JFP **31**, e6, 55 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S0956796821000022

Linear capabilities for fully abstract compilation of separation-logic-verified code

THOMAS VAN STRYDONCK AND FRANK PIESSENS
KU Leuven, Leuven, Belgium
(*e-mails:* thomas.vanstrydonck@cs.kuleuven.be, frank.piessens@cs.kuleuven.be)

DOMINIQUE DEVRIESE
Vrije Universiteit Brussel, Brussels, Belgium
(*e-mail:* dominique.devriese@vub.be)

Abstract

Separation logic is a powerful program logic for the static modular verification of imperative programs. However, *dynamic* checking of separation logic contracts on the boundaries between verified and untrusted modules is hard because it requires one to enforce (among other things) that outcalls from a verified to an untrusted module do not access memory resources currently owned by the verified module. This paper proposes an approach to dynamic contract checking by relying on support for capabilities, a well-studied form of unforgeable memory pointers that enables fine-grained, efficient memory access control. More specifically, we rely on a form of capabilities called *linear* capabilities for which the hardware enforces that they cannot be copied. We formalize our approach as a fully abstract compiler from a statically verified source language to an unverified target language with support for linear capabilities. The key insight behind our compiler is that memory resources described by spatial separation logic predicates can be represented at run time by linear capabilities. The compiler is *separation-logic-proof-directed*: it uses the separation logic proof of the source program to determine how memory accesses in the source program should be compiled to linear capability accesses in the target program. The full abstraction property of the compiler essentially guarantees that compiled verified modules can interact with untrusted target language modules as if they were compiled from verified code as well. This article is an extended version of one that was presented at ICFP 2019 (Van Strydonck *et al.*, 2019).

1 Introduction

Separation logic is the basis for tools that support sound, modular verification of C programs, such as VeriFast (Jacobs *et al.*, 2010). However, for such verification to be sound for a whole program, *all* modules of the program have to be verified (Agten *et al.*, 2015).

In this paper, we are concerned with scenarios where verified code interacts with untrusted code (e.g. when installing plug-ins from the internet). Our goal is to compile the verified code securely, that is, in such a way that we can preserve the guarantees obtained from verification, even under this interaction with untrusted code. To achieve this, the compiler has to dynamically enforce separation logic contracts on the boundary between verified and untrusted code.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

As a concrete example of our approach, consider a separation-logic-verified video player that runs locally on a user's computer and allows for the installation of untrusted and unverified plug-ins to extend its functionality. An example plug-in would be a codec (*coder-decoder*), that includes support for the decompression of specific video encodings before display. The separation logic contract for the decompression function of this plug-in could, for instance, provide it access to the buffer where the compressed video is stored, but forbid it from retaining references to this buffer afterward. The contract for the decompression function might then informally (we elided functional assertions) look as follows:

```
void decompress(char* b, format f)
//@pre   b ↦ contents_pre * ...
//@post  b ↦ contents_post * ...
```

The decompress function receives a pointer to the buffer b and is supposed to decompress b using the proper codec for format f. In the precondition //@pre, decompress receives ownership of the buffer's contents (represented by the points-to chunk $\mapsto$), so that it can perform decompression in-place (assuming sufficient buffer size). In the postcondition //@post, decompress returns ownership over the contents of this buffer and should consequently lose its access to it. However, this revocation of access is hard to enforce dynamically: an unverified plug-in could, for example, copy and store the reference to b and use it to freely read and write to it later, even after returning control. Plug-ins can violate their contracts in many other ways: they can deviate from their specified behavior, while they legitimately hold references to the internal state (e.g. decompress could use the wrong format, or do nothing); they might read or write outside the intended ranges of the references they are provided with (e.g. perform a buffer overread or -write outside of b's bounds) or might return incorrect values. In the current state of the art, separation-logic verification guarantees are not enforced at run time for partially verified programs.

To perform dynamic checking of separation logic contracts *efficiently*, some form of hardware support for memory protection is required. Agten *et al.* (2015) proposed an approach for dynamic checking of contracts based on a hardware protection primitive known as *protected module architectures* (Strackx *et al.*, 2010; Noorman *et al.*, 2013; Costan & Devadas, 2016). However, they only provide run-time preservation of integrity guarantees, not of confidentiality, meaning that non-verified adversaries could still read data they should not contractually be allowed to read. Hence, they do not ensure *full abstraction*: a formal property that is often used to define secure compilation (Abadi, 1999). This property requires that attacker code interacting with the compiled code in the target language should not have more power than arbitrary verified code interacting with the verified source code.

The main contribution of this paper is the development of a *fully abstract* compiler that dynamically enforces separation logic contracts by relying on another kind of hardware support. Our approach relies on support for *capabilities*: a well-studied form of unforgeable memory pointers that are in essence regular pointers enhanced with a field containing privileges (read, write, execute, ...) and fields describing a memory range these privileges can be exerted on. Capabilities allow for fine-grained, efficient memory access control and are implemented in special processors called *capability machines* (see Levy, 1984, for an

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

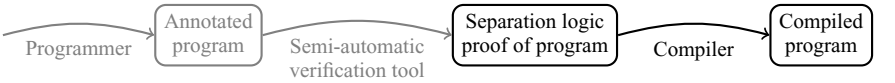


Fig. 1. A usage model of our compiler.

overview). The CHERI processor (Watson *et al.*, 2015) is an example of a recent design for a capability machine.

More specifically, we rely on a form of capabilities called *linear capabilities*. Linear capabilities are specially treated by the hardware to ensure that they can never be copied. They are related to, but different from, CHERI's local capabilities (which can, essentially, only be stored in registers or on the stack, not in memory) and should be implementable efficiently in capability machines. Skorstengaard *et al.* (2019) have previously used them in the secure calling convention `StkTokens`, and an early design for their implementation in CHERI is given in the latest CHERI ISA Spec (Watson *et al.*, 2020). Our compiler only requires basic linear capabilities with read/write permissions.

The key insight of our approach is that memory resources (described by spatial separation logic predicates) can be represented at run time as linear capabilities. Hence, transferring ownership of memory resources to another module on function call or return can be compiled to passing the corresponding linear capabilities as additional parameters and/or return values. Compiler-generated stubs, on the boundaries between verified and unverified code, can then dynamically check separation logic contracts. However, those linear capabilities cannot simply be substituted for regular pointers in existing programs, as they behave fundamentally differently (because of their hardware-enforced non-duplicability). We can pass them in addition to regular pointers, but then the difficulty is that every memory access in the original program (through a regular pointer) needs to happen through an appropriate linear capability in the compiled program. Since those linear capabilities are not necessarily in one-to-one correspondence with the regular pointers that the program works with, it is not clear how the compiler can decide which one to use.

However, for verified programs, this information is apparent from the separation logic proof of verified code. In that proof, every memory access is justified using a single memory resource, and by carefully tracking the capability corresponding to every such resource, we can decide which capability to use. Hence, our compiler is *separation-logic-proof-directed*: it requires not simply the raw source code, but a separation logic proof of this code as input, and uses the information in the proof to generate correct and secure target code. Although it does not matter to our compiler where this proof comes from, we envision scenarios as depicted in Figure 1, where the programmer writes a program with contracts and minimal annotations, a semi-automatic verification tool like VeriFast (Jacobs & Piessens, 2008) elaborates these into a full proof, and this proof is then passed to our compiler. To be clear, only the last (compilation) step is the topic of this paper, although we sometimes use VeriFast notations in our examples, for readability.

We clarify upfront that our goal is not gradual verification (Agten *et al.*, 2015; Bader *et al.*, 2018), that is, we do not support taking a large codebase, verifying parts of it and securely combining the verified and unverified parts. Instead, our compiled code can securely interact with arbitrary untrusted code (e.g. the downloaded plug-ins above). However, our compiled code will only interact correctly with other code that respects our calling convention: we only allow linking to untrusted code if it respects our compiled

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

interfaces, that is, sends and receives linear capabilities encoding memory resources as extra arguments and return values when handling incalls from or outcalls to our code.

Moreover, this paper contains but the first steps toward a practically applicable secure compilation scheme, since the power of the separation-logic-verified source language is limited. Concretely, the source language only consists of simple resources in the separation logic, has a simple type system and features restrictions on the separation logic contracts for calls to and from untrusted code. Perhaps most notably, our source language does not yet support a notion of C-like structures; it does not support any type of recursive data structures. Additionally, the separation logic does not support the abstract predicates that would be required to reason about such data structures. These restrictions and how to alleviate them will be further detailed in the future work section.

In summary, the contributions of this paper are:

- a novel approach to compile separation-logic-verified C code to linear-capability-enhanced unverified C code that dynamically checks the contracts at the boundaries of the verified code; we demonstrate our approach for an essential separation logic with array resources and explain how the approach might be extended to more advanced logics;
- a formalization on a model of C, and a proof that our compiler is fully abstract; and
- a new use case for linear capabilities in capability machines like CHERI.

This paper is structured as follows. The compiler is illustrated in Section 2. Section 3 discusses the compiler's source and target language. Section 4 formally defines the separation logic rules and the full compiler. Section 5 formally defines full abstraction and discusses our proof approach, motivating the need for a new target-to-source transformation called the back-translation. This back-translation is illustrated by means of an example in Section 6. Having discussed both an example of compilation and back-translation, Section 7 zooms in on one part of the full abstraction proof, namely how we simulate source versus target code. Sections 8 and 9, respectively, discuss future and related work. Section 10 concludes.

This paper is an extended version of one that was previously published and presented at ICFP 2019 (Van Strydonck *et al.*, 2019). The most important additions are:

- A better explanation of how linear capabilities work in the target language operational semantics (Section 3.3).
- The new Section 5.3 which explains the role played by two relations R and S that relate source and target code in the proof of full abstraction.
- A rewritten Section 6 explaining the back-translation in a more complete and more pedagogical way.
- Finally, a new Section 7 which further decomposes the aforementioned relations R and S, showcasing how their components, respectively, prove the correctness and security directions of full abstraction through simulation.

Nevertheless, in this paper, we have still omitted details and simplified inference rules to maintain readability and ease of understanding on multiple occasions. Interested readers can find full details and the entire full abstraction proof in a 120-page technical report submitted as Supplementary Material (available at: http://dx.doi.org/10.1017/S0956796821000022) (Van Strydonck *et al.*, 2020).

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

	Verified Component	Outcall Stub	Context Declaration

Source	void f(int∗ a)
	//@pre n: a ↦ [0]
	//@post n: a ↦ [1] {
	g(a);
	a[0] = 1
	}
			void g(int∗ a)
			//@pre n: a ↦ [0]
			//@post n: a ↦ [0]

Target	int∗ f(int∗$_0$ a,int∗ n) {
	n = g$_{stub}$(a,n);
	n[0] = 1;
	return n
	}
		int∗ g$_{stub}$(int∗$_0$ a,int∗ n) {
		n = g(a,n);
		guard(n != null); guard(length(n) == 1);
		guard(a == addr(n)); guard(n[0] == 0);
		return n
		}
			int∗ g(int∗$_0$ a,int∗ n)

Fig. 2. Motivating example: a verified function f interacts with an untrusted context function g.

2 Compiler illustration

Figure 2 illustrates the operation of our compiler with a trivial example in the C-like syntax (which includes separation logic annotations) we employ. As suggested in the introduction, the target language of the compiled code is actually again C-like and not assembly. The reason is that the compilation from C to assembly would require the (fully abstract) compilation of many concepts (e.g. function calls, stack accesses) which are orthogonal to the topic of this paper and in at least one case already covered elsewhere (Skorstengaard *et al.*, 2019).

The example contains the verified source function f, which performs an outcall to the context identity function g and afterward sets the contents of pointer a from 0 to 1. Function f only knows g's separation logic contract and not its implementation.

To see how function declarations (including separation logic contracts) are compiled, we look at the function f (g is similar). The void function f takes a single argument: an integer pointer variable a. The precondition states that f receives a memory resource n to read and write pointer a in the heap, where a points to the single element array [0]. The resource can be seen as a (heap) permission. When f returns, its postcondition states that f hands back this permission, but a will contain the value 1. The verification of f proves that f upholds this contract.

Having introduced the example, the remainder of this section discusses how we compile it in a *separation-logic-proof-directed* way. By *separation-logic-proof-directed compilation*, we mean that our compiler uses the separation logic proof of the source program to guide compilation, in order to ensure that the compiled code enforces the separation logic contracts. Concretely, our compiler combines three separate techniques to dynamically enforce all information contained in separation logic contracts:

1. The *shape* of separation logic resources (i.e. their linearity and bounds, but not their functional aspects such as contents) is enforced by reifying separation logic resources to linear capabilities.
2. The *contents* of these linear capabilities are updated (but *not* checked) in parallel with the resources they were reified from by having compiled source statements manipulate capabilities.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

3. The actual *checking* of all aspects of contracts (i.e. the shape and contents of separation logic resources originating from adversaries, but also other non-spatial constraints) is performed by so-called checking functions or stubs at trust boundaries.

We now further detail these three aspects of proof-directedness in order.

First, we want to enforce the shape of separation logic resources at the target level, that is, make sure that resources to access the heap are handled linearly and bounds-checked at the target level. The reason we care about the semantics of these resources is that, for example, resource n for address a determines whether f can access a's contents during the proof, not a itself. This fact is reflected in the compilation, where the heap resource n is reified into a target-level linear capability int$*$ n. Although separation logic resources exist only conceptually and are not represented in the source language, they are transformed into *real* target-level program variables, as concrete instantiations of the source-level permissions. The semantics of the linear capability n corresponds very well to the meaning of the heap resource n in the following ways:

- The fact that the reified resource int$*$ n is a *capability* ensures that anyone owning it cannot read outside of its intended bounds; the single location a in this case.
- The *linearity* of int$*$ n constitutes a guarantee to its owner, that they are the sole owner of the permission n to access a.

These reified resources are manipulated in parallel with the source-level resources and represent the otherwise erased separation logic proof guarantees. The precondition of f mentions that it receives the resource n at the start of execution. This resource is reified as an extra argument int$*$ n. The postcondition of f requires the resource n to be returned, so analogously, the reified version of n is returned in the compilation.

The source-language argument a, on the other hand, is compiled to a length-0 regular capability int$*_0$ a. The length-0 capability type int$*_0$ denotes a non-linear capability of type int that cannot be dereferenced (this is in effect just an address). The reason int$*_0$ a is kept is for performing address operations and checks, as these non-spatial manipulations require no separation-logic resources.

Representing both addresses and resources at run time introduces a certain duplication of information in cases where addresses coincide with the bounds of the resources. This is the case, for example, in Figure 2, where the address a always coincides with the start address of the resource $a \mapsto [\_]$. However, in general, there is not necessarily a one-to-one connection between addresses and resources for them. It is hence impossible to just discard the pointer int$*_0$ a and perform address operations and checks on the reified resource n directly. For example, the function f could create multiple aliases for a through an assignment $b = a$, and it would not be clear which address to track in the reified resource. For the same reason, we cannot simply erase heap resources and compile the source pointer a to a linear target pointer a. In this case too, it would not be clear what to do when aliases are created for a. The problem in both cases is precisely that a is not a resource that should be handled linearly, but a regular non-linear program variable. Compiling both a and its aliases to non-linear length-0 capabilities and separately reifying the resources to linear capabilities instead avoid this mismatch. By separately representing addresses and resources, we can keep our compiler general and uniform. In a subsequent phase, a

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

compiler with a sufficiently clever static analysis engine could detect duplication of information and, for example, remove addresses of type $\text{int}*_0$ when their value can always be recomputed from a corresponding linear capability.

Second, the introduction of linear capabilities to represent separation logic resources, combined with the fact that the capabilities need to have the correct contents when passed to the context as resources, again requires our compiler to be separation-logic-proof-directed. The compiler inspects the verification proof to see how a source statement affects the state of the separation logic resources and mirrors the change to these resources in the compiled version of this statement. For example, setting a to 1 in f is compiled to setting the reified resource for a, n, to 1. The call to g now also receives and returns the resource n along with the address a. In summary, separation-logic-proof-directedness entails that operations performed on pointers in the source language are performed on the reified resources corresponding to these pointer's resources in the target language, as these resources are what justify these operations in the separation logic verification in the first place.

Third, the interaction between f and g at the trust boundary of the compiled component requires a checking function or stub (in this case an outcall stub for the outcall) that wraps g and verifies that it does indeed uphold its postcondition. This is necessary since f can outcall arbitrary compiled code g that might or might not uphold the contract that f expects of g. The postcondition of g says that it returns a resource n for address a with single-element contents 0. These conditions correspond exactly to the four guard statements in the outcall stub of Figure 2. If the verified component was to export f, allowing it to be called by untrusted code, then the compiled component would additionally contain an incall stub, to verify that f's precondition is met at call-time. To simplify the generation of stub functions, and thereby limit the size of the proofs, we have placed extra restrictions on contracts of functions at trust boundaries. These restrictions are described in more detail in Section 4.4, and ways to alleviate them are discussed in the future work section.

3 Source and target languages

The separation logic and the source and target languages are first discussed in Section 3.1. Section 3.2 then introduces notation to extend source language programs to source language proofs (as our compiler is separation-logic-proof-directed). Section 3.3 finally briefly discusses the operational semantics (including memory model) of the source and target languages.

In this paper, k denotes an integer, id_ℓ a logical variable, id_p a program variable, f a function and n a heap resource. Logical and program variables are considered to have separate namespaces.

3.1 Source and target language definition

The formalization of our separation logic assertions and the source and target languages is given in order of structural complexity by the BNF grammar in Figure 3, where the notation *symbol*⟨*parameter*⟩ is used for parameterized symbols. The concrete notation for separation logic annotations and assertions is inspired by the VeriFast tool (Jacobs *et al.*,

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

$\tau ::=$ int $\mid \tau* \mid (\tau, \ldots, \tau) \mid \text{list}_\tau$ (LOGICAL TYPE)
$\tau_s ::=$ int $\mid \tau_s* \mid (\tau_s, \ldots, \tau_s)$ (SOURCE TYPE)
$\tau_t ::=$ int $\mid \tau_t* \mid (\tau_t, \ldots, \tau_t) \mid \tau_t*_0$ (TARGET TYPE)

$cexp\langle exp \rangle ::= k \mid \text{op1}\, exp \mid exp\, \text{op2}\, exp \mid \text{null} \mid (exp, \ldots, exp) \mid exp.k$ (COMMON EXPRESSIONS)
$exp ::= id_\ell \mid cexp\langle exp \rangle \mid \text{length}(exp) \mid exp[exp] \mid \forall id_\ell : \tau.exp \mid \exists id_\ell : \tau.exp \mid \text{repeat}(exp, exp) \mid$
$\text{append}(exp, exp) \mid \text{take}(exp, exp, exp) \mid \text{update}(exp, exp, exp)$ (LOGICAL EXPRESSIONS)
$sexp ::= id_p \mid cexp\langle sexp \rangle$ (SOURCE EXPRESSIONS)
$texp ::= id_p \mid cexp\langle texp \rangle \mid \text{addr}(texp) \mid \text{length}(texp)$ (TARGET EXPRESSIONS)

$cassert\langle assert \rangle ::= exp \mid assert * assert \mid exp \,?\, assert$ (COMMON ASSERTIONS)
$assert ::= n : exp \mapsto_{\tau_s} exp$ *array resource*
$\mid n : [assert_{in} \mid exp \leq id_\ell < exp]$ *range resource*
$\mid cassert\langle assert \rangle$ (OUTER SEPARATION LOGIC ASSERTION)
$assert_{in} ::= exp \mapsto_{\tau_s} exp$ *array resource*
$\mid [assert_{in} \mid exp \leq id_\ell < exp]$ *range resource*
$\mid cassert\langle assert_{in} \rangle$ (INNER SEPARATION LOGIC ASSERTION)

$sstm ::=$ skip $\mid id_p = \text{malloc}(sexp * \text{sizeof}(\tau_s)) \mid \text{foreach}(sexp \leq i < sexp)\{sstm\} \mid sstm; sstm$
$\mid \text{if}\, sexp\, \text{then}\, sstm\, \text{else}\, sstm \mid \tau\ id_p \mid id_p = sexp \mid (id_p^*) = f(sexp^*) \mid \text{guard}(sexp)$
$\mid id_p[sexp] = sexp \mid id_p = sexp[sexp]$ *regular statements*
$\mid //@\text{split}\ n[sexp] \mid //@\text{join}\ n\ n \mid //@\text{flatten}\ n \mid //@\text{collect}\ n^* \cdot \ldots \cdot n^*$ *ghost statements*
(SOURCE STATEMENTS)
$tstm ::=$ skip $\mid id_p = \text{malloc}(texp * \text{sizeof}(\tau_t)) \mid \text{foreach}(texp \leq i < texp)\{tstm\} \mid tstm; tstm$
$\mid \text{if}\, texp\, \text{then}\, tstm\, \text{else}\, tstm \mid \tau\ id_p \mid id_p = texp \mid (id_p^*) = f(texp^*) \mid \text{guard}(texp)$
$\mid id_p[texp] = texp \mid id_p = texp[texp]$ *regular statements*
$\mid (id_p, id_p) = \text{split}(id_p, texp) \mid id_p = \text{join}(id_p, id_p)$ *built-in functions*
(TARGET STATEMENTS)

$isfunc ::= \tau_s^*\ f((\tau_s\ id)^*)\ //@\text{pre}\ assert\ //@\text{post}\ assert\ \{sstm; \text{return}\ sexp^*\}$
(IMPLEMENTED SOURCE FUNCTION)
$csfunc ::= \tau_s^*\ f((\tau_s\ id)^*)\ //@\text{pre}\ assert\ //@\text{post}\ assert$ (CONTEXT SOURCE FUNCTION)
$itfunc ::= \tau_t^*\ f((\tau_t\ id)^*)\ \{tstm; \text{return}\ texp^*\}$ (IMPLEMENTED TARGET FUNCTION)
$ctfunc ::= \tau_t^*\ f((\tau_t\ id)^*)$ (CONTEXT TARGET FUNCTION)

$scomp ::= isfunc^+\ //@\text{import}\ csfunc^*\ //@\text{export}\ csfunc^*$ (SOURCE COMPONENT)
$tcomp ::= itfunc^+\ //@\text{import}\ ctfunc^*\ //@\text{export}\ ctfunc^*$ (TARGET COMPONENT)

$sprog ::= scomp^+\ //@\text{main} = id$ (SOURCE PROGRAM)
$tprog ::= tcomp^+\ //@\text{main} = id$ (TARGET PROGRAM)

Fig. 3. Grammar describing our separation logic and the source and target languages.

2010). For the features and syntax of the source language, we drew some inspiration from Clight, one of the intermediate languages used in CompCert (Leroy, 2006; Leroy *et al.*, 2016).

Both the source and target languages (i.e. the program domain) build statements *sstm*/*tstm* out of expressions *sexp*/*texp*, components *scomp*/*tcomp* out of functions containing statements and programs *sprog*/*tprog* out of components. The separation logic (i.e. the logical domain), used in source function contracts and separation logic proofs, builds its assertions *assert* out of symbolic expressions *exp*. The remainder of this section discusses the BNF grammar in order.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Types. To simplify types, the type bool is embedded in the type int, where 0 is true and $k \neq 0$ is false. The target type τ_t* is assumed linear, requiring value erasure (i.e. replacing the value with null) whenever such a value is copied (e.g. assigned to another variable, passed to a function, stored in an array, ...), whereas the source type τ_s* is a regular non-linear heap pointer. A type $(\tau, \ldots, \tau)$, representing length-n ($n \geq 0$) tuples, is present for all three cases. Pointers and arrays are seen as identical types $\tau*$ in our formalization, for simplicity's sake.

The target language has an extra type τ_t*_0 of length-0 non-linear capabilities, used by the compiler to store the compiled version of the source-level permissionless program variables. The separation logic has a type list_τ, which is a type of logical list variables. These variables are used to represent the contents of source-level arrays in resources.

Expressions. Separation logic makes a distinction between program variables id_p, which appear in source programs, and expressions *sexp* over them on the one hand and logical (or symbolic) proof-only variables id_ℓ, which only appear in separation logic contracts and proofs, and logical expressions *exp*, which are a third type of expression, on the other hand. Source expressions are the least expressive, allowing program variables and the common expressions only.

Target expressions *texp* additionally contain a function addr, which returns the τ_t*_0-type address of a τ_t*-type value, and a function "length", which returns the length of the region addressed by a linear capability τ_t*. These functions are realistic (since linear capabilities encode their own length and address information) and needed (for the contract checks performed in incall and outcall stubs).

Logical expressions contain extra functions to manipulate list_τ-typed values: a length function and an indexing construct $exp[exp]$. Universal and existential quantification allow constraining elements of non-statically sized logical lists. For readability, we also provide repeat, append, take and update list constructs, but these can be desugared to the other logical expression constructs. The repeat(exp_1, exp_2) construct returns a logical exp_1-length list where each element equals exp_2, append(exp_1, exp_2) appends lists exp_1 and exp_2, take(exp_1, exp_2, exp_3) constructs a new list from elements exp_2 up to (but not including) exp_3 of list exp_1 and update(exp_1, exp_2, exp_3) updates index exp_2 of list exp_1 with exp_3.

Assertions. Assertions are the building blocks of function contracts and separation logic proofs. Logical expressions *exp* are assertions, and so is the separating conjunction $*$ of two assertions. Conditional assertions of the form $exp\,?\,assert$ express that *assert* only needs to hold if $exp == \text{true}$ can be derived. Last, two types of assertions represent spatial resources: array and range resources. Array resources $exp_1 \mapsto_{\tau_s} exp_2$ represent an array at address exp_1, containing the elements of list exp_2. In order to talk about fixed-size array resources, we use the syntax $exp_1 \mapsto_{\tau_s} [exp_2^1, \ldots, exp_2^k]$, which desugars to $exp \mapsto_{\tau_s} l * \overline{l[i] = exp_2^i} * \text{length}(l) = k$ and is already demonstrated in Figure 2.

Range resources $[assert \mid exp_1 \leq id_\ell < exp_2]$ represent the separating conjunction of *assert* for each value from exp_1 to (but not including) exp_2, where *assert* usually depends on id_ℓ. Range resources can be nested and will be useful for the back-translation in Section 6.

An alternative design could have made use of a single, primitive points-to resource $a \mapsto_p v$, representing permissions to access the single value v at address a, instead of our array resources. All types of array resources could then be desugared in terms of this

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

primitive primitive resource as follows: $a \mapsto l \triangleq [a + i \mapsto_{\mathrm{p}} l[i] \mid 0 \leq i < \mathrm{length}(l)]$. This change should not impact the rest of the formalization in any major way.

Since both types of resources will be reified during compilation, as demonstrated for array resources in Section 2, we have to associate names n with both types. However, within a named range resource, no more names should occur, as the outer resource will be reified as a whole. This is the reason for the distinction between outer and inner resources in the grammar.

Statements. Statements in the target correspond one to one to statements in the source, except for the //@flatten and //@collect statements, which do not appear in the target language. The guard statement gets stuck during execution if its condition evaluates to false. The foreach statement executes its statement for every value of i in the given range. The foreach statement could technically be left out, since we have recursive calls, but it is kept for conciseness. Finally, both source and target languages have array assignment and array lookup statements.

The malloc statement used in the target language does not correspond to the vanilla malloc function in C. First, it returns a linear capability, not a regular pointer (since it essentially creates a target-level reified resource). Second, it guarantees a fresh heap location for the allocated variable. This avoids reuse of locations after free; if any newly allocated location could have been previously used by the context, it could have kept a reference to it and hereby broken the linearity guarantees. The target malloc statement hence respects *temporal safety*, an important desired property in any capability machine, even for non-linear capabilities (Watson *et al.*, 2015). In a practical implementation, freshness of the malloced heap locations could realistically be achieved by a form of garbage collection, much like in libgc (Insolvibile, 2003). Interestingly, the run-time bounds that capabilities inherently provide will allow for more precise garbage collection, instead of the conservative variant that libgc necessarily employs.

For simplicity reasons, we do not consider free (see Section 8 for further discussion).

In addition to these regular statements, there are also ghost statements. These operate on logical state instead of program state in the source language and are only relevant for the construction of the separation logic proof. VeriFast-style syntactic ghost statements are usually used as hints for a semi-automated proof tool during construction of the proof. We hence do not technically need them, since, as mentioned in the introduction, our compiler assumes a full separation logic proof as input (cfr. Figure 1). Instead, we could simply have separation logic rules manipulating ghost state without requiring corresponding syntactic constructs. For readability reasons in symbolic executions, and to make the source language correspond better to the target syntactically, we still use VeriFast-style syntactic ghost statements in this paper. Since separation logic resources are reified during compilation, ghost statements will have to be reified as well. Ghost statements and their target-level counterparts will hence be syntactically different between the source and target languages. We now discuss each ghost statement.

The source split statement splits a separation logic resource n into two at a given index *sexp*. For example, //@split $n[1]$ applied to the array resource $n : a \mapsto [1, 2, 3]$ splits this resource into $n' : a \mapsto [1]$ and $n'' : a + 1 \mapsto [2, 3]$. When applied to the range resource $n : [assert \mid 0 \leq i < 3]$, the same //@split $n[1]$ creates $n' : [assert \mid 0 \leq i < 1]$ and $n'' : [assert \mid 1 \leq i < 3]$. The source join statement is the inverse of split and merges two

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

adjacent resources into one, for example, both previous sets of resources n' and n'' are merged into n by //@join n' n''. As the resources that split and join operate on are reified, so are the operations themselves: built-in target functions to analogously split and join linear capabilities are provided in the target language.

The source flatten and collect statements are each other's inverse and, respectively, strip a top-level range resource or create it. This only works for statically sized range resources. For example, the resources $n' : a + 1 \mapsto [1]$ and $n'' : a + 2 \mapsto [2]$ can be combined into $n : [a + i \mapsto [i] \mid 1 \leq i < 3]$ by the statement //@collect $n' \cdot n''$, where the $\cdot$ is used to delimit the sequence of resource names for each individual index of the constructed range resource. Notice again how only the top-level resource is named. The postcondition in the proof specifies which exact range resource is created. Conversely, the statement //@flatten n creates resources n' and n'' from n.

Interestingly, the target language does *not* contain reified, built-in functions to flatten or collect linear capabilities. The reason is that ghost statements are the only way to manipulate resources in the source language, and flatten and collect statements to switch representations are hence required. In the target language however, all resources are reified to linear capabilities, which *can* be manipulated by target code. The flatten and collect statements can hence be compiled to regular target-level statements, obviating the need for built-in functions. For example, the effect of //@collect $n' \cdot n''$ could be realized by the compiled code: int$**$ n; $n = \text{malloc}(2 * \text{sizeof}(\text{int}*))$; $n[0] = n'$; $n[1] = n''$.

Functions. Two classes of functions exist: implemented and context functions. Implemented functions consist of both a function declaration and a body. Context functions solely consist of the declaration that a component expects of this function. For simplicity, tuple return types (τ^*) exist in both source and target languages and every function has to end in a single return statement return exp^*. Source language functions are annotated with separation logic contracts that use the separation logic assertions mentioned before for pre- and postcondition. As mentioned, contracts are situated in the separation logic domain and hence range over logical variables id_ℓ, not program variables id_p.

Components and programs. A sequence of implemented functions that uses context functions in its function import and export lists is called a component. A sequence of components with a main function *id* forms an entire program.

3.2 Source language proofs

As explained, our compiler is separation-logic-proof-directed, that is, not a regular source program, but its separation logic proof is the input to the compiler. In addition to the grammar defining the syntax of source language programs, we need a notation for separation logic proofs (in this subsection) and a set of inference rules that describe how to construct such proofs starting from the source code (Section 4). *Hoare triples* are the building blocks of separation logic rules (Reynolds, 2002).

Classical separation logic uses Hoare triples of the form $\{P\}\ c\ \{Q\}$. In this paper, they have a partial correctness semantics: $\{P\}\ c\ \{Q\}$ states that given precondition P, either postcondition Q holds after execution of the piece of source code c, or c diverges (Reynolds, 2002). A triple $\{P\}\ c\ \{Q\}$ is only provable if there exists a proof tree, constructed from the individual separation logic rules, that has this triple as the root. In our formalization

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

of separation logic, however, we split the condition P (and Q) into two separate parts, partly inspired by the approach of VeriFast (Vogels *et al.*, 2015). These parts are called the *symbolic heap* P and the *environment* γ and give rise to the extended Hoare triple notation $\{P\}_\gamma\ c\ \{Q\}_{\gamma'}$, stating that if (P, γ) holds, then either c diverges or (Q, γ') holds after execution of c. If $\gamma == \gamma'$, we shorten the Hoare triple notation to $\{P\}\ c\ \{Q\}$. The two aspects of extended Hoare triples and the triples themselves are defined by the following BNF grammar:

$$\begin{array}{l|l} \langle P \rangle ::= \mathit{assert} \quad \text{(SYMBOLIC HEAP)} & \langle c \rangle ::= \mathit{sstm} \\ \langle \gamma \rangle ::= \bullet[\mathit{id}_\mathrm{p} : \mathit{exp}]^* \quad \text{(ENVIRONMENT)} & \quad \mid \mathit{sstm}; \text{return } \mathit{sexp} \quad \text{(SOURCE CODE)} \\ & \langle \mathit{triple} \rangle ::= \{P\}_\gamma\ c\ \{P\}_\gamma \ \text{(HOARE TRIPLE)} \end{array}$$

We will often use this notation c independently, to denote a piece of source (or target) code.

The two parts, γ and P, of separation logic states have the following meaning:

- The *environment* γ maps program variables id_p to expressions *exp* over logical, proof-level variables id_ℓ. The environment γ hence relates the program domain to the logical domain.
- The *symbolic heap* P is a $*$-separated list of assertions representing the symbolic program state. It is of the same form *assert* as the contracts described in Figure 3.

Hoare triple syntax is only useful for verifying (parts of) function bodies. In Section 4, we verify entire functions, components and programs. Given *any* piece of source code $\mathfrak{s}$, be it (part of) a function (body), a component or a program, the notation $\vdash \mathfrak{s}$ represents a specific, valid separation logic proof tree for $\mathfrak{s}$. This proof $\vdash \mathfrak{s}$ is what our proof-directed compiler uses as input.

3.3 Operational semantics

We define C-style small-step operational semantics for both the source and target languages. The operational semantics rules in both languages are of the following form:

$$\frac{\mathit{Premise}}{\langle \overline{s}, h \rangle \mid \overline{c} \hookrightarrow \langle \overline{s'}, h' \rangle \mid \overline{c'}} \ \text{(RULENAME)}$$

The small-step operational semantics $\hookrightarrow$ transform a program state $\langle \overline{s}, h \rangle \mid \overline{c}$ into a state $\langle \overline{s'}, h' \rangle \mid \overline{c'}$, where $\overline{s}$ is the list of stack frames containing local variables, h is the heap, and $\overline{c}$ is a list of partly executed function bodies, separated by return statements, where the sequence $\overline{c}$ corresponds to the sequence of stack frames $\overline{s}$. A function call creates a new stack frame and accompanying executing function body and adds them to $\overline{s}$ and $\overline{c}$, respectively. A return statement erases one of each. Erroneous programs get stuck because no operational semantics rules apply to them. The same happens for false guard statements.

One special case should be considered; at the very start of execution, $\overline{c}$ is a single, monolithic source or target program *prog*. Execution of a full program *prog* starts off by executing the function whose *id* is given in //@main = *id* , thereby creating the first stack frame. Since we are interested in the termination behavior of our programs, rather than their exact output, we require the main function *id* to have return type void. Main functions are

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

not allowed to have arguments either. The initial state before calling the main function is $\langle \bullet, \bullet \rangle \mid prog$: both the stack and heap are empty, denoted by $\bullet$. We say that *prog* terminates, denoted $prog \Downarrow$, if a sequence of small-step transitions exists that reduces the program to a single return statement with no arguments (i.e. the original return statement of the main function). We can then define termination formally as follows:

$$prog \Downarrow \triangleq \exists s, h, \overline{exp}.\ \langle \bullet, \bullet \rangle \mid prog \hookrightarrow^* \langle s, h \rangle \mid \text{return}$$

Note that termination does not include getting stuck.

The memory model for both source and target languages is location-based, that is, addresses are pairs (l, i) of an opaque location and an index i and hence $h \in (\text{Loc}, \text{Index}) \xrightarrow{\text{fin}} \text{Val}$. A malloc statement that allocates k units of type τ creates a new location l in the heap, populated with default values (out of simplicity considerations) for type τ at indexes 0 through $k - 1$. The default value for pointers is the null pointer, which implies that it is currently impossible to have a non-nullable pointer type in the source or target language. A possible solution to avoid this technical limitation would be to stick closer to the C semantics by not specifying a default value and making a dereference of an uninitialized pointer undefined behavior instead. Separately malloced variables are hence logically separate.

Source pointer values of type τ_s* follow the heap memory model and are hence denoted as either null, in case of the null pointer, or pairs (l, i). Target-level linear capabilities τ_t*, on the other hand, are denoted as either null or $l^{[a,b]}$, where $[a, b]$ is the closed interval of indexes at location l that they carry authority over. They do not need an index i, as source pointer arithmetic is compiled to target pointer arithmetic on length-0 capabilities and not on linear capabilities (see Section 2). A capability value $l^{[a,b]}$ hence always points to index a. Target-level length-0 capabilities are represented by l_0^i-values, which do not carry any authority, but keep an index i for pointer arithmetic.

The operational semantics for the source language are standard, but those for the target are not. The two main differences between these semantics are discussed in the following paragraphs, where the target-level semantics are illustrated by means of some representative rules in Figure 4. The rules in this figure make the simplifying assumption that $\overline{s} = s$, that is, only a single stack frame is considered, and that $\overline{c}$ consists of a single source statement. In the following discussion, $h[(l, i) \to v]$ denotes an update of the existing value of the heap h at location l and index i with value v. For the stack, $s[id_p \to v]$ similarly denotes an assignment of the value v to the previously declared variable id_p. The evaluation of *exp* in stack-frame s is denoted by $[\![exp]\!]_s$.

The first difference between the source and target semantics is caused by the linearity of capabilities, as they cannot be duplicated. When a linear capability $l^{[a,b]}$ is copied, the original value is set to null. We call this process linear capability *erasure*. The target-level judgment $v \rightsquigarrow_{\text{ValErase}} v'$ describes how a target value v is erased into a result value v' by replacing linear capabilities with null pointers. For example:

$$l^{[a,b]} \rightsquigarrow_{\text{ValErase}} \text{null} \qquad (l_1^{[a_1,b_1]}, 5, l_2^{[a_2,b_2]}) \rightsquigarrow_{\text{ValErase}} (\text{null}, 5, \text{null})$$

The rule ARRAYLKUP in Figure 4 describes the operational semantics of target-level array lookup. It is equivalent to its source-level counterpart, except for the addition of a $\rightsquigarrow_{\text{ValErase}}$ judgment. It illustrates how the read value v should be erased inside the array, by setting $h'(l, a + n) = v'$.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

$$\frac{\begin{array}{c}[\![texp_1]\!]_s = l^{[a,b]} \quad [\![texp_2]\!]_s = n \quad h(l, a+n) = v \\ 0 \le n \le b-a \qquad v \rightsquigarrow_{\text{ValErase}} v' \\ s' = s[id_{\text{p}} \to v] \qquad h' = h[(l, a+n) \to v']\end{array}}{\langle s, h\rangle \mid id_{\text{p}} = texp_1[texp_2] \hookrightarrow \langle s', h'\rangle \mid \text{skip}} \text{ (ARRAYLKUP)}$$

$$\frac{\begin{array}{c}s(id_{\text{p}}) = l^{[a,b]} \quad [\![texp_1]\!]_s = n \quad [\![texp_2]\!]_s = v \\ 0 \le n \le b-a \qquad texp_2 \rightsquigarrow^{s}_{\text{StoreLinCap}} [env] \\ s' = s[env] \qquad h' = h[(l, a+n) \to v]\end{array}}{\langle s, h\rangle \mid id_{\text{p}}[texp_1] = texp_2 \hookrightarrow \langle s', h'\rangle \mid \text{skip}} \text{ (ARRAYMUT)}$$

$$\frac{\begin{array}{c}s(id_{\text{p3}}) = l^{[a,b]} \quad [\![texp]\!]_s = n \quad 1 \le n \le b-a \\ s' = s[id_{\text{p1}} \to l^{[a,a+n]}][id_{\text{p2}} \to l^{[a+n,b]}][id_{\text{p3}} \to \text{null}]\end{array}}{\langle s, h\rangle \mid (id_{\text{p1}},\ id_{\text{p2}}) = \text{split}(id_{\text{p3}}, texp) \hookrightarrow \langle s', h\rangle \mid \text{skip}} \text{ (SPLIT)}$$

$$\frac{\begin{array}{c}s(id_{\text{p2}}) = l^{[a,n]} \quad s(id_{\text{p3}}) = l^{[n,b]} \\ s' = s[id_{\text{p1}} \to l^{[a,b]}][id_{\text{p2}} \to \text{null}][id_{\text{p3}} \to \text{null}]\end{array}}{\langle s, h\rangle \mid id_{\text{p1}} = \text{join}(id_{\text{p2}}, id_{\text{p3}}) \hookrightarrow \langle s', h\rangle \mid \text{skip}} \text{ (JOIN)}$$

Fig. 4. Rules illustrating the target language operational semantics and its linear aspects.

The rule ARRAYMUT in Figure 4 describes the operational semantics of target-level array mutation. The sole difference with the source-level version is caused by the judgment

$$texp_2 \rightsquigarrow^{s}_{\text{StoreLinCap}} [env]$$

which we illustrate below. This judgment is used to erase any linear capabilities present in $texp_2$ that were written into the array $l^{[a,b]}$ by the ARRAYMUT operation, to avoid them being duplicated. The resulting environment $[env]$ nulls these capabilities in the current stack frame, as shown by the assignment $s' = s[env]$ in the ARRAYMUT rule. Additionally, this judgment makes the semantics get stuck if the same linear capability is used twice in $texp_2$.

Generally, $texp \rightsquigarrow^{s}_{\text{StoreLinCap}} [env]$ can be seen as the lifting of $v \rightsquigarrow_{\text{ValErase}} v'$ from values v to expressions $texp$. Rather than erasing capabilities inside values v, now stack variables id_{p} that appear inside $texp$ have to be reassigned in order to erase their linear capabilities. This is the reason the judgment's output is not an expression $texp'$, but rather, a reassignment of these local variables, that is, an environment $[env]$. The judgment $texp \rightsquigarrow^{s}_{\text{StoreLinCap}} [env]$ is used whenever a target operational semantics rule evaluates and uses $texp$, and the linear capabilities that are moved in the process have to be erased. The judgment is therefore also used when, for example, calling functions using linear arguments or when assigning variables.

More concretely, $texp \rightsquigarrow^{s}_{\text{StoreLinCap}} [env]$ erases the linear capabilities that appear inside $texp$ by reassigning (in the current stack frame s) local variables id_{p} appearing inside $texp$. Notice that solely the linear capabilities that are actually used linearly should be erased; for example, id_{p} appearing under an equality or as an argument to the addr function does not require erasure. The environment $[env]$ computes values for the erased id_{p} by using $\rightsquigarrow_{\text{ValErase}}$. For example, assuming $[\![id_{\text{p1}}]\!]_s = l_1^{[a_1,b_1]}$ and $[\![id_{\text{p2}}]\!]_s = (l_2^{[a_2,b_2]}, l_3^{[a_3,b_3]})$, we have:

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

$$\frac{\{P\}_\gamma\ c\ \{Q\}_{\gamma'} \rightsquigarrow p \qquad \mathrm{dom}(\gamma_{\mathrm{post}}) \subseteq \mathrm{dom}(\gamma') \qquad \vdash Q \Rightarrow Q^{\mathrm{post}} \qquad \forall x \in \mathrm{dom}(\gamma_{\mathrm{post}}).\, Q \vdash \gamma'(x) == \gamma_{\mathrm{post}}(x)}{\{P\}_\gamma\ c\ \{Q_{\mathrm{post}}\}_{\gamma_{\mathrm{post}}} \rightsquigarrow p}\ (\textsc{ConseqPost}) \qquad \frac{\{P\}_\gamma\ c\ \{Q\}_{\gamma'} \rightsquigarrow p \qquad \mathrm{CN}(R) = \overline{n} \qquad \overline{n}\ \text{fresh} \qquad \gamma_{\mathrm{s}} = \gamma \uplus \gamma_{\mathrm{frame}} \qquad \gamma'_{\mathrm{s}} = \gamma' \uplus \gamma_{\mathrm{frame}}}{\{P * R\}_{\gamma_{\mathrm{s}}}\ c\ \{Q * R\}_{\gamma'_{\mathrm{s}}} \rightsquigarrow p}\ (\textsc{Frame})$$

$$\frac{\{P\}_\gamma\ sstm_1\ \{Q\}_{\gamma'} \rightsquigarrow p_1 \qquad \{Q\}_{\gamma'}\ sstm_2\ \{R\}_{\gamma''} \rightsquigarrow p_2}{\{P\}_\gamma\ sstm_1; sstm_2\ \{R\}_{\gamma''} \rightsquigarrow p_1; p_2}\ (\textsc{Seq}) \qquad \frac{\{P * sexp_\gamma\}_\gamma\ sstm_1\ \{Q\}_{\gamma'} \rightsquigarrow p_1 \qquad \{P * !sexp_\gamma\}_\gamma\ sstm_2\ \{Q\}_{\gamma'} \rightsquigarrow p_2}{\{P\}_\gamma\ \mathrm{if}\ sexp\ \mathrm{then}\ sstm_1\ \mathrm{else}\ sstm_2\ \{Q\}_{\gamma'} \rightsquigarrow \mathrm{if}\ sexp\ \mathrm{then}\ p_1\ \mathrm{else}\ p_2}\ (\textsc{If})$$

Fig. 5. Structural separation logic rules that can be extended to compilation rules.

$$\begin{aligned}
(id_{\mathrm{p1}}, id_{\mathrm{p2}}) &\rightsquigarrow^{s}_{\mathrm{StoreLinCap}} [id_{\mathrm{p1}} \to \mathrm{null}][id_{\mathrm{p2}} \to (\mathrm{null}, \mathrm{null})] \\
\mathrm{addr}(id_{\mathrm{p2}}) &\rightsquigarrow^{s}_{\mathrm{StoreLinCap}} [\,] \\
id_{\mathrm{p2}}.2 &\rightsquigarrow^{s}_{\mathrm{StoreLinCap}} [id_{\mathrm{p2}} \to (l_2^{[a_2,b_2]}, \mathrm{null})] \\
(id_{\mathrm{p1}}, id_{\mathrm{p2}}).1 &\rightsquigarrow^{s}_{\mathrm{StoreLinCap}} [id_{\mathrm{p1}} \to \mathrm{null}]
\end{aligned}$$

Additionally, $\rightsquigarrow^{s}_{\mathrm{StoreLinCap}}$ should ensure that a single linear capability is not used multiple times in the same *texp*, since this would cause duplication. For example, assuming the same stack frame as before, we have $\neg\exists v'.\, (id_{\mathrm{p1}}, id_{\mathrm{p1}}) \rightsquigarrow^{s}_{\mathrm{StoreLinCap}} v'$, causing the semantics to avoid duplication by *getting stuck*.

The second big difference between source and target occurs where the built-in target-level functions join and split are concerned. These ghost statements and their reification were already discussed in Section 3.1. As mentioned, source-level ghost statements solely have an effect on the separation logic proof and are hence equivalent to skip in the source semantics. In the target, on the other hand, the reified ghost statements manipulate physical linear capabilities instead. The rules for these reified ghost statements are given by SPLIT and JOIN in Figure 4. As expected, they, respectively, split and join linear capabilities. Notice that both rules erase their source operands to ensure linearity, by explicitly setting them to null. A use of $\rightsquigarrow^{s}_{\mathrm{StoreLinCap}}$ is not required, given that the source operands are constrained to be simple program variables.

4 Inference rules and compilation by example

This section introduces the separation logic inference rules that constitute separation logic proof trees. Because our compiler is separation-logic-proof-directed, the compilation rules directly derive from these rules, so we present both simultaneously. The rules we present in this section are syntactic, that is, presented axiomatically rather than derived from the operational semantics. While it allowed us to focus more on the essential points of this paper, this approach has disadvantages. For a discussion of syntactically versus semantically derived rules, see the future work section. A relevant selection of rules is spread over Figures 5, 6, 7 and 8. The separation logic rules and the compilation rules are obtained by, respectively, ignoring and not ignoring all green text. Compilation of a separation logic proof $\vdash \mathfrak{s}$ to target code $\mathfrak{t}$ is denoted $\vdash \mathfrak{s} \rightsquigarrow \mathfrak{t}$. Other judgments appearing in these figures are explained as needed below.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

$$\tau_s \rightsquigarrow_{\text{CompileType}} \tau_t \qquad \tau_s \rightsquigarrow_{\text{def}} v$$

$$\frac{\tau \rightsquigarrow_{\text{def}} v \quad \tau \rightsquigarrow_{\text{CompileType}} \tau' \quad n, id_\ell \text{ fresh} \quad id_p \in \text{dom}(\gamma) \quad \gamma' = \gamma[id_p : id_\ell]}{(\textsc{Malloc})\ \{sexp_\gamma > 0\}_\gamma\ id_p = \text{malloc}(sexp * \text{sizeof}(\tau))\ \{n : id_\ell \mapsto_\tau \text{repeat}(sexp_\gamma, v)\}_{\gamma'} \rightsquigarrow \tau' * n;\ n = \text{malloc}(sexp * \text{sizeof}(\tau'));\ id_p = \text{addr}(n)}$$

$$\frac{n', n'' \text{ fresh} \quad \tau \rightsquigarrow_{\text{CompileType}} \tau'}{(\textsc{Split})\ \{n : exp_a \mapsto_\tau l * \text{length}(l) == exp_1 * 0 < sexp_\gamma < exp_1\}_\gamma\ //@\text{split } n[sexp]\ \{n' : exp_a \mapsto_\tau \text{take}(l, 0, sexp_\gamma) * n'' : (exp_a + sexp_\gamma) \mapsto_\tau \text{take}(l, sexp_\gamma, exp_1)\}_\gamma \rightsquigarrow \tau' * n'; \tau' * n''; \{n', n''\} = \text{split}(n, sexp)}$$

$$\frac{id_p \notin \text{dom}(\gamma) \quad \tau \rightsquigarrow_{\text{def}} v \quad \tau \rightsquigarrow_{\text{CompileType}} \tau' \quad \gamma' = \gamma[id_p : v]}{(\textsc{VarDecl})\ \{\}_\gamma\ \tau\ id_p\ \{\}_{\gamma'} \rightsquigarrow \tau'\ id_p} \qquad \frac{id_p \in \text{dom}(\gamma) \quad \gamma' = \gamma[id_p : sexp_\gamma]}{(\textsc{VarAsgn})\ \{\}_\gamma\ id_p = sexp\ \{\}_{\gamma'} \rightsquigarrow id_p = sexp} \qquad \frac{}{(\textsc{Guard})\ \{P\}\ \text{guard}(sexp)\ \{P * sexp_\gamma\} \rightsquigarrow \text{guard}(sexp)}$$

$$\frac{}{(\textsc{ArrayMut})\ \{n : id_{p,\gamma} \mapsto exp * \text{length}(exp) == exp_1 * 0 \le sexp_{1,\gamma} < exp_1\}_\gamma\ id_p[sexp_1] = sexp_2\ \{n : id_{p,\gamma} \mapsto \text{update}(exp, sexp_{1,\gamma}, sexp_{2,\gamma})\}_\gamma \rightsquigarrow n[sexp_1] = sexp_2} \qquad \frac{\{P\}_\gamma\ sstm\ \{Q\}_{\gamma'} \rightsquigarrow p \quad \text{CN}(Q) = \overline{n}}{(\textsc{Return})\ \{P\}_\gamma\ sstm;\ \text{return } \{\overline{sexp}\}\ \{Q * \overline{result} == \overline{sexp_{\gamma'}}\}_{\gamma'} \rightsquigarrow p; \text{return } (\overline{sexp}, \overline{n})}$$

$$\frac{\Sigma(f) = \{PRE_f, POST_f, \overline{id_{\text{arg}}}\} \quad PRE_f \approx_{\text{Names}} PRE \quad POST_f \approx_{\text{Names}} POST \quad \overline{id} \in \text{dom}(\gamma) \quad \gamma' = \gamma[\overline{id} : \overline{id_{\text{res}}}] \quad \overline{id_{\text{res}}}, \overline{n} \text{ fresh} \quad [subst_{\text{pre}}] = [\overline{id_{\text{arg}}} \mapsto \overline{sexp}_\gamma] \quad [subst_{\text{post}}] = [subst_{\text{pre}}][\overline{result} \mapsto \overline{id_{\text{res}}}] \quad \text{CN}(PRE) = \overline{m} \quad POST \rightsquigarrow_{\text{resDecl}} \overline{\tau_n}\ \overline{n}}{(\textsc{FApp})\ \{PRE[subst_{\text{pre}}]\}_\gamma\ \overline{id} = f(\overline{sexp})\ \{POST[subst_{\text{post}}]\}_{\gamma'} \rightsquigarrow \overline{\tau_n}\ \overline{n};\ \{\overline{id}, \overline{n}\} = f_{\text{comp}}(\overline{sexp}, \overline{m})}$$

Fig. 6. Basic separation logic rules that can be extended to compilation rules.

$$\frac{isfunc = \overline{\tau_{\text{ret}}}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}})\ //@\text{pre } PRE\ //@\text{post } POST\ \{sstm; \text{return } \overline{sexp}\}\ isfunc \rightsquigarrow_{\text{Decl}} \{\overline{\tau'_{\text{ret}}}, \overline{\tau_{\text{post}}}\}\ f(\overline{\tau'_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_{\text{pre}}\ m})\ \{PRE\}_{[\overline{id_{\text{arg}}}:id_{\text{arg}}]}\ sstm; \text{return } \overline{sexp}\ \{POST\}_\gamma \rightsquigarrow p_1; \text{return } \{\overline{texp}, \overline{n}\}}{(\textsc{ImplFVerif})\ \vdash isfunc \rightsquigarrow \{\overline{\tau'_{\text{ret}}}, \overline{\tau_{\text{post}}}\}\ f_{\text{comp}}(\overline{\tau'_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_{\text{pre}}\ m})\ \{p_1; \text{return } \{\overline{texp}, \overline{n}\}\}} \qquad \frac{csfunc = \overline{\tau_{\text{ret}}}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}})//@\text{pre } PRE\ //@\text{post } POST\ csfunc \rightsquigarrow_{\text{Decl}} \{\overline{\tau'_{\text{ret}}}, \overline{\tau_{\text{post}}}\}\ f(\overline{\tau'_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_{\text{pre}}\ m})}{(\textsc{ContFVerif})\ \vdash csfunc \rightsquigarrow \{\overline{\tau'_{\text{ret}}}, \overline{\tau_{\text{post}}}\}\ f(\overline{\tau'_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_{\text{pre}}\ m})}$$

$$\frac{scomp = \overline{isfunc}//@\text{import } \overline{csfunc_i}\ //@\text{export } \overline{csfunc_e}\quad \vdash_{\text{WF}} scomp \quad \vdash \overline{csfunc_i} \rightsquigarrow_{\text{Outcall}} \overline{ctfunc_i}, \overline{stub_{\text{out}}} \quad \vdash \overline{isfunc} \rightsquigarrow \overline{itfunc} \quad \vdash \overline{csfunc_e} \rightsquigarrow_{\text{Incall}} \overline{ctfunc_e}, \overline{stub_{\text{in}}}}{(\textsc{CompVerif})\ \vdash scomp \rightsquigarrow (\overline{itfunc}\ \overline{stub_{\text{out}}}\ \overline{stub_{\text{in}}})\ //@\text{import } \overline{ctfunc_i}\ //@\text{export } \overline{ctfunc_e}} \qquad \frac{sprog = \overline{scomp}\ //@\text{main} = id \quad \vdash_{\text{WF}} sprog \quad \vdash \overline{scomp} \rightsquigarrow \overline{tcomp}}{(\textsc{ProgVerif})\ \vdash sprog \rightsquigarrow \overline{tcomp}\ //@\text{main} = id}$$

Fig. 7. Higher-level separation logic rules that can be extended to compilation rules.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

$$\frac{}{exp \rightsquigarrow_{\mathrm{p}} \mathrm{guard}(exp)}\ (\textsc{ConditionReify}) \qquad \frac{C \rightsquigarrow_{\mathrm{p}} c_1 \quad C' \rightsquigarrow_{\mathrm{p}} c_2}{C * C' \rightsquigarrow_{\mathrm{p}} c_1; c_2}\ (\textsc{SepConjPReify})$$

$$\frac{\begin{array}{c}\tau \rightsquigarrow_{\mathrm{CompileType}} \tau' \\ check = (\mathrm{guard}(n\ !{=}\ \mathrm{null}); \mathrm{guard}(\mathrm{length}(n) == k)) \\ decl = (\tau *_0 x;\ \tau'\ x_1;\ \ldots;\ \tau'\ x_k) \\ assign = \\ (x = \mathrm{addr}(n); x_1 = n[0];\ \ldots;\ x_k = n[k-1])\end{array}}{n : x \mapsto_{\tau} [x_1, \ldots, x_k] \rightsquigarrow_{\mathrm{s}} (check, decl, assign)}\ (\textsc{ResourceReify}) \qquad \frac{C \rightsquigarrow_{\mathrm{s}} (c_1, d_1, a_1) \quad C' \rightsquigarrow_{\mathrm{s}} (c_2, d_2, a_2)}{C * C' \rightsquigarrow_{\mathrm{s}} (c_1; c_2, d_1; d_2, a_1; a_2)}\ (\textsc{SepConjCReify})$$

$$\frac{\begin{array}{c}f_i = \overline{\tau'_{\mathrm{ret}}}\ f\ (\overline{\tau_{\mathrm{arg}}\ id_{\mathrm{arg}}}) \\ //@\mathrm{pre}\ \overline{m} : PRE_{\mathrm{s}} * PRE_{\mathrm{p}}\ //@\mathrm{post}\ \overline{n} : POST_{\mathrm{s}} * POST_{\mathrm{p}} \\ p_i = \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f\ (\overline{\tau'_{\mathrm{arg}}\ id_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m}) \qquad \vdash f_i \rightsquigarrow p_i \\ \overline{m} : PRE_{\mathrm{s}} \rightsquigarrow_{\mathrm{s}} (\_, d_{\mathrm{pre}}, a_{\mathrm{pre}}) \qquad \overline{n} : POST_{\mathrm{s}} \rightsquigarrow_{\mathrm{s}} (cs_{\mathrm{post}}, d_{\mathrm{post}}, a_{\mathrm{post}}) \\ POST_{\mathrm{p}} \rightsquigarrow_{\mathrm{p}} cp_{\mathrm{post}} \\ stub_{\mathrm{outcall}} = \left\{\begin{array}{l}\{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f_{\mathrm{comp}}\ (\overline{\tau'_{\mathrm{arg}}\ id_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m})\{ \\ d_{\mathrm{pre}}; a_{\mathrm{pre}}; \overline{\tau'_{\mathrm{ret}}\ result}; \overline{\tau_{\mathrm{post}}\ n}; \\ (\overline{result}, \overline{n}) = f(\overline{id_{\mathrm{arg}}}, \overline{m}); \\ cs_{\mathrm{post}}; d_{\mathrm{post}}; a_{\mathrm{post}}; cp_{\mathrm{post}}; \\ \mathrm{return}\ (\overline{result}, \overline{n})\}\end{array}\right\}\end{array}}{\vdash f_i \rightsquigarrow_{\mathrm{Outcall}} p_i, stub_{\mathrm{outcall}}}\ (\textsc{Outcall})$$

Fig. 8. Compilation rules for generating outcall stubs.

We illustrate the rules using the running example in Figure 9. Since our compiler is separation-logic-proof directed, it receives a proof of the verified component in Figure 9 as input. To keep the example simple and concise, we avoid foreach loops and range resources, stick to arrays of statically known size, and we use the fixed-size array resource syntax discussed in Section 3.1. The inference and compilation rules will still be presented in their general form.

By f's contract in Figure 9, it receives a pointer a and a resource m to access a two-element integer array corresponding to this pointer. For simplicity, f leaks the memory resource m by not handing it back in its postcondition. It adds 1 to either the second element of a or its negation, depending on the first element c. It will use an untrusted library function *add1* to add the 1. The contract of *add1* specifies that it takes a pointer a and a resource m to access a one-element array corresponding to a. It returns the value *result*, equaling the contents of a increased by 1, and returns the same resource from the precondition, now named n, in its postcondition. The symbolic variable identifier *result* is a privileged name, used to denote a function's return value(s) in its postcondition ($\overline{result_i}$ is used in case of multiple return values). Based on the value of a's first element c, f either calls *add1* directly or emulates adding 1 to the negation by inverting the last element of a, storing it in a new array b and only then calling *add1*.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Source

Verified Component

```
int f(int* a)
//@pre m: a ↦int [c,a1]
//@post result == (c == 0 ? a1 + 1 : −a1 + 1) {
  int res; int c; c = a[0];
  //@split m[1];
  if c == 0
  then  {res = add1(a + 1)}
  else  {int* b; int a1; a1 = (a + 1)[0];
         b = malloc(1 * sizeof(int));
         b[0] = −a1;
         res = add1(b)};
  return res }
```

Context Declaration

```
int add1(int* a)
//@pre m: a ↦int [a1]
//@post n: a ↦int [a1] * result == a1 + 1
```

Target

```
int fcomp(int*0 a,int* m){
  int res; int c; c = m[0];
  int* m1; int* m2; m1,m2 = split(m,1);
  if c == 0
  then {int* r1; res,r1 = add1comp(a + 1,m2)}
  else  {int*0 b; int a1; a1 = m2[0];
         int* l; l = malloc(1 * sizeof(int));
         b = addr(l);
         l[0] = −a1;
         int* r2; res,r2 = add1comp(b,l)};
  return res }
```

```
(int,int*) add1(int*0 a,int* m)
```

Outcall Stub

```
(int,int*) add1comp(int*0 a,int* m){
  int*0 apre; int a1pre;
  apre = addr(m); a1pre = m[0];
  int result; int* n;
  (result,n) = add1(a,m);
  guard(n!=null); guard(length(n) == 1);
  int*0 apost; int a1post;
  apost = addr(n); a1post = n[0];
  guard(result == a1post + 1);
  guard(apost == a); guard(a1post == a1pre);
  return (result,n) }
```

Fig. 9. Illustrative example: conditionally add 1 to the second element of a length-2 array or its negation.

There are four types of compilation rules: structural, basic, higher-level and stub compilation rules. We discuss these classes in the next subsections and illustrate them using the running example.

4.1 Structural rules

Structural rules are rules that build more complex proofs from simpler proofs. They are more involved in the construction of the separation logic proof itself than in the proof-directed aspects of the compilation and therefore have very straightforward compilation rules. The structural rules are ConseqPost, Frame, Seq and If, presented in Figure 5.

The consequence and frame rules ConseqPost and Frame are pure proof glue rules that allow altering proofs and do not influence the compiled program. The consequence rule ConseqPost allows weakening of both the symbolic heap and the environment in the postcondition of a separation logic triple, in order to link it to a subsequent triple. The judgment $assump \vdash cond$ denotes that the boolean condition *cond* holds under the assumptions in *assump*. In fact, we also need a dual rule ConseqPre that allows strengthening the precondition of a separation logic triple. The rules ConseqPre and ConseqPost combine to form the full consequence rule Conseq. Our full Conseq rule also allows renaming outer separation logic resources n and manipulating conditional assertions. Both of these operations will influence the compiled code p, but are omitted for brevity.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

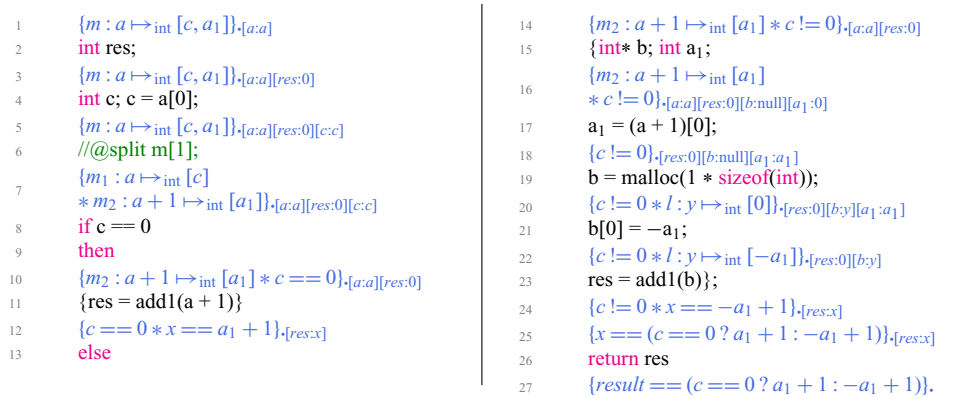


Fig. 10. Separation logic proof of the function given in the illustrative example.

The frame rule is a classical separation logic rule. It allows neglecting a redundant part of the symbolic heap and the environment in order to simplify the separation logic state. The function CN(R) returns all resource names that appear in the separation logic assertion R. We require these names to be fresh, to avoid name clashes. The sequence and conditional rules SEQ and IF describe proofs for the sequencing and conditional source statements, respectively. If all applications of these four structural proof rules are left implicit in a function body's separation logic proof, the proof tree can be represented as a *symbolic execution* (Vogels *et al.*, 2015). Such a symbolic execution of the body of f is used in Figure 10 to illustrate the rules in this subsection and the next.

The CONSEQ rule is used to omit information that is no longer useful as quickly as possible in order to keep the proof concise. Examples are the resource m_1 that is dropped after line 7 and the environment entry $[a : a]$ that is omitted after line 16. Consequence is also used to reshape postconditions to match the conditions required by a different rule: CONSEQ unifies, for example, the symbolic heaps on lines 12 and 24 by weakening them to the symbolic heap on line 25. Because of this unification, the IF rule becomes applicable.

The IF rule obviously creates the separation logic triple for the if-statement on lines 8–24. Notice how the IF rule introduces $c == 0$ and $c != 0$ in the symbolic heaps on lines 10 and 14, respectively.

4.2 Basic rules

Basic rules construct a proof triple from the ground up for a single non-sequenced source statement. They are the elementary building blocks of the symbolic execution in Figure 10 and the workhorses of the separation-logic-driven compiler. The rules are named after the source statements they create a proof for, that is, SKIP, MALLOC, FOR, FLATTEN, COLLECT, SPLIT (has 2 versions: one for range resources and one for array resources), JOIN (again has 2 versions), VARDECL, VARASGN, ARRAYMUT, ARRAYLKUP, GUARD, FAPP and RETURN. A representative selection is presented in Figure 6.

In the following descriptions of the inference rules, $sexp_\gamma$ denotes $sexp$ where each program variable id_p is substituted by $\gamma(id_\mathrm{p})$ (implicitly requiring that $id_\mathrm{p} \in dom(\gamma)$). Also note that compilation is the identity for expressions $sexp$, since the variables that

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

$$\frac{(\textsc{CompileInt})}{\text{int} \rightsquigarrow_{\text{CompileType}} \text{int}} \quad \frac{(\textsc{CompileSrcPtr})}{\tau* \rightsquigarrow_{\text{CompileType}} \tau*_0}$$

$$\frac{\tau_1 \rightsquigarrow_{\text{CompileType}} \tau'_1 \quad \dots \quad \tau_k \rightsquigarrow_{\text{CompileType}} \tau'_k \quad (\textsc{CompileTuple})}{(\tau_1, \dots, \tau_k) \rightsquigarrow_{\text{CompileType}} (\tau'_1, \dots, \tau'_k)}$$

Fig. 11. Inference rules defining. $\tau_s \rightsquigarrow_{\text{CompileType}} \tau_t$

$$\frac{(\textsc{DefInt})}{\text{int} \rightsquigarrow_{\text{def}} 0} \quad \frac{(\textsc{DefPtr})}{\tau* \rightsquigarrow_{\text{def}} \text{null}}$$

$$\frac{\tau_1 \rightsquigarrow_{\text{def}} def_1 \quad \dots \quad \tau_k \rightsquigarrow_{\text{def}} def_k \quad (\textsc{DefTuple})}{(\tau_1, \dots, \tau_k) \rightsquigarrow_{\text{def}} (def_1, \dots, def_k)}$$

Fig. 12. Inference rules defining. $\tau_s \rightsquigarrow_{\text{def}} v$

represent pointers contained in *sexp* are automatically converted to address capabilities by the compilation.

The auxiliary judgments $\tau_s \rightsquigarrow_{\text{CompileType}} \tau_t$, which compiles a source type τ_s to the corresponding target type τ_t, and $\tau_s \rightsquigarrow_{\text{def}} v$, which returns the default value v for the type τ_s, are first defined in Figures 11 and 12, respectively.

The remainder of this section consists of a discussion of the depicted basic separation logic rules and compilation rules. Separation logic rules are illustrated by referencing lines from Figure 10. Compilation rules are illustrated using a combination of source code lines from Figure 10 and lines from the compiled verified function f_{comp} in Figure 9.

The MALLOC rule assigns a fresh logical variable id_ℓ to id_p in γ and creates a new array resource n, consisting of the default value v repeated $sexp_\gamma$ times. In the corresponding MALLOC compilation rule, the variable $\tau' * n$ is declared and assigned the malloced linear capability, that is, the resource corresponding to id_p in the source language. This target-level assignment to the variable n clearly makes it the reified version of the source resource n (also freshly introduced by the rule). As id_p is itself merely a permissionless address on the target level, we assign it using the addr function, maintaining the correspondence between a and n from the separation logic proof. The MALLOC rule is demonstrated on lines 18–20, where a new resource l is created. Lines 18–20 are compiled to lines 7–8 in f_{comp}.

The SPLIT and JOIN rules for both array resources and range resources are dual, with the difference that SPLIT has to check whether the given splitting index to split on is in bounds, whereas JOIN has to check memory adjacency of the two resources it is given. Given the similarities, only the array version of the split rule is detailed in Figure 6. Notice how this rule indeed performs the same operation on separation logic resources that the target operational semantics rule SPLIT in Figure 4 performed on linear capabilities. Consequently, the SPLIT compilation rule simply mirrors the source-level split statement in the target language, using the built-in split operation on the corresponding reified linear capabilities. The SPLIT rule is demonstrated on lines 5–7, where resource m is split. Lines 5–7 are compiled to line 3 in f_{comp}.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

The rule VARDECL and VARASGN prove variable declaration and assignment, respectively. Their compilation rules are straightforward, apart from the change in type in VARDECL. The VARDECL rule is demonstrated on, for example, lines 1–3, which compile to the first declaration on line 2 in f_{comp}.

The rules ARRAYMUT and ARRAYLKUP are very similar, so Figure 6 only shows the former. The mutation of source arrays is again compiled to the same action on the corresponding reified resource. Both rules are demonstrated on lines 20–22 and lines 16–18, respectively. The ARRAYMUT rule enforces the logical address of the resource n to be exactly equal to $id_{\text{prog},\gamma}$, rather than allowing for some additional offset, out of simplicity considerations (ARRAYLKUP enforces something similar). For this reason, line 17 contains $(a+1)[0]$ and not $a[1]$; $(a+1)_\gamma$ equals the logical address of the resource m_2, whereas a_γ does not. These lines, respectively, compile to lines 9 and the end of line 6 in f_{comp}.

The GUARD rule adds the asserted conditions to the symbolic heap, and compilation is the identity.

Function application FAPP is the most intricate basic rule. The variable Σ denotes a component-wide function environment that contains the contract and argument names for each function f (including imported functions). The caller does not need to match the called function's contract exactly: we can allow outer resource names to differ. That is why the relation $\approx_{\text{Names}}$ is used, to enforce equality up to resource names of pre- and postcondition. In the caller's pre- and postcondition *PRE* and *POST*, the concrete logical expressions $\overline{sexp_\gamma}$ used in the function call are substituted for the arguments $\overline{id_{\text{arg}}}$, instantiating the callee's contract with the caller-provided arguments. Additionally, in the caller's postcondition, the privileged $\overline{result}$ variables are substituted with fresh logical variables $\overline{id_{\text{res}}}$, linked to $\overline{id}$ in γ'. Fresh resource names $\overline{n}$ are required to avoid name clashes.

Given this rule, the FAPP compilation can now be discussed. The resource names $\overline{m}$ in *PRE* will be reified and are extracted using the function CN. The resource names $\overline{n}$ in *POST* have to be fresh and will hence need to be declared in the compiled code first, before reification. We use an auxiliary judgment $assert \rightsquigarrow_{\text{ResDecl}} \overline{\tau_n\ n}$ that extracts all resource names $\overline{n}$ in *assert* and pairs them with their reified types $\overline{\tau_n}$ in target-level declarations $\overline{\tau_n\ n}$. This judgment extracts the correct names $\overline{n}$ from *POST* and immediately tells us what declarations $\overline{\tau_n\ n}$ to create. The compiled function call contains the reified versions of the precondition resources $\overline{m}$ as extra arguments and receives the reified postcondition resources $\overline{n}$ as extra return values. The reason each function f is renamed to f_{comp} during compilation is related to incall and outcall stubs and will become more clear in Section 4.4.

The FAPP rule is illustrated on lines 10–12 and lines 22–24. For lines 10–12, for example, $[subst_{\text{pre}}] == [a \mapsto a+1]$ and $[subst_{\text{post}}] == [a \mapsto a+1][result \mapsto x]$, where $[res : x]$ is substituted in the environment after the call. Given these substitutions, we can see that the preconditions indeed only differ in the chunk names m_2 versus m, hereby satisfying $\approx_{\text{Names}}$. The same holds for the postconditions (where the returned resource has already been omitted by CONSEQ on line 12). Lines 10–12 are hence compiled to line 5 in f_{comp}, where $add1_{\text{comp}}$ denotes the outcall stub for *add1*.

The RETURN rule forms a special case because return is not a source statement; it appears exactly once at the end of each function body. Since SEQ can only sequence source statements, the RETURN rule has to manually construct a new proof from a previous

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

one. Conceptually, however, RETURN is a basic rule. Given a proof of *sstm*, the RETURN rule introduces the privileged $\overline{result}$ logical variables to the symbolic heap, equaling the returned expressions. The return compilation rule produces a target return statement, which additionally returns all reified resources $\overline{n}$. The CONSEQ rule reshapes the contract Q after the return into the function body's postcondition (in this phase, leaking resources is disallowed because the set of reified resources $\overline{n}$ is already fixed). Lines 25–27 demonstrate the return rule and are compiled to line 11 in f_{comp}. No returned variables are added because f leaks its resources. Notice that line 27 follows from RETURN's postcondition $\{x == (c == 0\,?\,a_1 + 1 : -a_1 + 1) * result == x\}$ by the CONSEQ rule.

4.3 Higher-level rules

Given a separation-logic proof of a function's body, constructed as in the previous subsections, we now introduce rules that define the notion of separation logic proof $\vdash$ for entire functions, components and source programs, as these higher-level structures are what we are most interested in compiling. The higher-level rules are IMPLFVERIF, CONTFVERIF, COMPVERIF and PROGVERIF, presented in Figure 7 and discussed below. For all compilation-related judgments $\rightsquigarrow_{\text{X}}$, we define the following tuple-based shorthand: $\vdash \overline{s_i} \rightsquigarrow_{\text{X}} \overline{t_i} \triangleq \forall i.\, \vdash s_i \rightsquigarrow_{\text{X}} t_i$.

First, we have the rules for implemented functions IMPLFVERIF and context functions CONTFVERIF. The main difference between these rules is that CONTFVERIF does not reference any function body, as expected, whereas IMPLFVERIF requires a proof of the function body to construct a function proof $\vdash$. The precondition environment of this proof is $[\overline{id_{\text{arg}} : id_{\text{arg}}}]$, since our separation logic contract preconditions always implicitly map the function arguments $\overline{id_{\text{arg}}}$ to logical variables of the same names $\overline{id_{\text{arg}}}$. Non-coincidentally, this environment is the starting environment in Figure 10. This initial environment explains how we can allow function contracts to be entirely logical assertions, but still reference function arguments $\overline{id_{\text{arg}}}$.

The corresponding compilation rules both use an auxiliary judgment $sfunc \rightsquigarrow_{\text{Decl}} tfunc$ that compiles a function declaration *sfunc* to a declaration *tfunc* by reifying all resources in the given pre- and postcondition into arguments and return values, respectively, and compiling existing argument and return types using $\rightsquigarrow_{\text{CompileType}}$. The rule IMPLFVERIF also changes any implemented function f's name to f_{comp} during compilation, again for stub-related reasons explained in Section 4.4. The proof of Figure 10 suffices to construct a proof $\vdash$ of f using IMPLFVERIF, which can then be compiled to the target-level function f_{comp} in Figure 9. The declaration of *add1*, on the other hand, is compiled to the declaration of *add1* using CONTFVERIF.

The component verification rule COMPVERIF allows verification and compilation of entire components. A component *scomp* has a proof $\vdash scomp$ if it is well-formed (denoted by $\vdash_{\text{WF}} scomp$, which includes some restrictions mentioned in Section 4.4) and if all its implemented and (exported and imported) context functions have proofs. A compiled component is constructed from the compilation of its functions. The compilation rules $\rightsquigarrow_{\text{Incall}}$ and $\rightsquigarrow_{\text{Outcall}}$ both subsume the CONTFVERIF rule and additionally generate the incall and outcall stubs for exported and imported functions, respectively. Both of these rules are discussed in the next subsection.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

As an example, given the proofs of f and *add1* constructed in the previous paragraph, the CompVerif rule proves the source component in Figure 9, which has f as an internal function, an empty export list and the declaration of *add1* as the import list. The source component is compiled to the target component in the bottom half of Figure 9, where $add1_{\text{comp}}$ is the outcall stub resulting from the application of $\rightsquigarrow_{\text{Outcall}}$ on *add1*. In a real-life setting, we would have made f callable by including it in the export list of the source component, such that an incall stub f would have been created by $\rightsquigarrow_{\text{Incall}}$ as well. We omitted this detail for simplicity's sake.

Finally, the program verification rule ProgVerif allows verification and compilation of entire programs. A program *sprog* has a proof $\vdash$ *sprog* if it is well-formed ($\vdash_{\text{WF}}$ *sprog*, which e.g. states that every imported function should be exported by another component) and if all of its components have a proof. The compilation of a program is constructed from the compilations of its components.

4.4 Stub compilation

As illustrated in Section 2, we require checking functions or stubs in our compiled code to enforce separation logic contracts at trust boundaries, both when receiving an untrusted incall to an exported function and performing an outcall to an untrusted imported function. A verified component of course trusts itself, requiring no stubs when internal calls are performed, as no trust boundary is crossed. We call functions that require the generation of stubs during compilation, that is, functions that are imported or exported by any module, *boundary functions*.

This section discusses how our compiler generates outcall stubs specifically, by means of the Outcall compilation rule in Figure 8. The Incall compilation rule that generates the incall stubs for exported functions is an analogous but simpler version of Outcall and hence not detailed. No Outcall or Incall separation logic rule exists, since stubs are not part of the source code; separation logic contracts are enforced at trust boundaries by the separation logic proof itself.

The Outcall rule generates the outcall stubs for a component's imported functions by defining the previously mentioned compilation rule $csfunc \rightsquigarrow_{\text{Outcall}} ctfunc, stub_{\text{outcall}}$ that both compiles a context function *csfunc* to *ctfunc* using CompFVerif and generates an outcall stub $stub_{\text{outcall}}$ for it. The latter is a wrapper around the outcall to f and reifies f's postcondition in the form of guard statements that check, after f has returned, whether it has upheld its postcondition.

Before generating outcall stubs for imported boundary functions, we make three assumptions on the form of the contracts of boundary functions. These assumptions allow us to easily generate both types of stubs by means of contract reification.

First, we only allow fixed-size, non-conditional array resources $n : exp \mapsto [exp_1, \ldots, exp_k]$ to appear in boundary function contracts. This allows us to do away with quantification in boundary contracts, making the reification of conditions in stubs easier, since every condition ranges over a predetermined set of variables. Since boundary contracts do not contain nested or conditional resources, pre- and postcondition symbolic heaps *PRE* and *POST* are separable into a spatial heap $\overline{m} : PRE_{\text{s}}$ or $\overline{n} : POST_{\text{s}}$, consisting of a sequence of separating-conjunction separated fixed-size array resources

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

(whose names $\overline{n}$ and $\overline{m}$ we externalize in our notation), and a pure heap PRE_p or $POST_\mathrm{p}$, consisting of pure conditions. The OUTCALL rule will make handy use of this separability. Possible measures to weaken this restriction are future work.

Second, we require boundary functions to have *linear contracts*, in the sense that all argument names, all logical variable names in PRE_s and $POST_\mathrm{s}$ and the set of names $\overline{result}$ must be mutually distinct and cannot contain duplicates. This makes any otherwise implicit conditions in PRE_s and $POST_\mathrm{s}$ explicit in PRE_p and $POST_\mathrm{p}$ and hence easier to check. This assumption can be made without loss of generality, as non-linear contracts can easily be linearized. For example, the programmer-written contract for *add1* in Figure 9 is linearized to:

$$//@\mathrm{pre}\ m : a^{\mathrm{pre}} \mapsto_{\mathrm{int}} [a_1^{\mathrm{pre}}] * a^{\mathrm{pre}} == a$$
$$//@\mathrm{post}\ n : a^{\mathrm{post}} \mapsto_{\mathrm{int}} [a_1^{\mathrm{post}}] * result == a_1^{\mathrm{post}} + 1 * a^{\mathrm{post}} == a * a_1^{\mathrm{post}} == a_1^{\mathrm{pre}}\ .$$

Last, we require boundary functions not to introduce any new logical variables (except for $\overline{result}$) in their spatial heaps PRE_p and $POST_\mathrm{p}$. This will make the constraints in PRE_p and $POST_\mathrm{p}$ easier to reify into program-level guards, as all their logical variables either correspond to arguments, result variables or spatial values in the symbolic heap. For example, in the above linearized contract, we can easily access the values for a, a^{pre}, a_1^{pre}, a^{post}, a_1^{post} and *result* in the compiled code. The restrictions imposed by these last two assumptions are included as part of the component well-formedness $\vdash_{\mathrm{WF}} scomp$ in COMPVERIF.

An outcall stub then needs to generate code in order to check any constraints present in the untrusted function's postcondition *POST*.

We first investigate what information from $\overline{m} : PRE_\mathrm{s}$ and $\overline{n} : POST_\mathrm{s}$ we need to reify to be able to check *POST*. Both $\overline{m} : PRE_\mathrm{s}$ and $\overline{n} : POST_\mathrm{s}$ are linear and hence use fresh variable names that can reappear in conditions in $POST_\mathrm{p}$. These variables hence need to be reified, that is, declared and assigned the right values, so they can be used when reifying $POST_\mathrm{p}$'s conditions. Additionally, any constraints present in the linear spatial heap $\overline{n} : POST_\mathrm{s}$ need to be checked in the target language. Remember that an outcall stub solely checks the postcondition. The only information to check is that none of the reified resources $\overline{n}$ can be null, together with the fact that each reified resource n has its correct fixed length k. Both of these checks need to be performed by a guard statement for each n. We need a way to reify the aforementioned checks *check*, declarations *decl* and assignments *assign* for a given spatial assertion $assert_\mathrm{s}$ (i.e. $\overline{m} : PRE_\mathrm{s}$ or $\overline{n} : POST_\mathrm{s}$). This is the function of the auxiliary compilation rule $assert_\mathrm{s} \rightsquigarrow_\mathrm{s} (check, decl, assign)$, defined by RESOURCEREIFY and SEPCONJCREIFY in Figure 8. The *checks* generated for $\overline{m} : PRE_\mathrm{s}$ are simply discarded.

Next, we investigate what information from PRE_p and $POST_\mathrm{p}$ we need to reify to check *POST*. Neither one is allowed to introduce fresh variables, due to the third assumption we made above. Therefore, no declarations or assignments will be reified; only checks. Since outcall stubs only check postconditions, we can disregard PRE_p. All constraints present in $POST_\mathrm{p}$ can simply be reified to guard statements over the identical constraints, due to the three form assumptions on contracts we made above and since we already declared and assigned all variables occurring in these constraints when we reified $\overline{m} : PRE_\mathrm{s}$ and $\overline{n} : POST_\mathrm{s}$ in the previous paragraph. The auxiliary compilation rule $assert_\mathrm{p} \rightsquigarrow_\mathrm{p} check$, defined by CONDITIONREIFY and SEPCONJPREIFY in Figure 8, generates these checks *check* when given a pure assertion $assert_\mathrm{p}$ (i.e. $POST_\mathrm{p}$ in this case).

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

The OUTCALL compilation rule in Figure 8 integrates $\rightsquigarrow_s$, $\rightsquigarrow_p$ and CONTFVERIF to create the outcall stub $stub_{\text{outcall}}$. Precondition-related declarations d_{pre} and assignments a_{pre} happen before the function call to f, since the reified resources $\overline{m}$ might be altered by f. Postcondition-related declarations d_{post}, assignments a_{post} and checks cs_{post} and cp_{post} happen after the call.

The reason the rule IMPLFVERIF renamed every function f to f_{comp} had everything to do with stubs. First of all, any incall stub generated for an exported source function f can now simply be called f and internally call the compiled target function f_{comp}, so that the names of a component's exported functions do not change during compilation. Conversely, outcall stubs for imported functions f are named f_{comp} as well, as OUTCALL demonstrates, so that the FAPP rule does not need to know whether an internal or imported function is being called in order to derive the compiled function's name.

The outcall stub $add1_{\text{comp}}$ for *add1* in Figure 9 gives an example of the generated stub $stub_{\text{outcall}}$ in the OUTCALL compilation rule (compiled using the linearized contract above). As always, the stub has the same declaration (bar function name) as the function *add1* it wraps. The declarations d_{pre} can be found on line 2 and the assignments a_{pre} on line 3. The declarations d_{post} are generated on line 7, assignments a_{post} on line 8 and checks cs_{post} and cp_{post} on line 6 and lines 9–10, respectively.

5 The full abstraction proof

This section summarizes the full abstraction proof for the compiler presented in Section 4. The full abstraction proof by itself takes up roughly 80 pages in the technical report and is therefore too long and detailed to include in this paper. This section and sections 6 and 7 hence summarize the essential concepts in an example-driven fashion, after which the reader should be able to digest the technical report, should they wish to read it.

In the following, notions relating to the source and target languages are typeset in **green** and pink, respectively. The first Subsection 5.1 formally defines both directions of full abstraction. Subsection 5.2 motivates the need for a back-translation and illustrates how both directions of full abstraction can be proved by proving equi-termination between source and target code. Subsection 5.3 further reduces these proofs of equi-termination to a proof of relatedness under specific adequate relations between source and target code.

5.1 Full abstraction definition

To define our notion of full abstraction, we require a notion of *behavioral equivalence*. As is standard in the literature, we define behavioral equivalence to be *contextual equivalence* (Abadi, 1999; Patrignani *et al.*, 2015; New *et al.*, 2016; Devriese *et al.*, 2016). Terms x and x' are contextually equivalent, denoted $x \simeq_{\text{ctx}} x'$, if $\forall C : C[x] \Downarrow \Leftrightarrow C[x'] \Downarrow$ where $\Downarrow$ denotes termination of execution and C is any program context with a hole that x and x' can be plugged into. Both x and x' are either source or target components in our case. A context C consists of two parts in both source and target languages: a *component context* $\mathfrak{C}_s$ or $\mathfrak{C}_t$, which is a sequence of components, and a *main function identifier*, denoted by the metavariable *id*, identifying the main function to execute when starting program execution. A context is hence denoted $(\mathfrak{C}, id)$ and an entire program $\mathfrak{C}[x]//@\text{main} = id$. In

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

our source language, the notion of *plugging* from the contextual equivalence definition above also requires (in addition to the program well-formedness constraints denoted by $\vdash_{\mathrm{WF}} \mathit{scomp}$ in Figure 7 in the previous section) that given the source component proof $\vdash \mathbf{s}$ and the context $(\mathfrak{C}_s, \mathit{id})$, a program proof $\vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = \mathit{id}$ exists. The notion of plugging in the target language solely requires program well-formedness, expressed by a similar judgment $\vdash_{\mathrm{WF}} \mathit{tcomp}$ for target programs, defined in the technical report.

Full abstraction is then defined as the reflection and preservation of contextual equivalence $\simeq_{\mathrm{ctx}}$ (Abadi, 1999). Given source components $\mathbf{s}$ and $\mathbf{s}'$ and target components t and t', we have that compilation is fully abstract iff $\vdash \mathbf{s} \rightsquigarrow \mathrm{t}$ and $\vdash \mathbf{s}' \rightsquigarrow \mathrm{t}'$ imply that $(\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}' \Leftrightarrow \mathbf{s} \simeq_{\mathrm{ctx}} \mathbf{s}')$. This statement depends on the chosen proofs $\vdash$ of $\mathbf{s}$ and $\mathbf{s}'$, but has to hold for *any* such choice. Notice how our formulation of full abstraction does not make a distinction between code that gets stuck and code that diverges. In other words, diverging source code could in theory be compiled to target code that gets stuck. This is, however, not a real concern; since our compiler does not alter control flow, it should be easy to prove that it preserves divergence and stuckness individually, if so desired.

Fully abstract compilation proofs are usually split up in a correctness proof direction $\Rightarrow$ that states (by contraposition) that non-equivalent source programs should yield non-equivalent target programs and a security proof direction $\Leftarrow$ that (by contraposition) states that any non-equivalence in the target programs should already have been there in the source programs, and hence attackers have no more power in the absence of contracts than they do in their presence. Both proof directions are summarized by the following equations:

$$\forall\, \mathbf{s}, \mathbf{s}', \mathrm{t}, \mathrm{t}'.\, \vdash \mathbf{s} \rightsquigarrow \mathrm{t} \;\wedge \vdash \mathbf{s}' \rightsquigarrow \mathrm{t}' \Rightarrow (\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}' \Rightarrow \mathbf{s} \simeq_{\mathrm{ctx}} \mathbf{s}') \qquad \textbf{(CORRECTNESS)}$$
$$\forall\, \mathbf{s}, \mathbf{s}', \mathrm{t}, \mathrm{t}'.\, \vdash \mathbf{s} \rightsquigarrow \mathrm{t} \;\wedge \vdash \mathbf{s}' \rightsquigarrow \mathrm{t}' \Rightarrow (\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}' \Leftarrow \mathbf{s} \simeq_{\mathrm{ctx}} \mathbf{s}') \qquad \textbf{(SECURITY)}$$

5.2 Full abstraction as source-to-target equi-termination

We first dissect the **CORRECTNESS** direction above. To prove $\mathbf{s} \simeq_{\mathrm{ctx}} \mathbf{s}'$, we need to prove (by definition) that $\mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = \mathit{id} \Downarrow \Leftrightarrow \mathfrak{C}_s[\mathbf{s}']//@\mathrm{main} = \mathit{id} \Downarrow$, given any source context $(\mathfrak{C}_s, \mathit{id})$ such that we can construct proofs $\vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = \mathit{id}$ and $\vdash \mathfrak{C}_s[\mathbf{s}']//@\mathrm{main} = \mathit{id}$ from $\vdash \mathbf{s}$ and $\vdash \mathbf{s}'$. If we can prove that verified code and its compilation equi-terminate, that is, if it holds for any contexts $(\mathfrak{C}_s, \mathit{id})$ and $(\mathfrak{C}_t, \mathit{id})$ and any components $\mathbf{s}$ and t (with $\vdash \mathbf{s} \rightsquigarrow \mathrm{t}$) that:

$$\vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = \mathit{id} \rightsquigarrow \mathfrak{C}_t[\mathrm{t}]//@\mathrm{main} = \mathit{id} \Rightarrow (\mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = \mathit{id} \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathrm{t}]//@\mathrm{main} = \mathit{id} \Downarrow) \qquad \textbf{(COMP-}\Updownarrow\textbf{)}$$

then we can prove $\mathbf{s} \simeq_{\mathrm{ctx}} \mathbf{s}'$. The reason is that we know from $\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}'$ that $\mathfrak{C}_t[\mathrm{t}]//@\mathrm{main} = \mathit{id}$ and $\mathfrak{C}_t[\mathrm{t}']//@\mathrm{main} = \mathit{id}$ equi-terminate, and hence that:

$$\mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = \mathit{id} \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathrm{t}]//@\mathrm{main} = \mathit{id} \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathrm{t}']//@\mathrm{main} = \mathit{id} \Downarrow \Leftrightarrow \mathfrak{C}_s[\mathbf{s}']//@\mathrm{main} = \mathit{id} \Downarrow.$$

We would like to repeat the above process for the **SECURITY** direction, that is, prove $\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}'$ through some form of equi-termination between source and target code. To prove $\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}'$, we need to prove (by definition) that

$$\mathfrak{C}_t[\mathrm{t}]//@\mathrm{main} = \mathit{id} \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathrm{t}']//@\mathrm{main} = \mathit{id} \Downarrow$$

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

given *any* target context $(\mathfrak{C}_t, id)$ and source component proofs $\vdash \mathbf{s}$ and $\vdash \mathbf{s}'$. There is, however, one problem: since we start from a target context $(\mathfrak{C}_t, id)$ rather than a source context $(\mathfrak{C}_s, id)$, we cannot use our compiler to construct an equi-terminating source program for us. Simply inverting the compilation function is impossible, since it is not a bijection; the compiler's range is a strict subset of the target language. Hence, we need a new transformation, this time from target to source, to create equi-terminating source code, starting from any target-context $(\mathfrak{C}_t, id)$ and a source component proof $\vdash \mathbf{s}$. This target-to-source code transformation is called the *back-translation*, denoted $\vdash \mathbf{s}, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id$, and is a standard tool in full abstraction proofs. The proof $\vdash \mathbf{s}$ is necessary because the back-translated context $(\mathfrak{C}_s, id)$ needs to result in a sound program proof $\vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id$ (since $\mathbf{s} \simeq_{\mathrm{ctx}} \mathbf{s}'$ requires verified code). To back-translate individual target statements *tstm*, no such proof is required and we hence simply write $tstm \rightsquigarrow_b \vdash sstm$.

Having introduced a back-translation, we can repeat the process used to prove **CORRECTNESS**. If we can prove that target code and its back-translation equi-terminate, that is, if it holds for any contexts $(\mathfrak{C}_s, id)$ and $(\mathfrak{C}_t, id)$ and any components $\mathbf{s}$ and t (with $\vdash \mathbf{s} \rightsquigarrow \mathrm{t}$) that:

$$\begin{array}{l} \vdash \mathbf{s}, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id \;\Rightarrow \\ (\mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id \;\Downarrow \Leftrightarrow \mathfrak{C}_t[\mathrm{t}]//@\mathrm{main} = id \;\Downarrow) \end{array} \qquad \textbf{(BT-}\Updownarrow\textbf{)}$$

Then $\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}'$. The reasoning is analogous to the one for **CORRECTNESS** above.

For the sake of brevity, we introduce short notations for the compilation of general source code $\mathfrak{s}$ and the back-translation of target contexts $\mathfrak{C}_t$:

- The target code $[\![\vdash \mathfrak{s}]\!]$ denotes the result t of the compilation $\vdash \mathfrak{s} \rightsquigarrow \mathrm{t}$.
- The source context (without proof) $\langle\!\langle \vdash \mathbf{s}, (\mathfrak{C}_t, id) \rangle\!\rangle$ denotes the context $\mathfrak{C}_s$ in the back-translation $\vdash \mathbf{s}, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id$. To avoid notational clutter, we usually simply write $\langle\!\langle \mathfrak{C}_t \rangle\!\rangle$ when $\vdash \mathbf{s}$ is clear from context.

5.3 Proof decomposition: Relational view

This subsection provides a more detailed account of the proofs of **CORRECTNESS** and **SECURITY** by further decomposing their proof obligations **COMP-**$\Updownarrow$ and **BT-**$\Updownarrow$ from the previous subsection. The proof schemata in Figures 13 and 14 (inspired by the schemata of Devriese *et al.*, 2016) illustrate this decomposition graphically. The equi-termination in both **COMP-**$\Updownarrow$ and **BT-**$\Updownarrow$ is proved by using two auxiliary relations R and S, for the correctness and security directions of full abstraction, respectively. Both R and S are binary relations that relate source language states to target language states during execution. These states are of the same form as the program states in the operational semantics, $\langle \overline{s}, h \rangle \mid \overline{c}$. Again, $\overline{c}$ is either a sequence of partially executed function bodies, or an entire program.

These relations internally make use of *simulation relations* (see e.g. Chlipala, 2017), to capture the lock-step execution of source and target code. However, R and S are not technically simulation relations themselves (see Section 7). In this section, we consider both relations to be black boxes.

One important caveat should already be made regarding the source language states: since our compilation is separation-logic-driven, the target language states mirror the states of

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

the separation logic *proof* in the source language, and not just the states of the executing source *code* itself. For example, at any given point during execution of code and its compilation, the linear capabilities present in the heap and stack in the target language will correspond to the separation logic resources present in the proof of the current source code. An analogous argument holds for the back-translation; the linear capabilities in the target language are reflected as separation logic resources in the source language at any point during execution and hence require the inclusion of a proof into the source-language states.

As an example of why raw, unverified source code does not suffice to define R (or S), consider the following single-statement verified source program $\vdash \mathbf{s}$:

$$\{n : a \mapsto_{\text{int}} [2]\}_{\bullet[a:a]} \quad \text{a}[0] = 3 \quad \{n : a \mapsto_{\text{int}} [3]\}_{\bullet[a:a]}$$

and its compilation $[\![\vdash \mathfrak{s}]\!] \equiv$ n[0] = 3. In the correctness setting, that is, when relating $\vdash \mathbf{s}$ and $[\![\vdash \mathfrak{s}]\!]$ through R, we need to relate the contents of the linear capability n in the target to the separation logic resource n in the source. If we solely used the raw source code without proof as the source state in R, that is, the program a[0] = 3, then it would be impossible to know what part of the source heap the target linear capability n corresponds to because we erased the connection between n and a by erasing the verification proof. Additionally, if R would not constrain the contents of both to correspond and be equal to the single element 2 (in the target-level stack and the separation logic proof, respectively), simulation would get stuck if, for example, a conditional statement was encountered that checked whether $a[0] == 2$ in the source language (and hence whether $n[0] == 2$ in the target). Of course, in addition to enforcing this correspondence between source proof and target-level capabilities, R and S will also need to relate the concrete stack and heap in the source language to the current logical state in the precondition. More details about this can be found in the technical report.

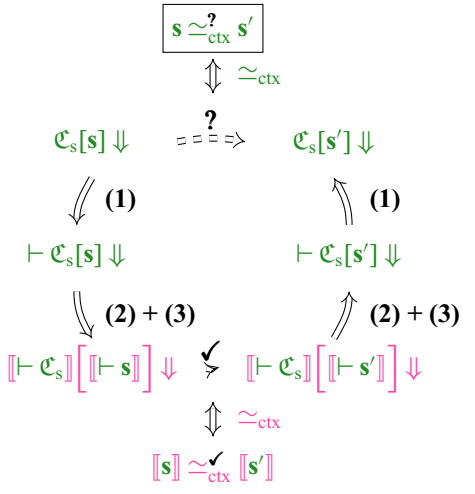
In order to relate source and target states as they execute, we need a notion of separation logic proof that evolves along with the executing source code. We obtain this by *lifting* the source-level operational semantics to the verified source code. This new *lifted operational semantics* is detailed in the technical report. The definition relies on the property of *proof preservation* (the analog to type preservation in type systems, see e.g. Pierce, 2002). The property states the following: if we have a transition $\langle \bar{\mathbf{s}}, \mathbf{h} \rangle \mid \bar{\mathbf{c}} \hookrightarrow \langle \bar{\mathbf{s}}', \mathbf{h}' \rangle \mid \bar{\mathbf{c}}'$ in the non-lifted operational semantics, and proofs $\overline{\vdash \mathbf{c}}$, then the resulting program $\bar{\mathbf{c}}'$ is also provable, that is, we can construct proofs $\overline{\vdash \mathbf{c}'}$. Using this lemma, the lifted operational semantics essentially lets verified programs $\vdash \mathbf{c}$ step to $\vdash \mathbf{c}'$.

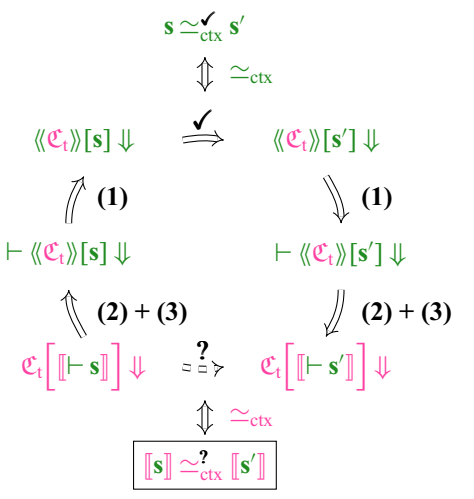
To better understand the lifted semantics, let us consider the execution of $\vdash \mathbf{s}$ and $[\![\vdash \mathfrak{s}]\!]$ in the above example. Assuming appropriate stacks and heaps, both $\mathbf{s}$ (the source program without proof) and $[\![\vdash \mathfrak{s}]\!]$ evaluate to a single skip statement in one step (cfr. the ARRAYMUT rule in Figure 4). Proof preservation updates the proof of $\vdash \mathbf{s}$ to a proof of the resulting skip statement:

$$\{n : a \mapsto_{\text{int}} [3]\}_{\bullet[a:a]} \quad \text{skip} \quad \{n : a \mapsto_{\text{int}} [3]\}_{\bullet[a:a]}$$

Note how we have "executed" the precondition of $\vdash \mathbf{s}$ to match the now executed array mutation.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press





(1) $\vdash sprog \Downarrow \Leftrightarrow sprog \Downarrow$ **(COHERENCE)**

(2) $\vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id \rightsquigarrow [\![\vdash \mathfrak{C}_s]\!]\big[[\![\vdash \mathbf{s}']\!]\big]//@\mathrm{main} = id$ $\Rightarrow$

$(\langle \bullet, \bullet \rangle \mid \vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id\ )\ R$

$(\langle \bullet, \bullet \rangle \mid [\![\vdash \mathfrak{C}_s]\!]\big[[\![\vdash \mathbf{s}]\!]\big]//@\mathrm{main} = id\ )$

(COMPATIBILITY)

(3) $(\langle \bullet, \bullet \rangle \mid \vdash sprog)\ R\ (\langle \bullet, \bullet \rangle \mid tprog)$ $\Rightarrow$ $\vdash sprog \Downarrow \Leftrightarrow tprog \Downarrow$

(ADEQUACY)

Fig. 13. **CORRECTNESS** proof outline.

(1) $\vdash sprog \Downarrow \Leftrightarrow sprog \Downarrow$ **(COHERENCE)**

(2) $\vdash \mathbf{s}, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \langle\!\langle \mathfrak{C}_t \rangle\!\rangle[\mathbf{s}]//@\mathrm{main} = id$ $\Rightarrow$

$(\langle \bullet, \bullet \rangle \mid \vdash \langle\!\langle \mathfrak{C}_t \rangle\!\rangle[\mathbf{s}]//@\mathrm{main} = id\ )\ S$

$(\langle \bullet, \bullet \rangle \mid \mathfrak{C}_t\big[[\![\vdash \mathbf{s}]\!]\big]//@\mathrm{main} = id\ )$

(COMPATIBILITY)

(3) $(\langle \bullet, \bullet \rangle \mid \vdash sprog)\ S\ (\langle \bullet, \bullet \rangle \mid tprog)$ $\Rightarrow$ $\vdash sprog \Downarrow \Leftrightarrow tprog \Downarrow$

(ADEQUACY)

Fig. 14. **SECURITY** proof outline.

Now that we understand how R and S relate source states of the form $\langle \bar{\mathbf{s}}, \mathbf{h} \rangle \mid \overline{\vdash \mathbf{c}}$ (where $\overline{\vdash \mathbf{c}}$ is either a monolithic, verified source program or a sequence of verified, partially executed function bodies) with target states of the form $\langle \bar{\mathrm{s}}, \mathrm{h} \rangle \mid \bar{\mathrm{c}}$, let us take a close look at Figures 13 and 14. Note how their visual similarity illustrates the similarities between the two proof directions. For compactness' sake, the main function specification $//@main = id$ has been left out of the source and target program descriptions in both schemata, for example abbreviating $\vdash \mathfrak{C}_s[\mathbf{s}]//@\mathbf{main} = \mathbf{id}$ to $\vdash \mathfrak{C}_s[\mathbf{s}]$ and $\mathfrak{C}_t[\mathrm{t}]//@\mathrm{main} = \mathrm{id}$ to $\mathfrak{C}_t[\mathrm{t}]$. The proof steps are denoted by arrows $\Rightarrow$, where a ✓ denotes a given and **?** a proof obligation. The contextual equivalence we need to prove has been boxed and is situated across from the given contextual equivalence. The proof in both proof schemata starts at the left side of the dashed $\overset{?}{\Rightarrow}$ and traces the entire circle before arriving at its right side. For both correctness and security, all proof steps $\Rightarrow$ are explained by either the definition of contextual equivalence $\simeq_{\mathrm{ctx}}$, or one of a set of three auxiliary

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

lemmas. These three lemmas are similar between correctness and security and numbered **(1)**, **(2)** and **(3)** in both. Notice how the number-annotated proof steps, considered in isolation, indeed constitute a decomposition of the proof obligations **COMP-⇳** and **BT-⇳** in the respective figures. For example, starting from the top left corner in Figure 13, consecutively applying **(1)**, **(2)** and **(3)** and ending up in the bottom left corner, corresponds to the left-to-right implication in **COMP-⇳**. Starting from the bottom right and moving upwards results in the right-to-left implication direction. We now discuss the role of both contextual equivalence and the three lemmas in order.

The arrows annotated with $\simeq_{\mathbf{ctx}}$ and $\simeq_{\mathrm{ctx}}$ denote an application of the definition of source- and target-level contextual equivalence, respectively. The universal quantification over (well-formed) contexts $(\mathfrak{C}_s, id)$ and $(\mathfrak{C}_t, id)$ is left implicit in the unfolding of the definition. For correctness (and similarly for security), the implication in the proof obligation $\mathfrak{C}_s[\mathbf{s}] \Downarrow \stackrel{?}{\Rightarrow} \mathfrak{C}_s[\mathbf{s}'] \Downarrow$ is sufficient to prove contextual equivalence in the source language. The other direction follows by symmetry.

The **COHERENCE** lemma **(1)** states that source programs in the regular operational semantics (i.e. without their proofs) and the same source programs in the lifted operational semantics (i.e. including their proofs) equi-terminate. This lemma allows adding proofs to source programs and conversely stripping them away, all the while preserving termination. This conversion is necessary since $\simeq_{\mathbf{ctx}}$ is defined using the regular operational semantics, whereas the relations S and R make use of the lifted semantics.

The **COMPATIBILITY** **(2)** and **ADEQUACY** **(3)** lemmas are used in combination to prove equi-termination between a source program and its compilation in the correctness case, and between a target program and its back-translation in the security case. **COMPATIBILITY** proves that any source program and its compilation are related by R under the empty stack and heap for correctness and that any target program and its back-translation are related by S under the empty stack and heap for security. **ADEQUACY** finishes the combined equi-termination argument by stating that source and target programs related by S or R equi-terminate (from the empty stack and heap). The proof of **ADEQUACY** follows straightforwardly from the fact that S and R internally make use of simulation relations. This will be clarified further in Section 7.

6 Proving security: the back-translation

Similarly to how Section 4 introduced compilation using Figure 9, this section will introduce the back-translation by means of an example, that builds on top of the compilation example. The goal of this section is to illustrate how the back-translation of this example satisfies one specific instance of **BT-⇳** from the previous section, in the process highlighting the key concepts behind the back-translation. Concretely, we will back-translate a target-level implementation of the context function *add1* from Figure 9 and provide intuitions for why the back-translation and the original implementation equi-terminate. The concrete implementation of *add1* that we will back-translate is given in the bottom-right of Figure 16.

Let us identify what concrete instance of **BT-⇳** we have to prove here. The verification of f from Figure 9 acts as our verified component $\vdash \mathbf{s} = \vdash (f \ //@\mathrm{import}\ add1\ )$. Note the

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

```
void main()
//@pre true
//@post true {
  int* a; a = malloc(2 * sizeof(int));
  a[0] = 0; a[1] = 1;
  int res; res = f(a);
  return }
```

Fig. 15. Example implementation of a main function wrapping f from Figure 9.

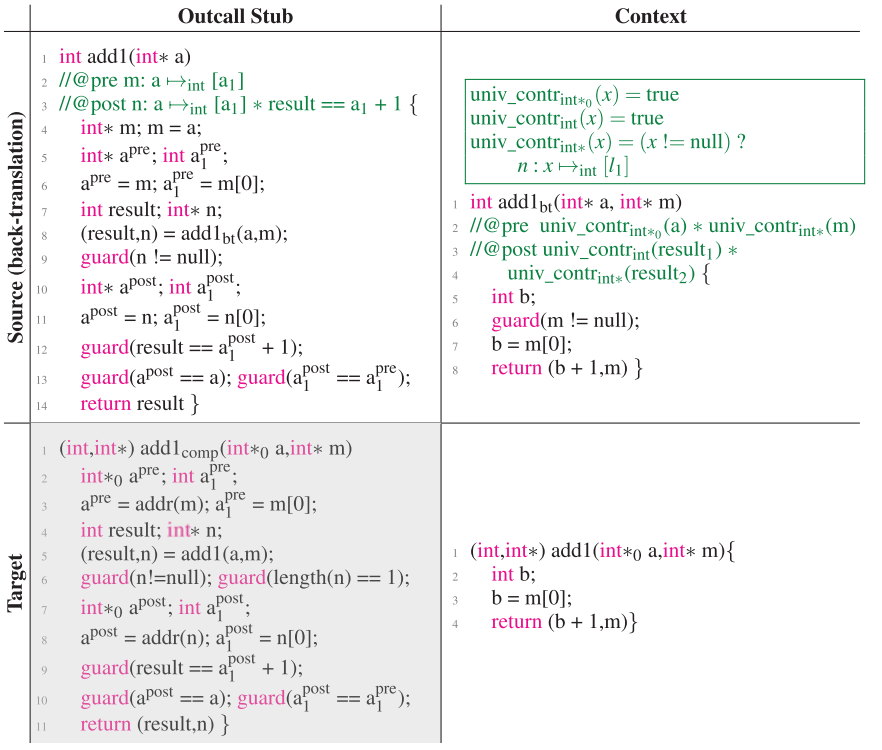


Fig. 16. Illustrative example: Naive back-translation of a context that implements *add1*.

slight abuse of notation, where we use the name of a function (f) for its entire implementation. Consequently, f_{comp} and $add1_{\text{comp}}$ together act as our compiled target component $\mathrm{t} = [\![\vdash \mathbf{s}]\!] = f_{\text{comp}}\ add1_{\text{comp}}$ //@import *add1* . Our target context $(\mathfrak{C}_{\mathrm{t}}, id)$ then consists of the target function *add1*'s implementation in Figure 16 and an arbitrary choice for the main function *id*.

Since the main function in our formalization is not allowed to have arguments, needs to have void as its return type and needs to be exported by some component, we cannot have f_{comp} nor *add1* as a main function. Instead, we could, for example, add an exported function *main* to $\vdash \mathbf{s}$, solely serving as a wrapper for f, with an implementation as shown in Figure 15.

This function *main* is compiled by our compiler to a function $main_{\text{comp}}$, and since it is an exported function, an incall stub (without guard statements, since the precondition of *main* is true) *main* is generated, which is the target main function and would in turn have to be

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

back-translated as well. However, to not needlessly clutter our example, we simply assume the main function in both source and target languages to be called *main*, respectively *main*, and do not explicitly represent or back-translate this main function anywhere.

In other words, $(\mathfrak{C}_t, id) = ((add1\ //@\text{export}\ add1\ ),\ main)$, where we will not explicitly write *main* in t. Notice how t and $(\mathfrak{C}_t, id)$ together form a sound target-program, as required by the definition of contextual equivalence $t \simeq_{ctx} t'$ in Section 5.2. To prove this specific instance of (**BT-⇕**), we have to back-translate the example context $(\mathfrak{C}_t, id)$ to a context $(\mathfrak{C}_s, id)$ such that $\vdash \mathfrak{C}_s[\mathbf{s}]//@\text{main} = id$ is a valid separation logic proof, and $\mathfrak{C}_s[\mathbf{s}]//@\text{main} = id$ and $\mathfrak{C}_t[t]//@\text{main} = id$ equi-terminate. As we will see, $\mathfrak{C}_s$ will contain both a back-translation $add1_{bt}$ of *add1* and a back-translation *add1* of the outcall stub $add1_{comp}$, resulting in the source context

$$(\mathfrak{C}_s, id) = (\langle\!\langle \mathfrak{C}_t \rangle\!\rangle, id) = ((add1\ add1_{bt}\ //@\text{export}\ add1\ ),\ main)$$

where, again, we will not explicitly write *main* in $\vdash \mathbf{s}$.

Notice how the implementation of the context function *add1* in Figure 16 upholds the source-level contract that the verified component $\vdash \mathbf{s}$ expects of *add1* in Figure 9: it reads the first element of resource *m*, increments and returns it, together with resource *m*, without altering the address of *m* or its contents. The context $(\mathfrak{C}_t, id)$ is *not* required to behave properly like this! It might also add 2 instead, change the contents of the resource *m*, etc., effectively ignoring the expectations of $\vdash \mathbf{s}$ and causing execution to get stuck at the guard statements of the outcall stub $add1_{comp}$. This extra freedom of the target context to misbehave, and the requirement for guard statements to detect such misbehavior, is at the core of the full abstraction proof: the security proof direction states that we can reinterpret (i.e. back-translate) even possibly misbehaving contexts as equi-terminating, verified source contexts that are incapable of breaking the verification guarantees of $\vdash \mathbf{s}$. Section 6.3 will further illustrate this point, by briefly demonstrating the back-translations of a few misbehaving implementations of *add1*.

The remainder of this section introduces the back-translation incrementally, introducing key concepts gradually. First, Section 6.1 starts off with a naive back-translation of *add1* that does not generalize to arbitrary code. After pointing out some problems in generalizing this back-translation, Section 6.2 introduces a version that works for any target code not using nested pointer types (such as int∗∗) in the target language. As mentioned, Section 6.3 briefly investigates the back-translation of misbehaving contexts. Finally, Section 6.4 sketches what the most general back-translation looks like by investigating the back-translation of nested pointers.

6.1 Naive back-translation

This subsection constructs a naive version of the back-translation of *add1* from Figure 16 and discusses the results in the **Source** row of this same figure. The back-translation is *naive* because it assumes a statically known size of 1 for all target capabilities (the *naive assumption*) and does not support back-translating nested pointer types. These assumptions are unproblematic for *add1*.

We first define the back-translation of types and expressions. Target types that can result from compilation of source types, that is, int, length-0 capabilities $\tau_s *_0$ and tuples (τ_t^*), are

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

simply back-translated inversely to how they are compiled. However, linear capabilities are reified resources and did not originally exist in the source language, so we have to come up with a way to represent them.

Target linear capabilities $l^{[a,b]}$ inherently contain both an address l and a length $b - a + 1$, as discussed in Section 3.3. To extract these, the target language contains built-in addr and length functions. Pointers in the source language are of the form (l, i), are non-linear and do not have built-in length information, and the source language does not have (or need) addr or length functions. Fortunately, the naive back-translation assumes all target pointers to have a statically known length of 1, so there is no need to keep any length information in the source language. It is hence possible to simply back-translate linear capabilities of type $\tau*$ to source-level pointers $\tau'*$, where τ' is obtained by recursively back-translating τ. The pointer $\tau'*$ simultaneously represents the back-translated address, since $\tau'*$ is non-linear and allows for pointer arithmetic. Both the length and the address information of each capability are hence retained during back-translation. These back-translations of target types are formalized by the judgment $\tau \rightsquigarrow_{\text{InvCompileType}} \tau'$, dual to $\tau \rightsquigarrow_{\text{CompileType}} \tau'$ from Section 4.2, that recursively back-translates target types as follows:

$$\frac{}{\text{int} \rightsquigarrow_{\text{InvCompileType}} \text{int}}\ (\textsc{InvCompileInt}) \qquad \frac{\tau' \rightsquigarrow_{\text{InvCompileType}} \tau}{\tau'*_0 \rightsquigarrow_{\text{InvCompileType}} \tau*}\ (\textsc{InvCompileSrcPtr})$$

$$\frac{\tau_1 \rightsquigarrow_{\text{InvCompileType}} \tau_1' \quad \dots \quad \tau_k \rightsquigarrow_{\text{InvCompileType}} \tau_k'}{(\tau_1, \dots, \tau_k) \rightsquigarrow_{\text{InvCompileType}} (\tau_1', \dots, \tau_k')}\ (\textsc{InvCompileTuple}) \qquad \frac{\tau \rightsquigarrow_{\text{InvCompileType}} \tau'}{\tau* \rightsquigarrow_{\text{InvCompileType}} \tau'*}\ (\textsc{InvertCapability})$$

Section 6.2 will scrap the naive assumption and will therefore have to introduce a more involved back-translation for pointer types, retaining length information.

Unfortunately, the back-translated linear capabilities will not automatically behave linearly. Therefore, our back-translation needs to simulate their linear behavior. Extra statements have to be added during the back-translation to imitate the target-language erasure of capabilities. For example, when back-translating the assignment $x = y$ with y of type int*, an erasure assignment $y =$ null will be added in the source. Similarly, once we support back-translating nested pointers in Section 6.4, back-translating the assignment $x = n[2]$ with n of target type int** produces an erasure assignment $n[2] =$ null in the source. Additionally, since the target language gets stuck when the same target capability is used twice in one statement, the back-translation then has to artificially get stuck as well. For example, when back-translating the assignment $x = (y, y)$, with y as before, the source language needs to add a statement guard(false) to ensure equi-termination.

The back-translation of expressions *texp*, denoted $texp_{\text{b}}$, is now easy to define, since target and source expressions only differ in the addr and length functions mentioned before. The back-translation hence maps addr(*texp*) to $texp_{\text{b}}$ (the pointer doubles as address, since it is copyable) and length(*texp*) to ($texp_{\text{b}}$!= null) (all non-null capabilities are assumed to have length 1, whereas null has length 0). All other cases proceed structurally.

With these prerequisites out of the way, we can examine the back-translation $\mathfrak{C}_{\text{s}}$ of the target context $\mathfrak{C}_{\text{t}}$, shown in the top row of Figure 16. As mentioned, this back-translation consists of two separate parts: not only the expected back-translation $add1_{\text{bt}}$ of the target

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

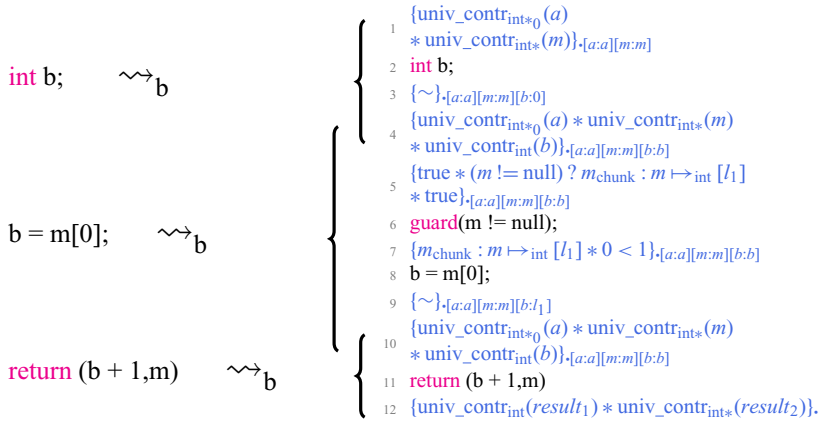


Fig. 17. Separation logic proof of the body of $add1_{\mathrm{bt}}$ from the naive back-translation example.

function *add1* on the right, but also a back-translation *add1* of the previously generated stub $add1_{\mathrm{comp}}$ on the left. Conceptually, the universal contract of $add1_{\mathrm{bt}}$ captures how the use of linear capabilities in the target language restricts the behavior of the context function *add1*, whereas guards in *add1* will enforce further functional conditions that f expects of *add1*. The function *add1* is derived from $add1_{\mathrm{bt}}$, using a separate back-translation for stubs that we will motivate below. First, let us investigate how regular target functions such as *add1* are back-translated.

Essentially, our goal is to understand Figure 17: the back-translation equivalent of the proof of $add1_{\mathrm{bt}}$ in Figure 10. The notation $\sim$ is used to denote an unaltered symbolic heap. A first question is what the separation-logic contract of the back-translated function $add1_{\mathrm{bt}}$ should be. The desired contract is the one $\vdash \mathbf{s}$ expects for *add1* in Figure 9, so that the resulting source program $\mathfrak{C}_{\mathrm{s}}[\mathbf{s}]//@\mathrm{main} = id$ has a sound verification. However, since the target context *add1* can freely misbehave as we saw earlier, proving this contract will in general be impossible. Additionally, if the context contains functions that are not imported by $\vdash \mathbf{s}$, there are no restrictions on their contract whatsoever.

The solution is to employ the most general admissible contract for the back-translation and adapt it to the expected contract separately (see below). This contract will express the permissions associated with target-language capabilities as separation logic resources in the source. Combining the resources represented by all arguments of a back-translated function gives us its precondition and the resources represented by its result are the post-condition. We call this type of contract a *universal contract* and define it in the next paragraph. Lines 2–3 of $add1_{\mathrm{bt}}$ in Figure 16 show the universal contract for $add1_{\mathrm{bt}}$.

Universal contracts $\mathrm{univ\_contr}_{\tau_{\mathrm{t}}}$ are predicates on logical expressions *exp*. They are indexed by the target type τ_{t} of id_{p}, since this type determines the target-language permissions associated with the variable. Universal contracts are separation logic assertions and hence cannot take program variables like id_{p} as a direct argument. The universal contract for a program variable id_{p} of type τ_{t} is obtained by applying $\mathrm{univ\_contr}_{\tau_{\mathrm{t}}}$ to the variable's logical interpretation $\gamma(id_{\mathrm{p}})$.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

We now present a simplified definition for universal contracts, which we will expand upon in Sections 6.2 and 6.4:

Definition ($\text{univ\_contr}_{\tau_t}(exp)$)**.**

$$\begin{aligned}
\text{univ_contr}_{\tau_t}(exp) &\triangleq \text{true } \textit{if } \tau_t = \text{int } \textit{or } \tau_t = \tau_s* _0 \\
\text{univ_contr}_{(\tau_1,\ldots,\tau_k)}(exp) &\triangleq \text{univ_contr}_{\tau_1}(exp.1) * \ldots * \text{univ_contr}_{\tau_k}(exp.k) \\
\text{univ_contr}_{\tau_t*}(exp) &\triangleq (exp \mathrel{!}= \text{null}) \,?\, \exists l_1.\, n : exp \mapsto_{\tau_s} [l_1] \\
&\qquad \text{given that } \tau_t \rightsquigarrow_{\text{InvCompileType}} \tau_s \text{ and } n \text{ fresh}
\end{aligned}$$

The case for target capabilities τ_t* is the only non-trivial one. It states that a linear capability is either the null-pointer or that it has length one (per the naive assumption made before) and allows access to its unspecified contents l_1.

Universal contracts are now used to back-translate each target-level statement to a block of verified source code. Both the separation-logic pre- and postconditions of such blocks consist of the separating conjunction of universal contracts for all declared target-level program variables. The universal contract will hence monotonically increase throughout the proof; if a *tstm* declares a set of variables V_{tstm}, if variables V_{pre} were previously declared and if $tstm \rightsquigarrow_b \vdash sstm$ holds, then *sstm* has as contract (omitting type subscripts)

$$\{\text{univ\_contr}(\gamma_{\text{pre}}(\text{V}_{\text{pre}}))\}_{\gamma_{\text{pre}}}\ sstm\ \{\text{univ\_contr}(\gamma_{\text{post}}(\text{V}_{\text{pre}} \cup \text{V}_{tstm}))\}_{\gamma_{\text{post}}}$$

One of the main efforts in defining the back-translation is proving that the above Hoare triple indeed holds for all single-statement back-translation rules. Figure 17 demonstrates this block-level proof; the three statements (including return) of *add1* are back-translated to the three annotated blocks on lines 1–4, 4–10 and 10–12 in the proof.

These separate proof blocks offer the advantage that the back-translation can be proven modularly, on a block-per-block basis, since we already know that the last block's postcondition and the next blocks's precondition will correspond. At the start of function execution, only the arguments id_{arg} have been declared; the precondition of a back-translated function is hence $\text{univ\_contr}(\overline{id_{\text{arg}}})$, as demonstrated on line 1 of Figure 17. The postcondition is an exception, since the caller only cares about the privileged $\overline{result}$ variables and resources over them. The universal postcondition is hence $\text{univ\_contr}(\overline{result})$, as demonstrated on line 12 of Figure 17. This postcondition is achieved through CONSEQ once the function's body (without the return statement) is proven.

Since each back-translated block of code has to start and end in a universal contract, each block consists of three separate phases. These three phases are summarized in Figure 18. To illustrate this figure, we investigate the back-translation of the array lookup on line 2 in *add1*, that is, the middle block on lines 4–10 in Figure 17.

First, to match the precondition of the ARRAYLKUP separation logic rule, we need a resource different from the null pointer and we need to know that our index (0 here) is within the bounds of our array. This last fact follows automatically from the naive assumption. Since the universal contract does not provide us guarantees about the pointer *m* not being null, lines 4–7 add this condition through a guard and derive the necessary preconditions for ARRAYLKUP using CONSEQ. Failing this inserted guard statement would make the back-translated program get stuck. This is desired behavior, preserving

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

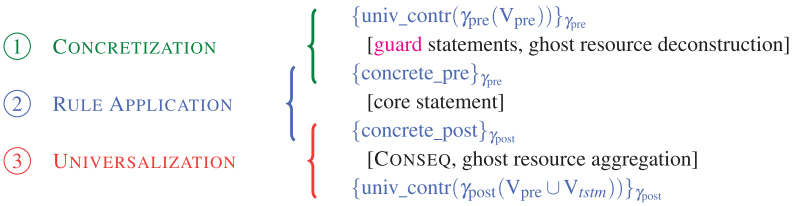


Fig. 18. Illustration of the general three-phase structure of back-translation rules.

equi-termination, since the target operational semantics would also get stuck in this faulty case. To summarize, this first phase, called CONCRETIZATION in Figure 18, starts from the universal contract and transforms it into the precondition concrete_pre of the separation logic rule we actually want to apply. As we will discuss in Section 6.2, this phase will also include transformations on ghost resources in the general case.

Second, the core rule ARRAYLKUP is applied on lines 7–9, leaving us with the concrete postcondition concrete_post. This second phase is called RULE APPLICATION in Figure 18. Finally, on lines 9–10, CONSEQ is applied to forget the information added through the guard statement and the array lookup, making the postcondition universal again. This third phase is called UNIVERSALIZATION in Figure 18 and transforms concrete_post back into a universal contract. As with CONCRETIZATION, transformations on ghost resources can be required in more complex cases. In general, it is non-trivial to see that the CONSEQ rule will always suffice to prove UNIVERSALIZATION. In the technical report, this fact is proven for each back-translation rule individually, by making use of Theorems 5 and 6.

The back-translations of other statements follow the same structure of Figure 18. For some simple statements, the first phase might be empty. Such is the case for, for example, the return block on lines 10–12 of Figure 17.

The aforementioned simulation of linearity in the source language has so far been swept under the rug in this discussion. Concretely, the UNIVERSALIZATION phase inserts erasure statements for linear capabilities, whereas the CONCRETIZATION phase makes sure that guard(false) statements are inserted when linear capabilities would otherwise be duplicated.

We can now back-translate regular target functions, but the resulting universal contracts do not match the concrete contracts that the source context $\vdash \mathbf{s}$ expects, for example, the contract for *add1* in Figure 9 does not match the universal contract of $add1_{\text{bt}}$ in Figure 16. When f performs an outcall to *add1*, the gap between the preconditions of *add1* and $add1_{\text{bt}}$ needs to be bridged: the back-translated argument m needs to be constructed and the universal contracts univ_contr of m and a need to be satisfied, starting from the concrete precondition of *add1*. Conversely, when *add1* returns control to f afterward, we need to find a way to transform the universal postcondition of $add1_{\text{bt}}$ into the concrete postcondition of *add1* that f expects.

When we had a similar mismatch between guarantees and expectations on trust boundaries during compilation, we introduced stubs to enforce contracts at the target level. Enforcing a contract is exactly what we need to do here, but now at the source level. We therefore reexamine the outcall stub $add1_{\text{comp}}$ (semi-transparently repeated in Figure 16) that $\vdash \mathbf{s}$ generates for *add1*, based on the contract it expects *add1* to uphold. This outcall

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

```
1  (. . .)
2  (result,n) = add1_bt(a,m);
   {univ_contr_int*_0(result)
3  * univ_contr_int*(n, 1)}_[a:a][a_1^pre:a_1][result:result][n:n]
4  guard(n != null);
5  {true * n_chunk : n ↦_int [l_1]}_[~]
6  int* a^post; int a_1^post;
7  {n_chunk : n ↦_int [l_1]}_[~][a^post:0][a_1^post:0]
8  a^post = n; a_1^post = n[0];
9  {n_chunk : n ↦_int [l_1]}_.[~][a^post:n][a_1^post:l_1]
10 guard(result == a_1^post + 1);
11 guard(a^post == a); guard(a_1^post == a_1^pre);
12 {n_chunk : n ↦_int [l_1] * result == l_1 + 1
   * n == a * l_1 == a_1}_[~]
13 {n_chunk : a ↦_int [a_1] *
   result == a_1 + 1}_[result:result]
14 return result
15 {n : a ↦_int [a_1] * result == a_1 + 1}_[result:result]
```

Fig. 19. Excerpt from the separation logic proof of the body of *add1* from the naive back-translation example.

stub contains reified checks (i.e. the guard statements on line 6 and lines 9–10) enforcing all conditions present in the postcondition of *add1*. In other words, if we can somehow back-translate the outcall stub $add1_{\text{comp}}$ and insert it between *f* and $add1_{\text{bt}}$, the back-translated guard statements should correctly add the missing concrete conditions to $add1_{\text{bt}}$'s universal postcondition. Additionally, but less crucially, this back-translated outcall stub will have to connect the arguments and preconditions of the two functions. Given that the back-translated outcall stub needs to convert $add1_{\text{bt}}$'s universal postcondition into *f*'s concrete one and vice versa for preconditions, the back-translation cannot make use of universal contract blocks, as it did for $add1_{\text{bt}}$. Stubs hence make use of an alternative, second, back-translation, which *does* retain concrete information across back-translated statements.

The back-translated outcall stub, called *add1* to match the name used by *f*, is shown in the top left corner of Figure 16. It mostly consists of straightforward back-translations of the individual statements of $add1_{\text{comp}}$, with a few caveats, mainly caused by the fact that *f* is a regular verified source function, whereas $add1_{\text{comp}}$ is a back-translated function that mimics a target function. The non-obvious aspects are the following:

- Notice how the guard statement checking the length of *n* on line 6 of $add1_{\text{comp}}$ is absent on line 9 of *add1*. This discrepancy is caused by the naive assumption, which allows us to know beforehand that *n* has length 1. This distinction between the functions is a clear hint that our current back-translation schema is not sufficiently general. It will naturally disappear in the next section, when we lift the naive assumption.
- A proof of the contract of *add1* needs to be constructed, to prove soundness of the back-translation. Contrary to the regular back-translation, no universal contract blocks are created for each back-translated statement, since we *want* to make a non-modular proof. This implies that only the RULE APPLICATION phase for each back-translated block is kept. An example of symbolic execution of the interesting part of *add1*, describing how the back-translated guard statements transform a universal contract into a concrete postcondition, is found in Figure 19.
- The function *f* is unaware of the back-translated reified resource *m*, whereas $add1_{\text{bt}}$ expects *m* as an argument. This value hence has to be declared and assigned on line 4 of *add1*, using the information about the logical resource *m* present in the precondition.
- The back-translated guard statements in *add1* guarantee equi-termination between source and target languages when $add1_{\text{bt}}$ misbehaves, since they mirror the guards in the target-level outcall stub $add1_{\text{comp}}$.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Because the example's back-translation (including source-level stubs) forms a sound separation-logic proof and closely mimics the target language, this concrete instantiation of equation (**BT-⇕**) will indeed hold, as we set out to illustrate at the start of this section.

6.2 The regular back-translation

In this section, we generalize the back-translation of *add1*, introduced in Section 6.1, by lifting the naive assumption made there. Concretely, we discard the assumption that each target-level linear capability should have size one and extend the back-translation to allow for linear capabilities of arbitrary size. In fact, the size of linear capabilities may not be statically determined, and we have to take this into account in the back-translation. For example, *add1* might be invoked by other functions than *f*, which may hand it a capability *m* with an arbitrary size. The current universal contract for *m* in Figure 16 clearly does not allow for this case.

In addition to this first generalization, we will reformulate the definition of universal contracts to use range instead of array resources. This reformulation is required for the back-translation of nested pointers, the discussion of which we defer to Section 6.4.

The back-translation of our example-context, *add1*, is updated to reflect both changes. The results are presented in Figures 20 and 21, which generalize Figures 16 and 17, respectively. We first revisit the back-translation of types and expressions, before redefining universal contracts.

Now that linear capabilities $l^{[a,b]}$ are no longer assumed to always have length 1 in the target language, an information discrepancy between source and target pointers arises. The naive back-translation of target pointers to source pointers in the InvertCapability rule made us forget their length $b - a + 1$. We add length information to back-translated pointers by introducing a form of *fat pointer* scheme. We back-translate each linear capability of type $\tau*$ to a pair $(\tau'*, \text{int})$. The first element $\tau'*$ is a pointer to contents of type τ' (the recursive back-translation of type τ). Again, $\tau'*$ simultaneously represents the pointer's address. The second int element is the externalized length of the capability. The null-pointer null is back-translated to the fat pointer (null, 0). In summary, we update the judgment $\tau \rightsquigarrow_{\text{InvCompileType}} \tau'$, by redefining its InvertCapability rule as follows:

$$\frac{\tau \rightsquigarrow_{\text{InvCompileType}} \tau'}{\tau* \rightsquigarrow_{\text{InvCompileType}} (\tau'*, \text{int})} \text{(InvertCapability)}$$

The simulation of linear behavior in the source language remains mostly unaltered. The only difference is that linear erasure should now make use of fat pointers. When, for example, back-translating the assignment $x = n[2]$ with n of target type int** from before, an erasure assignment $n[2] = (\text{null}, 0)$ has to be added in the source (remember that (null, 0) is the fat null-pointer).

The back-translation of expressions also largely remains the same, except for the length and address functions, which we created our fat pointer scheme for. The back-translation hence maps addr(*texp*) to $texp_{\text{b}}.1$ (the address is the first part of the fat pointer), length(*texp*) to $texp_{\text{b}}.2$ (the length is the second part of the fat pointer) and *null* to (null, 0) (as mentioned before). All other cases are still the identity.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

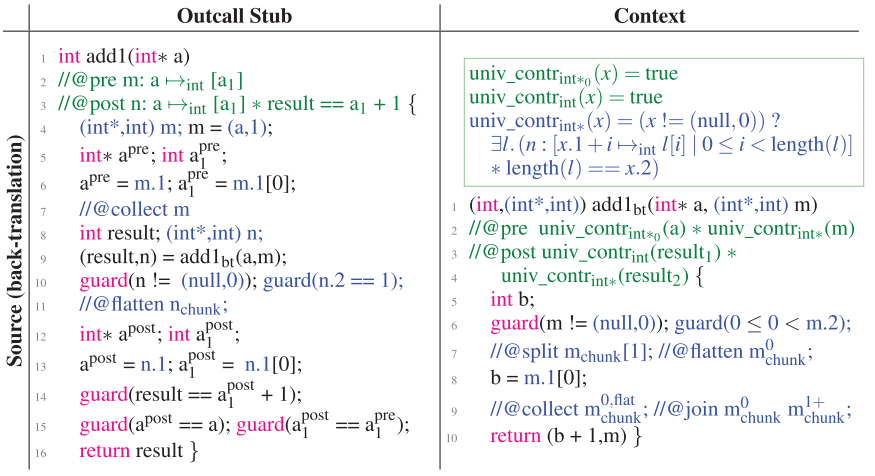


Fig. 20. Illustrative example: back-translating a context that implements *add1*. The differences with Figure 16 have been highlighted.

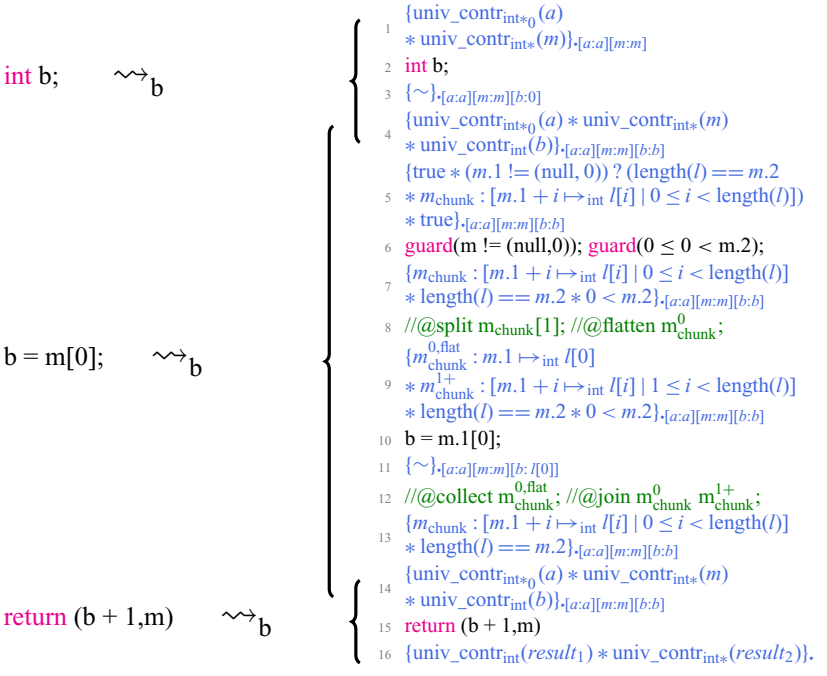


Fig. 21. Separation logic proof of the body of $add1_{bt}$ from the back-translation example.

We now examine the updated back-translation $add1_{bt}$ of *add1* in Figures 20 and 21. The differences with Figure 16 are highlighted in Figure 20.

The first thing to note is the introduction of the guard statements on line 10 of *add1* and line 6 of $add1_{bt}$. Since we discarded the naive assumption, we have to manually check whether $add1_{bt}$ returns a capability of size 1 as specified by the contract of *add1*. In this way, these guards reintroduce length information to the universal contract. With the addition of this length guard, *add1* now reflects all guard statements of $add1_{comp}$, as expected.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Second, fat pointers cause minor differences. Lines 4, 6, 8, 10 and 13 of *add1* and lines 1, 6 and 8 of $add1_{\text{bt}}$ have been adjusted to accommodate the fat pointer scheme.

Finally, as mentioned, the pointer case in the universal contracts on lines 2–3 of $add1_{\text{bt}}$ has been adjusted to allow non-statically sized capabilities and to use range resources. The new definition looks as follows:

Definition ($\text{univ\_contr}_{\tau_t}(exp)$-bis)**.**

$$\text{univ\_contr}_{\tau_t}(exp) = \text{true } \textit{if } \tau_t = \text{int } \textit{or } \tau_t = \tau_s{*_0}$$

$$\text{univ\_contr}_{(\tau_1,\ldots,\tau_k)}(exp) = \text{univ\_contr}_{\tau_1}(exp.1) * \ldots * \text{univ\_contr}_{\tau_k}(exp.k)$$

$$\text{univ\_contr}_{\tau_t*}(exp) =$$

$$exp\ != (null, 0)\ ?\ \exists l.\ (n : [exp.1 + i \mapsto_{\tau_s} l[i] \mid 0 \leq i < \text{length}(l)] * \text{length}(l) == exp.2)$$

given that $\tau_t \rightsquigarrow_{\text{InvCompileType}} \tau_s$ and n fresh

The case for target capabilities τ_t* now states that a linear capability is either the fat null-pointer (null, 0), or that we have a range resource that allows us to access each element $l[i]$ of the capability (without knowing anything about the value of $l[i]$, hence the existential quantification over l), where $exp.1$ is the capability's address and $exp.2$ is its length. The fat-pointer scheme appears here because the universal contracts are used to describe the permissions associated with back-translated τ_t*-typed variables id_p, and these back-translated variables are fat pointers. Again, the CONSEQ rule will allow us to introduce or eliminate the universal quantification.

Note that if we solely wanted to support non-statically sized capabilities, we might as well have defined the τ_t*-case as follows:

$$exp\ != (null, 0)\ ?\ \exists l.\ (n : exp.1 \mapsto_{\tau_s} l * \text{length}(l) == exp.2)$$

However, this formulation would not be sufficiently general to handle the back-translation of nested pointers in Section 6.4, since it does not allow for nested universal contracts. Specifically, when back-translating nested linear capabilities, each element of the back-translated capability is again a pointer whose permissions are described by a universal contract. This will require a recursive universal contract call inside the range resource in the pointer case above.

The use of range resources in universal contracts necessitates some changes in the different back-translation phases of Figure 18. Since the separation logic rules that are applied in the RULE APPLICATION phase require array resources in their pre- and postconditions, we need to convert the range resources from the universal contract to array resources before applying the rule, and back afterward. This is what happens on lines 7 and 9 of $add1_{\text{bt}}$ in Figure 20; the resource we have for the fat source pointer m is a range expression, but we need an array resource to apply ARRAYLKUP on line 8. Afterward, we need a range resource again to satisfy the universal contract. Lines 7–13 in Figure 21 perform this conversion between array and range resources in both directions. A length-1 range resource m^0_{chunk} (not explicitly shown) is split from m_{chunk} and flattened to the array resource $m^{0,\text{flat}}_{\text{chunk}}$ on lines 7–9. The rule ARRAYLKUP can now be applied on lines 9–11. Afterward, the resource is recollected to reobtain the range resource m^0_{chunk} and rejoined to the rest of the range resource on lines 11–13. In other words, the CONCRETIZATION phase for each

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Type of Misbehavior	Example Body
Functional	int b; b = m[0]; return (b − 1,m)
Out of bounds	int b; b = m[1]; return (b + 1,m)
Breaching linearity: • Storing	int b; b = m[0]; int∗ n; n = m; return (b + 1,m)}

• Duplicating	int b;
	b = m[0];
	(int∗,int∗) n; n = (m,m);
	m = n.1;
	return (b + 1,m)

Fig. 22. Examples of different classes of misbehaving implementations of *add1*.

back-translated statement uses split and flatten statements to convert range resources to array resources, and the UNIVERSALIZATION phase will use collect and join statement to do the inverse.

A similar conversion between array and range resources is now required in the back-translated stub *add1*, since the function *f* uses array resources, whereas $add1_{\text{comp}}$ uses range resources. Lines 7 and 11 of *add1* switch between the two representations, similarly to lines 7–13 of Figure 21. Line 11 of *add1* can simply be inserted (together with the new guard on line 10) into the proof of Figure 19 to make it go through with the new definition of universal contracts.

In the general case, lines 7 and 11 of *add1* will not suffice to convert between a range and array representation of resources. The reason for this is that the universal contract consists of a range resource containing length-1 array resources. If, for example, the resource *m* in the precondition of *add1* was to have a length greater than 1, split statements would be required before the collect on line 7 of *add1*. The technical report defines procedures to perform this conversion in the general case.

6.3 Back-translating misbehaving contexts

Having defined a more general back-translation, this section briefly investigates how the back-translation handles alternative, misbehaving implementations of *add1*. Notice that swapping out the implementation of *add1* does not affect the back-translated stub *add1*. These misbehaving target contexts will always get stuck, either due to a failing guard statement or due to the operational semantics. Since these alternative implementations of *add1* and their back-translations have to satisfy a specific equi-terminating instance of **BT-⇳**, their back-translation should get stuck as well. The different types of misbehavior are listed in Figure 22 and illustrated by means of a possible implementation of the body of *add1*. They can be subdivided into three main categories: functional misbehavior, out of bounds accesses and breaking the restrictions of linearity. We now discuss these in order.

First, *add1 functionally misbehaves* when it does not satisfy one or more non-spatial conditions that *f* expects *add1* to uphold in its postcondition. This will cause the failure of

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

one or more guard statements in both *add1* and $add1_{comp}$, ensuring equi-termination. In the concrete example from Figure 22, the variable *b* is decremented instead of incremented, causing the guard statement on line 14 of *add1* in Figure 20 and the corresponding guard on line 9 of $add1_{comp}$ in Figure 9 to fail. Similarly, the example could have set the value of *m*[0] to 0 before returning, reduced the bounds of *m* or tried to return a different linear capability altogether, each time making a different pair of guard statements fail.

Second, *out of bounds accesses* happen when *add1* reads from or writes to linear capabilities outside their intended bounds. The example in Figure 22 contains a read from index 1 of *m*, causing ARRAYMUT from Figure 4 to get stuck when f_{comp} provides a value for *m* with a length of 1. On the source level, the second guard statement on line 6 of $add1_{bt}$ in Figure 20 (now enforcing $0 \leq 1 < m.2$) would ensure equi-termination.

Last, *add1* can try to *breach linearity guarantees* and keep a copy of the linear pointer *m*, either by trying to *store* it for later use, or by using multiple copies of *m* in one statement, thereby trying to *duplicate m*.

The storing example in Figure 22 stores the value of *m* in *n* for later use. Currently, *n* in this example is modeled as local state for simplicity reasons. However, it would be more useful for a malicious context to store linear capabilities in context-global state. Our target language model does not include such global state, but it could be simulated by passing global state around using an additional parameter.

The third line of the example is back-translated to (int∗,int) n; n = m; m = (null,0). The last statement emulates erasure and ensures that both the first guard statement on line 10 of *add1* and the corresponding guard in $add1_{comp}$ fail. Notice that, if the postcondition of *add1* did not require the return of the resource *n*, then the guard statements on line 10 of *add1* would not have been generated, and the storing example would not have been problematic.

The duplication example in Figure 22 duplicates the value of *m* in *n*, before perhaps storing it for later use or causing aliasing in the return value. The third line of the example is back-translated to ((int∗,int),(int∗,int)) n; guard(false); n = (m,m); m = (null,0). Remember that a guard(false) statement is inserted when back-translating code that attempts to duplicate a linear capability. The guard statement ensures equi-termination, emulating the target-level semantics getting stuck.

6.4 Back-translating nested pointers

The back-translation in Section 6.2 did not yet support back-translating nested pointer types. This section will fill this gap, by defining universal contracts for back-translated, nested pointers and investigating how they are used in statements. Notice that back-translated nested pointers can never appear in the universal contracts of back-translated boundary functions, since Section 4.4 required boundary function contracts to solely contain array resources, which cannot result in nested pointers after compilation. Although alleviating this restriction is future work, this currently means that the conversion between range and array resources in back-translated stubs does not need to be generalized in this section.

We now investigate what the universal contract for a back-translated pointer of target type int∗∗ looks like. The general case can easily be derived from this, but contains some

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

additional uninteresting clutter, to do with resource names n. The τ_t*-case of the universal contracts from Section 6.2 contains a range resource

$$n : [exp.1 + i \mapsto_{\tau_s} l[i] \mid 0 \leq i < \text{length}(l)]$$

that in turn contains array resources to access each individual element of the pointer represented by *exp*. Unfortunately, this does not represent the permissions carried by nested pointers. The reason is that $l[i]$ itself is not necessarily a permissionless value with universal contract true, but rather a value of a type that carries its own permissions, again described by a universal contract. In our case where $\tau_t* = \text{int}**$ and $\text{int}** \rightsquigarrow_{\text{InvCompileType}} ((\text{int}*, \text{int})*, \text{int})$, the logical list element $l[i]$ represents a value of type $(\text{int}*, \text{int})$, that we should again define an *inner* universal contract for, using a range resource. The only difference between the outermost and inner universal contracts was highlighted in Section 3.1 already; the nested universal contracts do not require chunk names, as range resources are reified as a whole during compilation.

Given these observations, we can define universal contracts for int∗∗-pointers as follows:

Definition ($\text{univ\_contr}_{\text{int}**}(exp)$)**.**

$$\begin{aligned}
&\text{univ_contr_inner}_{\text{int}*}(exp) = \\
&\quad exp \mathrel{!}= (null, 0)\ ?\ \exists l.\ ([exp.1 + j \mapsto_{\text{int}} l[j] \mid 0 \leq j < \text{length}(l)] * \text{length}(l) == exp.2) \\
&\text{univ_contr}_{\text{int}**}(exp) = \\
&\quad exp \mathrel{!}= (null, 0)\ ?\ \exists l.\ (n : [exp.1 + i \mapsto_{(\text{int}*,\text{int})} l[i] * \\
&\qquad \text{univ_contr_inner}_{\text{int}*}(l[i]) \mid 0 \leq i < \text{length}(l)] * \text{length}(l) == exp.2) \\
&\quad \text{given that } n \text{ fresh}
\end{aligned}$$

The interesting aspects have been highlighted; notice the nesting of the inner contracts $\text{univ\_contr\_inner}_{\text{int}*}$ inside the outer contract $\text{univ\_contr}_{\text{int}**}$ and the single range resource name n. This nested contract structure would not have been possible using a single resource name n if universal contracts used array resources instead. This is the motivation for the reformulation in terms of range resources in Section 6.2. The above structure is easily generalized to arbitrary target types τ_t, by allowing inner contracts to contain more deeply nested inner contracts.

The back-translation of statements containing nested pointers happens very similarly to the non-nested examples we saw before, except that more intricate emulation of linearity in the source language is often required. To illustrate this, we rewrite line 2 of *add1* in Figure 16 with m now of type int∗∗, obtaining `int* b; b = m[0]`. This line gets back-translated to the code on the right in Figure 23. Notice how small the differences with the original back-translation on the left are: only lines 1 and 4 differ because b is now a linear value itself, instead of a duplicable integer. The proof is also very similar to the one from Figure 21, except that on line 10 of this figure, we would need to consume the inner universal contract for $m.1$ at index 0 to derive that the new value of b satisfies the universal contract of its type int∗. Since universal contracts are linear, we need the erasure of $m.1[0]$ on line 4 of Figure 23 to re-establish the universal contract of $m.1$ at index 0 afterward.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Excerpt from Figure 20

```
5  int b;
6  guard(m != (null,0)); guard(0 ≤ 0 < m.2);
7  //@split m_chunk[1]; //@flatten m^0_chunk;
8  b = m.1[0];
9      //@collect m^{0,flat}_chunk; //@join m^0_chunk m^{1+}_chunk;
```

Back-translation of int* b; b = m[0]

```
1  (int*,int) b;
2  guard(m != (null,0)); guard(0 ≤ 0 < m.2)
3  //@split m_chunk[1]; //@flatten m^0_chunk;
4  b = m.1[0]; m.1[0] = null;
5  //@collect m^{0,flat}_chunk; //@join m^0_chunk m^{1+}_chunk;
```

Fig. 23. Back-translation of line 2 of *add1* with and without nested pointers.

7 Simulation relations

This section takes the black-box relations for correctness and security, R and S, from Section 5.3 and decomposes both in Sections 7.1 and 7.2, respectively. Formulated differently, this section provides more details on how to prove the **ADEQUACY** proof step in Figures 13 and 14.

7.1 Decomposing R

We decompose R first, since its decomposition is easier. The reason is that the proof of **COMP-⇳** only involves compiled components, whereas **BT-⇳** requires simulating both compiled and back-translated components.

To illustrate this section, we need a source program and its compilation. We assume the source component from Figure 9 as our verified source component $\vdash \mathbf{s}$ and an arbitrary verifiable source context $(\mathfrak{C}_s, id)$ that implements and exports *add1*, with the main function *main* from Figure 15 as our main function *id*. Again, we ignore the actual implementation of *main* and its compilation *main* in, respectively, $\vdash \mathbf{s}$ and t to avoid uninteresting clutter. Including compiled components, we then have the following (we repeat the same notational abuse from Section 6 where the names of functions can be used to represent their entire implementation):

$$\mathbf{s} = f \;//@\text{import } add1$$
$$(\mathfrak{C}_s, id) = ((add1 \;//@\text{export } add1\;),\; main)$$
$$[\![\vdash \mathbf{s}]\!] = f_{\text{comp}}\; add1^{\text{out}}_{\text{comp}} \;//@\text{import } add1$$
$$(\mathfrak{C}_t, id) = ([\![\vdash \mathfrak{C}_s]\!], id) = ((add1\; add1_{\text{comp}} \;//@\text{export } add1\;),\; main)$$

The lower two lines follow from the COMPVERIF rule of our compiler in Figure 7. Note how we added the superscript out to the generated outcall stub $add1^{\text{out}}_{\text{comp}}$, to distinguish it from the compiled context function $add1_{\text{comp}}$. The context function *add1* is the generated incall stub.

The statement we need to prove as part of **COMPATIBILITY** is shown on the left in Figure 24. The right side of this figure shows how this statement, once proven, implies equi-termination of our source and target programs, or in other words, proves **ADEQUACY**. We now focus on explaining this right part.

When simulating our compiled code, a distinction has to be made between execution within the individual source components $\mathbf{s}$ and $\mathfrak{C}_s$, and execution when an outcall stub is called or is being returned from, that is, when a transition between $\mathbf{s}$ and $\mathfrak{C}_s$ (or back) occurs. Notice that an outcall stub is always called first, and then execution transitions to

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

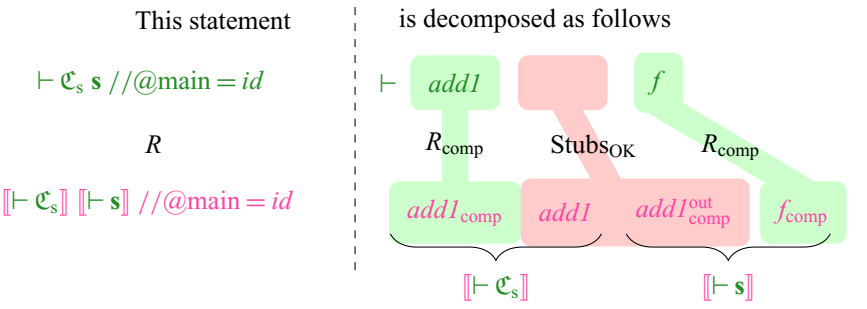


Fig. 24. Visual representation that illustrates the decomposition of R (inspired by the schemata of Devriese *et al.*, 2016).

the incall stub of the component that is being called. When returning, the order is inverted. In general, we will refer to this sequence of either two calls or two returns as a *component switch*.

The reason that R (and S) is not a simulation relation itself (as we mentioned in Section 5.3) is that R does not consider states to be related during in- or outcalls, but rather consists of two separate parts: an actual simulation relation R_{comp} and a *connecting lemma* Stubs_{OK} to link up different instances of R_{comp} across component switches.

First, R consists of a simulation relation R_{comp} that relates source code to its compilation and models how the compiler produces equi-terminating code in the target language. The relation R_{comp} is solely used to reason within a single domain of trust, that is, within a single source and target component (hence the comp subscript) during execution. The simulation halts right before a component switch occurs. More concretely, the technical report proves that R_{comp} satisfies the following definition of a *forward*, *strong* simulation relation (inspired by the definition of a *Simulation relation with multiple matching steps* in Chlipala, 2017):

Definition (source-to-target forward simulation relation)**.** *Given a relation R_{comp} relating source states $\mathbf{st} = \langle \overline{\mathbf{s}}, \mathbf{h} \rangle \mid \overline{\vdash \mathbf{c}}$ to target states $\text{st} = \langle \overline{\text{s}}, \text{h} \rangle \mid \overline{\text{c}}$. If the following 2 properties hold, then R_{comp} is a source-to-target forward simulation relation:*

1. *The first property is used to guarantee equi-termination in the proof of* **ADEQUACY**, *when we know that the source program terminates:*

$$\forall P, Q, \mathbf{s}, \mathbf{h}, \text{s}, \text{h}, \text{c}.\ (\langle \mathbf{s}, \mathbf{h} \rangle \mid \{P\}\ \text{return}\ \{Q\})\ R_{\text{comp}}\ (\langle \text{s}, \text{h} \rangle \mid \text{c}) \Rightarrow \text{c} = \text{return}$$

   It states that a terminated source statement (i.e. a single return statement) must correspond to a terminated target statement.
2. *The second property is the inductive part of the simulation relation:*

$$\forall\ \mathbf{st}, \text{st}, \mathbf{st}'.\ \mathbf{st}\ R_{\text{comp}}\ \text{st} \wedge \mathbf{st} \hookrightarrow \mathbf{st}' \Rightarrow \exists\ \text{st}'.\ \text{st} \hookrightarrow^{+} \text{st}' \wedge \mathbf{st}'\ R_{\text{comp}}\ \text{st}'$$

   Note that this condition requires the target operational semantics $\hookrightarrow$ to perform at least one step, denoted $\hookrightarrow^{+}$. Remember that the source level makes use of the lifted operational semantics.

Second, R also requires proving a connecting lemma Stubs_{OK}. This lemma essentially states that if the source and target states are related by R_{comp} before a component

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

switch, they will still be related after the switch, and R_{comp} can hence continue simulating. Additionally, Stubs_{OK} does not allow target code to get stuck while executing code in an in- or outcall stub, since source code does not have any stub code to execute, and this would otherwise break equi-termination. Fortunately, R only relates correctly behaving programs that live up to their contracts, and the guards will never fail. The situation will be different in the security direction, where we consider potentially misbehaving contexts.

Skipping the execution of the main functions *main* and *main* that simply perform setup and pass control to f and f_{comp}, the right side of Figure 24 now ensures equi-termination as follows; execution starts off on the far right in the function f (f_{comp} in the target), and R_{comp} simulates (using the second property in the above definition) until f_{comp} is about to perform an outcall to $add1^{\text{out}}_{\text{comp}}$. At this point, the Stubs_{OK} lemma is applied, guaranteeing us that we can bypass the stubs $add1^{\text{out}}_{\text{comp}}$ and $add1$ in the middle and resume simulation under R_{comp} at the left side, in the function $add1$ ($add1_{\text{comp}}$ in the target). Once $add1$ is about to return, the Stubs_{OK} lemma is again applied to make the inverse transition back to f. When execution terminates in *main* at the source level, the first property in the above definition is applied, thereby proving that the target program has also terminated (in *main* in this case).

7.2 Decomposing S

Having expanded our toolbox in the previous section, we now study the decomposition of the relation S. As stated before, this decomposition is slightly more involved, since both compiled and back-translated components are present in the statement of **BT-⇳**.

To illustrate this section, we reuse the example described in the introduction of Section 6 (including the *main* function defined there, the fact that we keep the main functions implicit, and the abuse of notation for functions). To reiterate, we had the following:

$$\begin{aligned}
\mathbf{s} &= (f \ \text{//@import } add1\ ) \\
(\mathfrak{C}_{\text{t}}, id) &= ((add1 \ \text{//@export } add1\ ),\ main) \\
[\![\vdash \mathbf{s}]\!] &= f_{\text{comp}}\ add1_{\text{comp}}\ \text{//@import } add1 \\
(\mathfrak{C}_{\text{s}}, id) &= (\langle\!\langle \mathfrak{C}_{\text{t}} \rangle\!\rangle, id) = ((add1\ add1_{\text{bt}}\ \text{//@export } add1\ ),\ main)
\end{aligned}$$

Figure 25 is similar to Figure 24, showing the **COMPATIBILITY** statement on the left and how it implies adequacy on the right. We now focus on explaining this right part.

In this case, we need to make a distinction between not two, but four different modes of execution. First off, there are two different regular, intra-component modes of execution: either execution is happening within the source component $\mathbf{s}$ and its compilation, or within the component $\mathfrak{C}_{\text{t}}$ and its back-translation. Furthermore, two different transitions can now be made: either $\mathbf{s}$ performs an outcall to the context, or the context performs an incall to $\mathbf{s}$. Since f is not an exported function, this second scenario cannot occur in our simple example. Unlike in Section 7.1, incalls and outcalls do not occur in pairs, since the target context does not result from compilation and hence does not generate its own stubs. Another distinction with Section 7.1 is that in- and outcalls now execute code in both the source and target languages, since stubs such as $add1_{\text{comp}}$ are back-translated into the source context.

Again, S as introduced in Section 5.3 was not really a simulation relation, but rather consisted of the four aforementioned parts: two simulation relations R_{comp} and S_{comp} (with R_{comp} as defined before) and two connecting lemmas Incall and Outcall.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

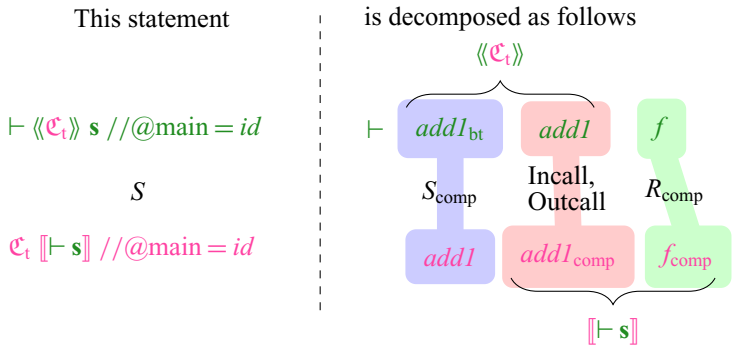


Fig. 25. Visual representation that illustrates the decomposition of S (inspired by the schemata of Devriese *et al.*, 2016).

First, S consists of two different simulation relations R_{comp} and S_{comp}, both used to reason within a single domain of trust, that is, halting before a component switch. The relation R_{comp}, discussed in the previous section, is used to reason about code and its compilation. On the other hand, the relation S_{comp} relates target code to its back-translation and models how the back-translation produces equi-terminating code in the source language.

Since S_{comp} performs a target-to-source simulation of back-translated code in terms of target code, whereas R_{comp} performed source-to-target simulation of compiled code in terms of the original source code, the technical report defines a second, different notion of *forward*, *strong* simulation relation. The relation S_{comp} is then proven to satisfy this notion. The second version of simulation we use is defined as follows (again inspired by the definition of a *Simulation relation with multiple matching steps* in Chlipala, 2017):

Definition (target-to-source forward simulation relation)**.** *Given a relation S_{comp} relating source states* $\mathbf{st} = \langle \overline{\mathbf{s}}, \mathbf{h} \rangle \mid \overline{\vdash \mathbf{c}}$ *to target states* $\mathrm{st} = \langle \overline{\mathrm{s}}, \mathrm{h} \rangle \mid \overline{\mathrm{c}}$ *. If the following 2 properties hold, then S_{comp} is a target-to-source forward simulation relation:*

1. *The first property is used to guarantee equi-termination in the proof of* **ADEQUACY***, when we know that the target program terminates:*

$$\forall\, \mathbf{s}, \mathbf{h}, \mathbf{c}, \mathrm{s}, \mathrm{h}.\ (\langle \mathbf{s}, \mathbf{h} \rangle \mid \mathbf{c})\ S_{comp}\ (\langle \mathrm{s}, \mathrm{h} \rangle \mid \mathrm{return}) \Rightarrow \exists\, P, Q.\ \mathbf{c} = \{P\}\ \mathrm{return}\ \{Q\}$$

   It states that a terminated target statement (i.e. a single return statement) must correspond to some valid proof of a terminated source statement.

2. *The second property is the inductive part of the simulation relation:*

$$\forall\, \mathbf{st}, \mathrm{st}, \mathbf{st}'.\ \mathbf{st}\ S_{comp}\ \mathrm{st} \wedge \mathrm{st} \hookrightarrow \mathrm{st}' \Rightarrow \exists\, \mathbf{st}'.\ \mathbf{st} \hookrightarrow^{+} \mathbf{st}' \wedge \mathbf{st}'\ S_{comp}\ \mathrm{st}'$$

   Note that this condition again requires the target operational semantics $\hookrightarrow$ to perform at least one step, denoted $\hookrightarrow^{+}$. Remember that the source level makes use of the lifted operational semantics.

Second, S also requires proving two connecting lemmas Incall and Outcall, used for incalls and outcalls, respectively. The Outcall lemma states that if R_{comp} holds in the verified component **s**, we can perform an outcall to the context, and after executing the outcall

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

stubs in both source and target languages, S_{comp} will hold in the context. Similarly, when returning from the outcall, R_{comp} will still hold in **s**. A crucial difference with last section is that execution is now allowed to get stuck in the outcall stubs, as long as it gets stuck in both source and target languages, preserving equi-termination. The Incall lemma makes similar claims, but for incalls.

The right side of Figure 25 illustrates the decomposition of S. Interestingly, note how $add1_{\text{comp}}$ is part of the compiled, verified component in the target language, whereas its back-translation *add1* is part of the source context. Equi-termination is proved as follows (again ignoring the execution of the main functions *main* and *main* in our example); execution starts off on the far right in the function f (f_{comp} in the target), and R_{comp} simulates (similarly to what we saw in the previous subsection) until f and f_{comp} are about to perform an outcall to the pair of outcall stubs *add1* and $add1_{\text{comp}}$.

In order to be able to apply the Outcall lemma here (and similarly for Incall), both the source and target code must reach their respective outcalls to *add1* and $add1_{\text{comp}}$ simultaneously during simulation (and similarly, return from them simultaneously). This follows easily from the definitions of R_{comp} and S_{comp} in the technical report. Consequently, the Outcall lemma can be applied, guaranteeing that either execution gets stuck in both stubs, or we can bypass the stubs in the middle and resume simulation under S_{comp} at the left side, in the function $add1_{\text{bt}}$ (*add1* in the target). Once $add1_{\text{bt}}$ is about to return, the Outcall lemma is again applied to make the inverse transition back to f.

Proof Conclusion. *In the previous Sections 5, 6 and 7, we discussed the main intuitions behind the full abstraction proof of our compiler, including the back-translation, in an example-driven way. These sections should provide the reader with sufficient anchoring points to understand the full proof in the aforementioned technical report (Van Strydonck et al., 2020), in case they are interested in the more formal and detailed approach.*

8 Discussion and future work

This section first provides more detail on two challenges in making our compiler more broadly applicable and subsequently discusses the benefits of semantically deriving our separation logic rules, instead of stating them syntactically.

8.1 Gradual verification

Function signatures are modified by our compiler, as apparent from its definition in Section 4: additional parameters and/or return values that represent the memory resources that are transferred are added. Additional effort is hence still required by third-party developers to produce code that follows our target-level calling convention. Two scenarios are possible.

First of all, a developer could write verified code themselves and compile it using our compiler, gaining the same secure compilation guarantees that our compiled code does. Although this could be realistic in some settings, it goes against our original goal of allowing interaction of our compiled code with arbitrary, non-verified attacker code.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Second, the developer could write unverified code in the target language. This code must then be written to call and be called with the modified function signatures. This might be realistic for applications like the video player with codec plug-in described in the introduction. However, we would also like to support a form of gradual verification, where we can take a large, unverified codebase, verify the critical parts and securely combine them with the rest. This type of use case is currently only supported when boundary functions solely use integer arguments and return values and do not receive or return memory resources, since the declaration of such functions is not altered during compilation. Even with this strong restriction, our strong security results might still be useful in some scenarios.

We plan to explore two ideas for extending our approach to large, partially verified codebases: either based on the use of an automatic verifier on the unverified code, like Smallfoot (Berdine *et al.*, 2005), Space Invader (Distefano *et al.*, 2006; Berdine *et al.*, 2007), Infer (Calcagno *et al.*, 2015) or SLAyer (Berdine *et al.*, 2011), or on a kind of universal contract for unverified code in terms of a pure predicate similar to the lowval predicate of Swasey *et al.* (2017). Such an approach could be valuable in practice, as many large code bases contain small, isolated components whose security is of high value and for which the verification effort might be realistic and cost-effective.

8.2 Extending the source language

A second direction we want to expand our work in is to extend the compiler itself. As mentioned in the introduction, this paper contains but the first steps toward a practically applicable secure compilation scheme. Notably, the source language only consisted of simple resources in the separation logic, had a simple type system and featured restrictions on the form of boundary contracts. We now discuss some ideas for extensions in these three directions in order. We do believe all suggested extensions to be within reach.

Resources. In this paper, we support only two kinds of spatial predicates, describing array and range resources. We believe our approach can be extended to a more general form of predicates, by relying on a notion of capability *sealing*. Support for such predicates would also allow us to formalize memory de-allocation, that is, a free-statement. The difficult part in supporting free is that the authority to deallocate a block of memory needs to be represented separately from the authority to access the memory (i.e. our array points-to predicate). This is because the latter can be subdivided, but the former should not be, since most memory allocators rely on the entire block being deallocated together. To accommodate this, separation logics like VeriFast represent the authority to deallocate memory with a special *malloc resource* abstract predicate. We could do the same and compile this resource in the same way as the discussed general predicates.

Type system. As mentioned in the introduction, the most obvious feature missing from the source language is support for recursive data types, for example, in the form of C-like structs. We believe the type system will scale in parallel with the introduction of resources to represent more complex permissions, for example, *struct* types would be introduced in parallel with the general predicate resources discussed above. The general resource reification principles demonstrated in the current submission would remain the same.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Boundary contract restrictions. Loosening boundary contract restrictions corresponds to loosening the constraints on calls to untrusted code. It seems possible to allow non-fixed-length array resources and range resources to appear in boundary contracts, by reifying (nested) foreach loops in our stubs, given some proof changes. Reification of foreach loops could also be used to allow quantifiers over finite domains in boundary contracts (although efficiency remains an open question here).

8.3 Semantic separation logic rules

The separation logic rules presented in Section 4 are *syntactic* in nature; they are presented as axioms in the logic, without any formal justification. In other words, they are assumed to be part of the *trusted computing base* of the source language. To decrease the trusted computing base and make our approach more foundational, it would be worthwhile to lift these rules out of the trusted computing base. There are two ways to do so:

- Perform a proof of *adequacy* for our current rules, proving that the syntactic rules are adequate with respect to the operational semantics.
- Derive the rules *semantically* from the operational semantics, on top of some appropriate program logic. Preferably, this program logic would contain a built-in proof of adequacy for any Hoare triples derived this way. For example, the Iris program logic framework (Jung *et al.*, 2016, 2018; Krebbers *et al.*, 2017) (and corresponding proof assistant, implemented in Coq, in case we mechanize our results) would likely be a good fit to perform these derivations in, as it has proofs of adequacy for its weakest precondition judgment.

The advantage of deriving rules semantically over our current approach seems to be twofold; it would be possible to derive a back-translation that does not have to be syntactic in nature, and the CONCRETIZATION phase in Figure 18 is likely not required. We briefly discuss both advantages in the following two paragraphs.

Because of the syntactic nature of our separation logic rules, it is currently impossible to "look under the hood" and derive what we call a *semantic back-translation*. By semantic back-translation, we mean a back-translation where not each individual back-translated statement is proven to preserve universal contracts by a derived separation logic rule, but rather, it suffices to prove that the overall back-translated code respects universal contracts (similar to the proofs of semantic type safety in RustBelt Jung *et al.*, 2018 or the semantic proofs of wrapper contracts that Sammler *et al.*, 2020 use). In this setting, we would still have some notion of "universal contract", but now defined in terms of some underlying base logic. It is not unlikely, however, that it would still be easiest to derive semantic verification rules for individual code blocks based on universal contracts, as we currently do, rather than try to construct an end-to-end proof of a function's universal contract directly. This would need to be investigated further.

As for the necessity of the CONCRETIZATION phase in Figure 18; this too has to do with the fact that our separation logic rules are derived syntactically. What we illustrate in Figure 18 is essentially a syntactic derivation of a proof rule for the back-translation of each type of target statement that operates on the shapes of resources (i.e. universal contracts), instead of using some concrete pre- and postconditions like the rules we presented

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

in Section 4. This new, universal rule is derived by proving that each back-translated block can be proven to respect a contract that consists of universal contracts of local variables. If our rules were to be derived semantically, it would be possible to derive these rules in a more direct way, without going through the concrete syntactic-style rules. It seems likely that we would be able to use information from the operational semantics themselves, instead of using the explicit "guard" statements in the CONCRETIZATION PHASE, to derive rules that operate on universal contracts. This would reduce the CONCRETIZATION phase to only inserting a "guard(false)" statement to simulate linearity where necessary.

9 Related work

Our work builds on three research lines with a long and rich history: capability machines, separation logic and full abstraction. It is not feasible to give complete surveys of these three research lines here, so we just provide some pointers to key papers. For an excellent introduction to separation logic and references, we refer to O'Hearn (2012).

Capability machines have been studied for decades. Levy (1984) provides a good survey of early systems. With the increased need for security and fine-grained protection, there is a renewed interest in these machines, or in generalizations where the hardware can track even more metadata. Two influential recent systems are the CHERI system developed in Cambridge (Watson *et al.*, 2015; Chisnall *et al.*, 2015) and the SAFE machine developed within the CRASH/SAFE project (Knight, Jr. *et al.*, 2012; de Amorim *et al.*, 2015, 2016). Linear capabilities have already been implemented in the latter. Skorstengaard *et al.* (2019) have used them in a secure calling convention `StkTokens`, and an early design for their implementation in CHERI is in the latest CHERI ISA Spec (Watson *et al.*, 2020).

To formalize secure compilation, we use the property of fully abstract compilation (Abadi, 1999), like many previous results (e.g., Abadi & Plotkin, 2012; Fournet *et al.*, 2013; Patrignani *et al.*, 2015; New *et al.*, 2016; Devriese *et al.*, 2016; Skorstengaard *et al.*, 2019). We refer to Patrignani *et al.* (2019) for an overview of the field. Recent research has investigated other formal characterizations of secure compilation: robust safety preservation (Swasey *et al.*, 2017; Patrignani & Garg, 2018), trace-preserving compilation (Patrignani & Garg, 2017) and robust hyperproperty preservation (Garg *et al.*, 2017). Although we only prove fully abstract compilation, it is important to understand that most of our proof consists of the construction of the back-translation and its properties, and those parts could be immediately reused to prove many of the alternative properties.

The fact that our back-translation depends just on the context, not on the compiled program, suggests that our compiler actually also satisfies the property that Garg *et al.* (2017) call Relational Hyperproperty Preservation. Technically, our back-translation and its use of universal contracts are reminiscent of the use of universal types and universal embeddings in previous work (New *et al.*, 2016; Devriese *et al.*, 2016).

Our work is also related to the body of work on contract enforcement, where the enforcement of higher-order contracts and the assignment of blame on contract violations have received significant attention. A recent Functional Pearl (Dimoulas *et al.*, 2016) provides an in-depth discussion of this line of work. Bader *et al.* (2018) recently demonstrated how dynamic checking of Hoare logic contracts can be obtained using the general AGT framework for gradual typing (Garcia *et al.*, 2016).

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Directly related to our work are other approaches to dynamic checking of separation logic. The main challenge for such dynamic techniques is the enforcement of framing. Nguyen *et al.* (2008) use a heap coloring technique and run-time checks at every method invocation and field access in unverified code to check framing. The performance overhead of this approach is substantial, and it is limited to safe languages such as Java. Agten *et al.* (2015) were the first to propose a contract checking approach for C, but, as we discussed in the Introduction, their approach is not fully abstract, it only guarantees integrity: safety properties expressed in separation logic assertions within a verified module are guaranteed to hold at run time in the presence of an unverified context, but confidentiality properties are lost. Building further on Agten *et al.* (2015)'s work, van Ginkel *et al.* (2017) developed a separation-logic-based specification language for Intel SGX enclaves, that allows the automatic generation of contract checking functions at the enclave's trust boundaries.

Last, the notion of universal contracts introduced here has implicitly been used by other papers to describe the most general constraints that arbitrary adversarial code satisfies. More concretely, Skorstengaard *et al.* (2018, 2019) encode the guarantees obtained when executing adversarial assembly code in the fundamental theorem of their logical relations. The semantic type systems defined by Jung *et al.* (2018) and Sammler *et al.* (2020) are similarly used to specify the behavior of arbitrary untrusted code. The lowval predicate, used to describe safely shareable values, defined by Swasey *et al.* (2017), again serves a similar purpose. More generally, the guarantees obtained when syntactically restricting adversaries can be seen as an instance of parametricity. Note that parametricity should be interpreted broadly here to refer to the use of logical relations to describe semantic properties that follow from syntactic restrictions in the language. This includes not just System F's parametric polymorphism (Reynolds, 1983) but also many other semantic properties like sequentiality in PCF (Sieber, 1992), capability safety in object capability languages (Swasey *et al.*, 2017; Skorstengaard *et al.*, 2018) or purity in dependently typed languages with effects (Pédrot *et al.*, 2019).

10 Conclusion

We have explored a fundamentally new approach for the dynamic checking of separation logic contracts. Our approach relies on hardware support for linear capabilities, a form of unforgeable and non-copyable memory pointers. A proof-directed compiler represents separation logic memory resources as linear capabilities and relies on the information in the proof to compile source code pointer dereferences to dereferences of the correct linear capability. We formalized and proved the correctness of our approach by showing that our compiler from verified source code to unverified target code is fully abstract.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, by the Research Foundation - Flanders (FWO) under grant number G0G0519N and by the Air Force Office of Scientific Research under award number FA9550-21-1-0054. Thomas Van Strydonck holds a PhD Fellowship of the Research Foundation - Flanders (FWO).

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Conflicts of Interest

None.

Supplementary materials

For supplementary material for this article, please visit doi.org/10.1017/S0956796821000022.

References

Abadi, M. (1999) Protection in programming-language translations. In *Secure Internet Programming*. Springer-Verlag, pp. 19–34.

Abadi, M. & Plotkin, G. D. (2012) On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.* **15**(2), 8:1–8:29.

Agten, P., Jacobs, B. & Piessens, F. (2015) Sound modular verification of C code executing in an unverified context. In Symposium on Principles of Programming Languages. POPL'15. New York, NY, USA: ACM, pp. 581–594.

Bader, J., Aldrich, J. & Tanter, É. (2018) Gradual program verification. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science. Springer International Publishing.

Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T. & Yang, H. (2007) Shape analysis for composite data structures. In International Conference on Computer Aided Verification. Springer, pp. 178–192.

Berdine, J., Calcagno, C. & O'Hearn, P. W. (2005) Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 115–137.

Berdine, J., Cook, B. & Ishtiaq, S. (2011) SLAyer: Memory safety for systems-level code. In *Computer Aided Verification*. Springer, pp. 178–183.

Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J. & Rodriguez, D. (2015) Moving fast with software verification. In NASA Formal Methods Symposium. Springer, pp. 3–11.

Chisnall, D., Rothwell, C., Watson, R. N. M., Woodruff, J., Vadera, M., Moore, S. W., Roe, M., Davis, B. & Neumann, P. G. (2015) Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'15, Istanbul, Turkey, March 14–18, 2015, pp. 117–130.

Chlipala, A. (2017) Formal reasoning about programs. Available at: http://adam.chlipala.net/frap.

Costan, V. & Devadas, S. (2016) Intel SGX explained. *IACR Cryptol. Eprint Arch.* **2016**(086), 1–118.

de Amorim, A. A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B. C., Pollack, R. & Tolmach, A. (2016). A verified information-flow architecture. *J. Comput. Secur.* **24**(6), 689–734.

de Amorim, A. A., Dénès, M., Giannarakis, N., Hritcu, C., Pierce, B. C., Spector-Zabusky, A. & Tolmach, A. (2015) Micro-policies: Formally verified, tag-based security monitors. In 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17–21, 2015, pp. 813–830.

Devriese, D., Patrignani, M. & Piessens, F. (2016) Fully-abstract compilation by approximate back-translation. In Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016, pp. 164–177.

Dimoulas, C., New, M. S., Findler, R. B. & Felleisen, M. (2016) Oh lord, please don't let contracts be misunderstood (functional pearl). In International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016, pp. 117–131.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Distefano, D., O'Hearn, P. W. & Yang, H. (2006) A local shape analysis based on separation logic. In Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006, Vienna, Austria, March 25–April 2, 2006, Proceedings, Hermanns, H. & Palsberg, J. (eds). Lecture Notes in Computer Science, vol. 3920. Springer, pp. 287–302.

Fournet, C., Swamy, N., Chen, J., Dagand, P.-É., Strub, P.-Y. & Livshits, B. (2013) Fully abstract compilation to JavaScript. In Symposium on Principles of Programming Languages, POPL'13, pp. 371–384.

Garcia, R., Clark, A. M. & Tanter, É. (2016) Abstracting gradual typing. In *Principles of Programming Languages*. ACM, pp. 429–442.

Garg, D., Hritcu, C., Patrignani, M., Stronati, M. & Swasey, D. (2017) Robust hyperproperty preservation for secure compilation (extended abstract). Available at: http://arxiv.org/abs/1710.07309.

Insolvibile, G. (2003) Garbage collection in C programs. *Linux J.* **2003**(113), 7.

Jacobs, B. & Piessens, F. (2008) The VeriFast program verifier. *Cw Reports*.

Jacobs, B., Smans, J. & Piessens, F. (2010) A quick tour of the VeriFast program verifier. In *Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 6461. Berlin, Heidelberg: Springer, pp. 304–311.

Jung, R., Krebbers, R., Birkedal, L. & Dreyer, D. (2016) Higher-order ghost state. In International Conference on Functional Programming. ICFP 2016. New York, NY, USA: Association for Computing Machinery, pp. 256–269.

Jung, R., Krebbers, R., Jourdan, J.-H., Bizjak, A., Birkedal, L. & Dreyer, D. (2018) Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* **28**, e20.

Jung, R., Jourdan, J.-H., Krebbers, R. & Dreyer, D. (2018) RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* **2**(POPL), 66:1–66:34.

Knight, Jr., T. F., DeHon, A., Sutherland, A., Dhawan, U., Kwon, A. & Ray, S. (2012) SAFE ISA (version 3.0 with interrupts per thread). Available at: http://ic.ese.upenn.edu/distributions/safe_processor/.

Krebbers, R., Jung, R., Bizjak, A., Jourdan, J.-H., Dreyer, D. & Birkedal, L. (2017). The essence of higher-order concurrent separation logic. In *Programming Languages and Systems*, Yang, H. (ed). Berlin, Heidelberg: Springer, pp. 696–723.

Leroy, X. (2006) Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'06. New York, NY, USA: Association for Computing Machinery, pp. 42–54.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M. & Ferdinand, C. (2016) CompCert - a formally verified optimizing compiler. In ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. SEE, Toulouse, France.

Levy, H. M. (1984) *Capability-Based Computer Systems*. Digital Press.

New, M. S., Bowman, W. J. & Ahmed, A. (2016) Fully abstract compilation via universal embedding. In International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016, pp. 103–116.

Nguyen, H. H., Kuncak, V. & Chin, W.-N. (2008) Runtime checking for separation logic. In Verification, Model Checking, and Abstract Interpretation, 9th International Conference, pp. 203–217.

Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I. & Piessens, F. (2013) Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In USENIX Security Symposium, pp. 479–494.

O'Hearn, P. W. (2012) A primer on separation logic (and automatic program verification and analysis). In *Software Safety and Security - Tools for Analysis and Verification*, pp. 286–318.

Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D. & Piessens, F. (2015) Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.* **37**(2), 6:1–6:50.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press

Patrignani, M., Ahmed, A. & Clarke, D. (2019) Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.* **51**(6), 125:1–125:36.

Patrignani, M. & Garg, D. (2017) Secure compilation and hyperproperty preservation. In Computer Security Foundations Symposium. IEEE, pp. 392–404.

Patrignani, M. & Garg, D. (2018) Robustly safe compilation or, efficient, provably secure compilation. Corr, abs/1804.00489.

Pédrot, P.-M., Tabareau, N., Fehrmann, H. J. & Tanter, É. (2019) A reasonably exceptional type theory. *Proc. ACM Program. Lang.* **3**(ICFP), 108:1–108:29.

Pierce, B. C. (2002) *Types and Programming Languages*. MIT Press. Google-Books-ID: ti6zoAC9Ph8C.

Reynolds, J. C. (1983) Types, abstraction, and parametric polymorphism. In *Information Processing*. North Holland, pp. 513–523.

Reynolds, J. C. (2002) Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*. IEEE, pp. 55–74.

Sammler, M., Garg, D., Dreyer, D. & Litak, T. (2020) The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.* **4**(POPL), 32:1–32:32.

Sieber, K. (1992) Reasoning about sequential functions via logical relations. *Appl. Categories Comput. Sci.* **177**, 258–269.

Skorstengaard, L., Devriese, D. & Birkedal, L. (2018) Reasoning about a machine with local capabilities. In *Programming Languages and Systems*, vol. 10801. Springer International Publishing, pp. 475–501.

Skorstengaard, L., Devriese, D. & Birkedal, L. (2019) StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.* **3**(POPL), 19:1–19:28.

Strackx, R., Piessens, F. & Preneel, B. (2010) Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*. Lecture Notes. Berlin, Heidelberg: Springer, pp. 344–361.

Swasey, D., Garg, D. & Dreyer, D. (2017) Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* **1**(OOPSLA), 89:1–89:26.

van Ginkel, N., Strackx, R. & Piessens, F. (2017) Automatically generating secure wrappers for SGX enclaves from separation logic specifications. In *Programming Languages and Systems*, Chang, B.-Y. E. (ed). Cham: Springer International Publishing, pp. 105–123.

Van Strydonck, T., Piessens, F. & Devriese, D. (2019) Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, **ICFP**.

Van Strydonck, T., Piessens, F. & Devriese, D. (2020) Linear capabilities for fully abstract compilation of separation-logic-verified code - technical appendix including proofs and details. Available at: https://soft.vub.ac.be/~dodevrie/seplogic-lincaps-tr20201130.pdf.

Vogels, F., Jacobs, B. & Piessens, F. (2015) Featherweight VeriFast. *Log. Methods Comput. Sci.* **11**(3), 19:1–19:57.

Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N. W., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A. T., Moore, S. W., Murdoch, S. J., Nienhuis, K., Norton, R., Richardson, A., Rugg, P., Sewell, P., Son, S. & Xia, H. (2020) *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Technical Report UCAM-CL-TR-951. University of Cambridge, Computer Laboratory.

Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., & Vadera, M. (2015) CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In IEEE Symposium on Security and Privacy, pp. 20–37.

https://doi.org/10.1017/S0956796821000022 Published online by Cambridge University Press