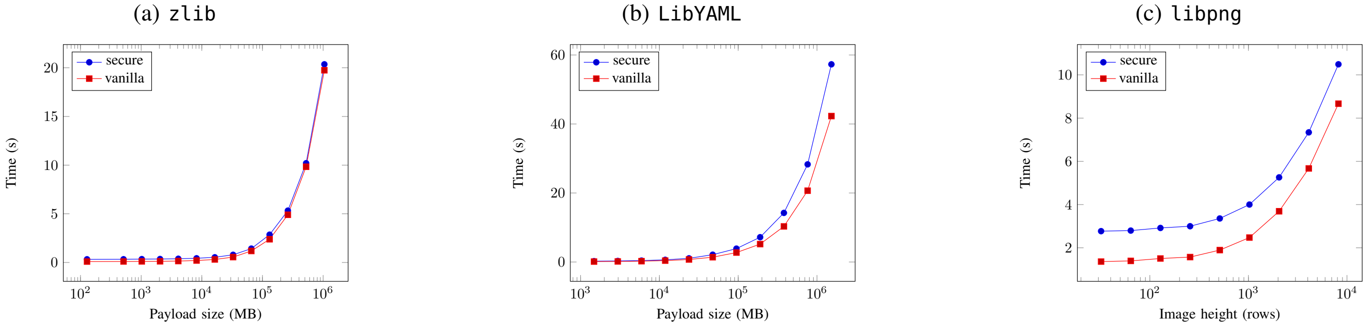

Fig. 6: Performance benchmark results. Note that x-axes are log-scale.

The number of cross-module calls is, of course, application dependent. To isolate this overhead, we compare the execution times of calls to library functions compiled with our scheme to those compiled with the vanilla *CHERI* compiler, which offers no module isolation and, hence, no security. We perform our experiments on a single-core *CHERI* VM implementing *CHERI* ISA version 5 [35] running our modified version of CheriBSD. This VM was hosted in another VirtualBox [36] VM with FreeBSD 9.1, which, in turn, runs on an arch-Linux host. The physical machine has a 4 core Intel Core i7-6700 CPU with 16 GB of RAM, of which 512 MB are allocated to the *CHERI* VM.

Figure 6 summarizes our results. The y-axes are execution times. The x-axes are log-scale. The lines "secure" and "vanilla" correspond to our scheme and vanilla *CHERI*, respectively. All times are averages of 10 runs. Standard deviations were all below 0.3s with the exception of the two longest `LibYAML` experiments, where they were 1.5s in each case. A small, constant delay of about 0.2s associated with the sandbox loading routines is included in both the baseline and the secure versions. We do not show the benchmark `GNU-barcode` as it takes very small inputs only.

Our overhead relative to vanilla *CHERI* is negligible for `zlib` as this benchmark has very few cross-module calls (43–441 as the payload size varies). In `LibYAML`, the number of cross-module calls grows linearly with input size, so the *ratio* of the two lines is nearly constant (our relative overhead is consistently between 35 and 40%). In `libpng`, the number of cross-module calls is constant but large (approx. 125,000), so the *difference* between the two lines is nearly constant and easily perceivable. Overall, these observations are in line with our expectation that the performance overhead of our isolation scheme is dominated by the number of cross-module calls and that the overhead of each such call is constant.

The absolute numbers reported here should not be extrapolated to other hardware as they were obtained on a *CHERI* simulator that is not cycle accurate. Nonetheless, we expect similar *trends* to hold, i.e, the overhead should be roughly linear in the number of cross-module calls on all similar hardware. Another noteworthy point is that a compiler may choose a different isolation boundary, e.g, it could isolate individual class objects to prevent access to their private fields from outside. In such a case, the primary overhead would shift to crossings of that isolation boundary. In the said example, the overhead would become proportional to the number of cross-object method invocations.

VIII. Related work

Verified compilation to capability machines: The work closest to ours is *StkTokens* [16] as it also verifies a compiler transformation that targets capability machines. However, *StkTokens* and our work are complementary: While we model and prove secure the *PAC* transformation, *StkTokens* models and verifies a transformation that implements a calling convention. To do this, *StkTokens* assumes a hardware extension called *linear capabilities*, for which experimental support exists in *CHERI*. We believe that *StkTokens* can be combined with our work to eliminate our trust in the control stack. Van Strydonck et al [37] also use linear capabilities. An important difference is that Van Strydonck et al's compiler relies on static contracts defining which memory is shared across trust boundaries, while ours considers components which share memory dynamically during execution.

Full abstraction for compiler security: The idea of using full abstraction to formalize compiler security was introduced by Abadi [20]. Compiler full abstraction (or security in general) is typically proved either with a trace-directed back-translation [38, 26, 27, 21] or a syntax-directed back-translation [39, 40, 16, 37, 41, 42]. A syntax-directed back-translation defines a reverse compiler that translates the assumed distinguishing target context (a piece of syntax) to a source context. While this approach has been used extensively in prior work, it can be difficult to use in situations where some target-level constructs have no obvious source counterparts. This is not the case for our languages, but we follow the trace-directed approach as we still find it technically simpler. We contribute to the trace-directed technique by introducing *TrICL*, which allows reuse of whole-program compiler correctness to simplify the proof of the "security direction" of full abstraction (Definition 2((ii))). We are not aware of similar reuse of whole-program compiler correctness in the syntax-directed method.

Other criteria for secure compilation: A recent line of work proposes alternate criteria for compiler security, based on preservation of classes of properties and hyperproperties in the presence of adversarial contexts [43, 44, 26, 45, 46]. Many of these criteria are incomparable to full abstraction, while some are stronger. Work on proof techniques for proving these criteria is still in early stages, but back-translation (both trace-directed and syntax-directed) features prominently. In particular, Abate *et al.* [26] describe a compiler and prove a strong robust safety property (not *FA*) for it using trace-directed back-translation. Their method also reuses whole-program compiler correctness to reduce a big part of cross-language reasoning to only target-level reasoning. However, their setting does not support sharing of memory across modules, which substantially simplifies their proof by eliminating the need for the strong and weak relations (Section V-B).

Another line of work [47, 48, 49, 50, 51] verifies that the compiler does not undo countermeasures that the programmer of cryptographic libraries implements in order to ensure protection against timing attacks or other secret-revealing attacks.

Fully-abstract trace semantics: Our trace labels are inspired by Laird's work for a functional language with general references [28]. Laird relies on a *bipartite* LTS in which nodes are partitioned between program configurations and environment (context) configurations. We do not use this segregation explicitly, but our checks on **pcc** in the trace semantics have the same effect.

IX. Limitations and Future Work

Our work shows formally that PAC compilers can provide strong guarantees for partial programs. While we believe that this is a significant step forward in the understanding of the security properties of PAC compilers, we still make some simplifications and assumptions that would be interesting to remove in future work. First, our memory allocation model does not support de-allocation. This simplification allows us to represent the state of the memory allocator as just the next-free-address, and this is essential in keeping our model manageable. To the best of our knowledge, nobody has yet developed fully abstract trace semantics for languages with a realistic model of deallocation. Work in this direction would be interesting. Second, we do not yet model side channels. As such, our compiler is not guaranteed to preserve resistance against side-channel leaks [52]. There is recent related work that specifically investigates how to secure compilers such that they preserve side-channel resistance [47, 50, 51, 48, 49]. Combining PAC with these ideas would also be interesting.

Acknowledgments: We thank several anonymous reviewers for their useful feedback. This research was partially funded by the Research Fund KU Leuven, by the Research Foundation - Flanders (FWO), by a gift from Intel Corporation, and by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

[1] R. S. Fabry, "Capability-based addressing," *Commun. ACM*, vol. 17, no. 7, pp. 403–412, Jul. 1974.

[2] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *Proc. ASPLOS*, 1994.

[3] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, 1975.

[4] 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*, vol. 42, no. 3, Jun. 2014.

[5] 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)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-927, 2019. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf

[6] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The m-machine multicomputer," *International Journal of Parallel Programming*, vol. 25, no. 3, pp. 183–212, 1997.

[7] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.

[8] R. N. Watson, P. G. Neumann, and S. W. Moore, "Balancing disruption and deployability in the CHERI instruction-set architecture (ISA)," in *New Solutions for Cybersecurity*. MIT Press, 2017.

[9] 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*, vol. 36, no. 5, 2016.

[10] 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 *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2015.

[11] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the PDP-11: Architectural support for a memory-safe C abstract machine," in *Proc. ASPLOS*, 2015.

[12] R. N. Watson, A. Richardson, B. Davis, J. Baldwin, D. Chisnall, J. Clarke, N. Filardo, S. W. Moore, E. Napierala, P. Sewell *et al.*, "CHERI C/C++ Programming Guide," 2020. [Online]. Available: https://github.com/CTSRD-CHERI/cheri-c-programming

[13] A. L. Georges, A. Guéneau, T. Van Strydonck,

A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, and L. Birkedal, “Efficient and provable local capability revocation using uninitialized capabilities,” in *Proc. POPL*, 2021.

[14] S. Tsampas, D. Devriese, and F. Piessens, “Temporal safety for stack allocated memory on capability machines,” 2019. [Online]. Available: https://lirias.kuleuven.be/retrieve/538854

[15] L. Skorstengaard, D. Devriese, and L. Birkedal, “Reasoning about a machine with local capabilities,” in *Proc. ESOP*, 2018.

[16] ——, “Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities,” in *Proc. POPL*, 2019.

[17] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson *et al.*, “CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety,” in *Proc. MICRO*, 2019.

[18] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin *et al.*, “Cornucopia: Temporal safety for CHERI heaps,” in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[19] K. Nienhuis, A. Joannou, A. Fox, M. Roe, T. Bauereiss, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann *et al.*, “Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process,” in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[20] M. Abadi, “Protection in programming-language translations,” in *Proc. ICALP*, 1998.

[21] M. Patrignani, A. Ahmed, and D. Clarke, “Formal approaches to secure compilation: A survey of fully abstract compilation and related work,” *ACM Comput. Surv.*, vol. 51, no. 6, Feb. 2019.

[22] “CapablePtrs Public Repository,” https://gitlab.mpi-sws.org/FCS/capabilities-compilation-public, 2021.

[23] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, “Fully abstract compilation to JavaScript,” in *Proc. POPL*, 2013.

[24] X. Leroy, “Formal verification of a realistic compiler,” *Comm. ACM*, vol. 52, no. 7, 2009.

[25] D. Patterson and A. Ahmed, “The next 700 compiler correctness theorems (functional pearl),” in *Proc. ICFP*, 2019.

[26] 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 *Proc. CCS*, 2018.

[27] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, “Secure compilation to protected module architectures,” *ACM Tran. Prog. Lang. Sys. (TOPLAS)*, vol. 37, no. 2, 2015.

[28] J. Laird, “A fully abstract trace semantics for general references,” in *Proc. ICALP*, 2007.

[29] R. Milner, *Communicating and mobile systems: The π-calculus*. Cambridge University Press, 1999.

[30] R. M. Norton, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff, “Capability Hardware Enhanced RISC Instructions: CHERI Programmer’s Guide,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-877, Sep. 2015. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-877.pdf

[31] M. A. Jean-loup Gailly, “zlib 1.2.11,” 2017. [Online]. Available: https://zlib.net/

[32] K. Simonov, “Libyaml 0.1.7,” 2017. [Online]. Available: https://pyyaml.org/wiki/LibYAML

[33] A. Rubini, “Gnu-barcode 0.99,” 2013. [Online]. Available: https://www.gnu.org/software/barcode/

[34] G. R.-P. Guy Eric Schalnat, Andreas Eric Dilger, “libpng 1.6.34,” 2017. [Online]. Available: http://www.libpng.org/pub/png/libpng.html

[35] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia, “Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6),” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-891, 2016. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-891.pdf

[36] P. Dash, *Getting Started with Oracle VM VirtualBox*. Packt Publishing, 2013.

[37] T. V. Strydonck, F. Piessens, and D. Devriese, “Linear capabilities for fully abstract compilation of separation-logic-verified code,” in *Proc. ICFP*, 2019.

[38] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, and B. C. Pierce, “Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation,” in *Proc. CSF*, 2016.

[39] M. S. New, W. J. Bowman, and A. Ahmed, “Fully abstract compilation via universal embedding,” in *Proc. ICFP*, 2016.

[40] D. Devriese, M. Patrignani, F. Piessens, and S. Keuchel, “Modular, fully-abstract compilation by approximate back-translation,” *Log. Methods Comput. Sci.*, vol. 13, no. 4, 2017.

[41] A. Ahmed and M. Blume, “Typed closure conversion preserves observational equivalence,” in *Proc. ICFP*, 2008.

[42] ——, “An equivalence-preserving CPS translation via multi-language semantics,” in *Proc. ICFP*, 2011.

[43] 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 *Proc. CSF*, 2019.

[44] M. Patrignani and D. Garg, “Robustly safe compilation,” in *Proc. ESOP*, 2019.

[45] ——, “Secure Compilation and Hyperproperties Preser-

vation," in *Proc. CSF*, 2017.
[46] 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 *Proc. ESOP*, 2020.
[47] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," in *Proc. POPL*, 2019.
[48] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"," in *Proc. CSF*, 2018.
[49] R. Sison and T. Murray, "Verifying That a Compiler Preserves Concurrent Value-Dependent Information-Flow Security," in *Proc. ITP*, 2019.
[50] F. Besson, A. Dang, and T. Jensen, "Securing compilation against memory probing," in *Proc. PLAS*, 2018.
[51] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-wasm: Type-driven secure cryptography for the web ecosystem," in *Proc. POPL*, 2019.
[52] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *IEEE Security and Privacy Workshops (SPW)*, 2015.

APPENDIX

A. *Some details of the proof of Lemma 6*

We show some excerpted definitions of key constituents of our *TrICL* relation and the proof of Lemma 6, the step-wise backward simulation condition of *TrICL*, which is an interesting proof because it puts together the whole-program compiler correctness lemmas. We group the (discharged) assumptions used in the proof of Lemma 6 (namely, Lemmas 7 to 14 below) by the components of *TrICL* they refer to. We recall that *TrICL* is defined in terms of four main relations/invariants:

1) the vanilla (whole-program) **compiler-correctness relation** ($\cong_{p}$) between the source state and the mediator state satisfying lifted forward- and backward-simulations (**Lemmas 7 and 8**)
2) a **strong-similarity relation** ($\approx_{p}$) between the mediator state and the given state, a relation that satisfies lock-step simulation (**Lemma 9**)
3) a **weak-similarity relation** ($\sim_{p}$) also between the mediator state and the given state, a relation that satisfies option simulation (**Lemma 11**), and – together with the strong similarity – satisfying both weakening (**Lemma 10**) and strengthening (**Lemma 12**)
4) **emulation invariants** about the source state satisfying both adequacy (**Lemma 13**) and preservation by trace steps (**Lemma 14**)

Definition 5 (Weak-similarity relation (excerpt))**.** $(\mathbf{s_1}, \varsigma) \sim_{\mathbf{p}} (\mathbf{s_2}, \varsigma) \overset{\text{def}}{=} \mathbf{s_1.stk} \sim_{\mathbf{p}} \mathbf{s_2.stk} \ \wedge\ \mathbf{s_1.mem} \sim_{\mathbf{p},\varsigma} \mathbf{s_2.mem}$
where weak stack similarity and weak memory similarity are defined as follows:
$\mathbf{s_1.stk} \sim_{\mathbf{p}} \mathbf{s_2.stk}$ *is defined to be the existence of a map between the indices of the entries of* $\mathbf{s_1.stk}$ *and the indices of the entries of* $\mathbf{s_2.stk}$*, such that this map is "successor preserving", monotone, and every pair of indices that is in the map corresponds to two equal stack entries (one from* $\mathbf{s_1.stk}$ *and the second from* $\mathbf{s_2.stk}$*). (See section 6.3 of the technical report for formal definitions.)*
$\mathbf{s_1.mem} \sim_{\mathbf{p},\varsigma} \mathbf{s_2.mem}$ *is defined to be equality of the contents of memories* $\mathbf{s_1.mem}$ *and* $\mathbf{s_2.mem}$ *at all the currently-private addresses of* $\mathbf{p}$*, where a currently-private address of* $\mathbf{p}$ *is any address that is so far not shared (i.e., not in the set* ς*) and that is reachable from* $\mathbf{p}$*'s statically-allocated memory.*

Definition 6 (Strong-similarity relation (excerpt))**.** $(\mathbf{s_1}, \varsigma) \approx_{\mathbf{p}} (\mathbf{s_2}, \varsigma) \overset{\text{def}}{=} \mathbf{s_1.stk} \approx_{\mathbf{p}} \mathbf{s_2.stk} \ \wedge\ \mathbf{s_1.mem} \approx_{\mathbf{p}} \mathbf{s_2.mem}$ *where strong stack similarity and strong memory similarity are defined as follows:*
$\mathbf{s_1.stk} \approx_{\mathbf{p}} \mathbf{s_2.stk}$ *is defined to be the same as weak stack similarity, i.e., the existence of a map with the same properties mentioned in Definition 5, but with the extra condition that the top-most stack indices must be in the map. (This extra condition ensures that all function calls and returns happen in sync, hence, satisfying the lock-step simulation condition of Lemma 9.)*
$\mathbf{s_1.mem} \approx_{\mathbf{p}} \mathbf{s_2.mem}$ *is defined to be equality of the contents of memories* $\mathbf{s_1.mem}$ *and* $\mathbf{s_2.mem}$ *at all the addresses reachable from* $\mathbf{p}$*'s statically-allocated memory.*

Definition 7 (Emulation invariants (simplified excerpt))**.** *The emulation invariants for a state* s *and the prefix of a finite trace* α *up to position* i *consist of invariants on the shape of the code that is currently executing (i.e., the code starting at address* $\mathsf{s.pc}$*), in addition to invariants on memory* $\mathsf{s.mem}$ *that ensure the metadata (that the back-translated context keeps) is compatible with the information from the trace prefix of* α *up to position* i*. The invariants on memory can be found in the technical report in definitions 116, 117, 118, and 126. The emulation invariants on the shape of the code are more interesting:*

$\texttt{emulate\_invariants}\,(\mathsf{s})_{\alpha,i,\mathbf{p}} \overset{\text{def}}{=}$
$\alpha(i) \in \overset{\bullet}{?} \implies \exists j.\ j \le i \ \wedge\ \texttt{upcoming\_commands}(\mathsf{s}, \texttt{emulate\_responses\_for\_suffix}\,(\alpha, j, \mathsf{s.pc}))$
where:
$\texttt{upcoming\_commands}(\mathsf{s}, c)$ *ensures that starting at address* $\mathsf{s.pc}$ *of the code memory of* s*, the commands* c *are allocated, and*
$\texttt{emulate\_responses\_for\_suffix}(\alpha, j, \ldots)$ *is defined recursively on* j *to be a nested switch statement:*

$\texttt{switch}(\mathsf{current\_trace\_idx})$ {
$\texttt{case}\ j:$
  $\texttt{emulate\_ith\_action}(\alpha, j, \ldots);$
  $\texttt{emulate\_responses\_for\_suffix}(\alpha, j+2, \ldots)$
$\texttt{case}\ j+2:$
  $\texttt{emulate\_ith\_action}(\alpha, j+2, \ldots);$
  $\texttt{emulate\_responses\_for\_suffix}(\alpha, j+4, \ldots)$
$\ldots$
$\texttt{case}\ |\alpha| - 1:$

```
    emulate_ith_action(α, |α| − 1, ...);
}
```

(See definitions 120 and 121 in the technical report for the formal definitions.)
(Notice that the maximum depth of the nesting of the switch statement is $(|\alpha| - j)/2$, i.e., half the length of the suffix starting at position j—the half originates from the fact that we are only emulating every other trace label.)
(Notice also that, interestingly, most of the code generated by `emulate_responses_for_suffix` *is actually unreachable; for a finite trace α, the size of the code generated is $O(|\alpha|^2)$, of which only $O(|\alpha|)$ is reachable. However, this extra code simplifies some proofs.)*
(Observe that `emulate_ith_action` *generates the code in the back-translation example of Listing V.1—lines 7 to 12)*

Lemma 7 (Compiler forward-simulation lifted to compressed trace steps). $\mathsf{s_s} \cong_{\mathsf{p}} \mathsf{s_t} \wedge (\mathsf{s_s}, \varsigma) \xrightarrow{\alpha}_{\mathsf{p}} (\mathsf{s'_s}, \varsigma') \implies \exists \mathsf{s'_t}.\ (\mathsf{s_t}, \varsigma) \xrightarrow{\alpha}_{[\![\mathsf{p}]\!]} (\mathsf{s'_t}, \varsigma') \wedge \mathsf{s'_s} \cong_{\mathsf{p}} \mathsf{s'_t}$

Lemma 8 (Compiler backward-simulation lifted to compressed trace steps). $\mathsf{s_s} \cong_{\mathsf{p}} \mathsf{s_t} \wedge (\mathsf{s_t}, \varsigma) \xrightarrow{\alpha}_{[\![\mathsf{p}]\!]} (\mathsf{s'_t}, \varsigma') \implies \exists \mathsf{s'_s}.\ (\mathsf{s_s}, \varsigma) \xrightarrow{\alpha}_{\mathsf{p}} (\mathsf{s'_s}, \varsigma') \wedge \mathsf{s'_s} \cong_{\mathsf{p}} \mathsf{s'_t}$

Lemma 9 (Lock-step simulation of strong similarity). $(\mathsf{s_1}, \varsigma) \approx_{\mathsf{p}} (\mathsf{s_2}, \varsigma) \wedge (\mathsf{s_1}, \varsigma) \xrightarrow{\tau}_{\mathsf{p}} (\mathsf{s'_1}, \varsigma) \implies \exists \mathsf{s'_2}.\ (\mathsf{s_2}, \varsigma) \xrightarrow{\tau}_{\mathsf{p}} (\mathsf{s'_2}, \varsigma) \wedge (\mathsf{s'_1}, \varsigma) \approx_{\mathsf{p}} (\mathsf{s'_2}, \varsigma)$

Lemma 10 (Strong similarity is weakened by an output action). $\lambda \in \dot{!} \wedge (\mathsf{s_1}, \varsigma) \approx_{\mathsf{p}} (\mathsf{s_2}, \varsigma) \wedge (\mathsf{s_1}, \varsigma) \xrightarrow{\lambda}_{\mathsf{p}} (\mathsf{s'_1}, \varsigma') \implies \exists \mathsf{s'_2}.\ (\mathsf{s_2}, \varsigma) \xrightarrow{\lambda}_{\mathsf{p}} (\mathsf{s'_2}, \varsigma') \wedge (\mathsf{s'_1}, \varsigma') \sim_{\mathsf{p}} (\mathsf{s'_2}, \varsigma')$

Lemma 11 (Option simulation of weak similarity). $(\mathsf{s_1}, \varsigma) \sim_{\mathsf{p}} (\mathsf{s_2}, \varsigma) \wedge (\mathsf{s_1}, \varsigma) \xrightarrow{\tau}_{\mathsf{p}} (\mathsf{s'_1}, \varsigma) \implies (\mathsf{s'_1}, \varsigma) \sim_{\mathsf{p}} (\mathsf{s_2}, \varsigma)$

Lemma 12 (Weak similarity is strengthened by aligned input actions). $\lambda \in \dot{?} \wedge (\mathsf{s_1}, \varsigma) \sim_{\mathsf{p}} (\mathsf{s_2}, \varsigma) \wedge (\mathsf{s_1}, \varsigma) \xrightarrow{\lambda}_{\mathsf{p}} (\mathsf{s'_1}, \varsigma') \wedge (\mathsf{s_2}, \varsigma) \xrightarrow{\lambda}_{\mathsf{p}} (\mathsf{s'_2}, \varsigma') \implies (\mathsf{s'_1}, \varsigma') \approx_{\mathsf{p}} (\mathsf{s'_2}, \varsigma')$

Lemma 13 (Adequacy of emulate_invariants). $\texttt{emulate\_invariants}\,(\mathsf{s})_{\alpha,i,\mathsf{p}} \wedge \alpha(i) \in \dot{?} \cup \{\checkmark\} \implies \exists \mathsf{s'}.\ (\mathsf{s}, \_) \xrightarrow{\alpha(i)}_{\mathsf{p}} (\mathsf{s'}, \_)$

Lemma 14 (Preservation of emulate_invariants). $\texttt{emulate\_invariants}\,(\mathsf{s})_{\alpha,i,\mathsf{p}} \wedge (\mathsf{s}, \_) \xrightarrow{\alpha(i)}_{\mathsf{p}} (\mathsf{s'}, \_) \implies \texttt{emulate\_invariants}\,(\mathsf{s'})_{\alpha,i+1,\mathsf{p}}$

Using these lemmas, we prove Lemma 6 as follows.

Proof. (of Lemma 6; simplified)
Hypotheses (unfolding Definition 4):

1) $\alpha \in \texttt{Alt}$
2) $\mathsf{s_{emu}} \cong_{\mathsf{p_s}} \mathbf{s_{med}}$
3) $\texttt{emulate\_invariants}\,(\mathsf{s_{emu}})_{\alpha,i,\mathsf{p_s}}$
4) $\alpha(i) \in \dot{!} \implies (\mathbf{s_{med}}, \varsigma) \approx_{[\![\mathsf{p_s}]\!]} (\mathbf{s_{given}}, \varsigma)$
5) $\alpha(i) \in \dot{?} \implies (\mathbf{s_{med}}, \varsigma) \sim_{[\![\mathsf{p_s}]\!]} (\mathbf{s_{given}}, \varsigma)$
6) $(\mathbf{s_{given}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{given}}, \varsigma')$

We consider two cases for the trace step $\alpha(i)$, which we obtain by unfolding the definition of `Alt` in hypothesis 1 (we ignore in this proof sketch the case of $\alpha(i) = \checkmark$):

- Case $\alpha(i) \in \dot{?}$: (**4 subgoals**)
  (a) Obtain $\mathsf{s'_{emu}}$ where $(\mathsf{s_{emu}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathsf{s'_{emu}}, \varsigma')$ as follows: Instantiate Lemma 13 (adequacy of the emulation invariants) with both hypothesis 3 and the case condition to obtain $\mathsf{s'_{emu}}$ that satisfies $(\mathsf{s_{emu}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathsf{s'_{emu}}, \varsigma'')$ for some ς''. Then, by inversion of this step (and unfolding the computation of the informative label), we know $\varsigma'' = \varsigma'$.
  (b) Obtain $\mathbf{s'_{med}}$ where $(\mathbf{s_{med}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{med}}, \varsigma')$ and $\mathsf{s'_{emu}} \cong_{\mathsf{p_s}} \mathbf{s'_{med}}$ as follows: Instantiate Lemma 7 with both hypothesis 2 and statement (a) from above.
  (c) Obtain strong similarity $(\mathbf{s'_{med}}, \varsigma) \approx_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{given}}, \varsigma)$ as follows: First, unfold both statement (b) and hypothesis 6 to obtain respectively the two states $\mathbf{s''_{med}}$ and $\mathbf{s''_{given}}$ just before the border crossing. Now instantiate Lemma 11 (option simulation) with hypothesis 5 twice, once with $\mathbf{s''_{med}}$ for $\mathbf{s'_1}$ and once with $\mathbf{s''_{given}}$ for $\mathbf{s'_1}$ (use symmetry of weak similarity before the second instantiation). Now (by transitivity of weak similarity), we know that $(\mathbf{s''_{med}}, \varsigma) \sim_{[\![\mathsf{p_s}]\!]} (\mathbf{s''_{given}}, \varsigma)$. Thus, instantiate Lemma 12 (strengthening) to obtain $(\mathbf{s'_{med}}, \varsigma) \approx_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{given}}, \varsigma)$.
  (d) Obtain $\texttt{emulate\_invariants}\,(\mathsf{s'_{emu}})_{\alpha,i+1,\mathsf{p_s}}$ as follows: Instantiate Lemma 14 (preservation of the emulation invariants) with hypothesis 3 and statement (a) from above.
- Case $\alpha(i) \in \dot{!}$: (**3 subgoals**)
  (a) Obtain $\mathbf{s'_{med}}$ where $(\mathbf{s_{med}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{med}}, \varsigma')$ and $(\mathbf{s'_{med}}, \varsigma) \sim_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{given}}, \varsigma)$ as follows: First, unfold the definition of the given step of hypothesis 6 to obtain star-many τ steps leading to some state $\mathbf{s''_{given}}$ where also $(\mathbf{s''_{given}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{given}}, \varsigma')$. Now instantiate (the star version of) Lemma 9 (lock-step simulation) with both hypothesis 4 (after applying symmetry of strong similarity) and state $\mathbf{s''_{given}}$ (the τ steps) to obtain some $\mathbf{s''_{med}}$ where $(\mathbf{s''_{given}}, \varsigma) \approx_{[\![\mathsf{p_s}]\!]} (\mathbf{s''_{med}}, \varsigma)$. Next, use this together with the step $(\mathbf{s''_{given}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathbf{s'_{given}}, \varsigma')$, which we obtained above, to instantiate Lemma 10 (weakening). Next, apply symmetry of weak similarity to obtain the subgoal.
  (b) Obtain $\mathsf{s'_{emu}}$ where $(\mathsf{s_{emu}}, \varsigma) \xrightarrow{\alpha(i)}_{[\![\mathsf{p_s}]\!]} (\mathsf{s'_{emu}}, \varsigma')$ and $\mathsf{s'_{emu}} \cong_{\mathsf{p_s}} \mathbf{s'_{med}}$ as follows: Instantiate Lemma 8 (backward simulation) using both hypothesis 2 and the step of $\mathbf{s_{med}}$ that we obtained in the previous subgoal.
  (c) Obtain $\texttt{emulate\_invariants}\,(\mathsf{s'_{emu}})_{\alpha,i+1,\mathsf{p_s}}$ as follows: Instantiate Lemma 14 (preservation of the emulation invariants) with both hypothesis 3 and the step of $\mathsf{s_{emu}}$ that we obtained in the previous subgoal.

∎