Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code

THOMAS VAN STRYDONCK, KU Leuven, Belgium
FRANK PIESSENS, KU Leuven, Belgium
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium

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.

CCS Concepts:
• Security and privacy → Logic and verification; Security in hardware; • Software and its engineering → Compilers; General programming languages.

Additional Key Words and Phrases: fully abstract compilation, capabilities, verification, separation logic

ACM Reference Format:

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 (for example, when installing plugins from the internet). Our goal is to compile the verified code securely, i.e. 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.

Authors’ addresses: Thomas Van Strydonck, KU Leuven, Belgium, thomas.vanstrydonck@cs.kuleuven.be; Frank Piessens, KU Leuven, Belgium, frank.piessens@cs.kuleuven.be; Dominique Devriese, Vrije Universiteit Brussel, Belgium, dominique.devriese@vub.be.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of the ACM on Programming Languages, https://doi.org/10.1145/3341688.
As a concrete example of our approach, consider a separation-logic-verified password manager that runs locally on a user’s computer and allows for the installation of untrusted plug-ins (e.g. a password quality checker) to extend the functionality. The required separation logic contract for the downloaded plug-ins could, for instance, give the plug-ins access to the list of usernames and passwords, but forbid them from retaining references to it. This is, however, hard to enforce dynamically: an unverified plug-in to check password strengths could e.g. copy and store these references it has received and use them to freely read and write to the list of usernames and passwords, 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; they might read or write outside the intended ranges of the references they are provided with, 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 [Costan and Devadas 2016; Noorman et al. 2013; Strackx et al. 2010]. 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 verified attacker 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 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. 2018]. 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 really matter 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 and 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], i.e. 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), but to ensure interoperability, this code still needs to respect our calling convention: send and receive 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 towards a practically applicable secure compilation scheme, since the power of the separation-logic-verified source language is limited. Notably, 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. 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;
- 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 axioms and the full compiler. Section 5 formally defines full abstraction and discusses our proof approach. The back-translation needed for the full abstraction proof is illustrated in section 6. Sections 7 and 8 respectively discuss future and related work. Section 9 concludes.

In this paper, we have 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 technical report submitted as supplementary material.

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 afterwards 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.

The resource $n$ for address $a$ is what determines if $f$ 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 $\text{int}^\ast n$. Although these 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. These reified variables 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 $\text{int}^\ast n$. 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 $\text{int}^\ast_0 a$. The length-0 capability type $\text{int}^\ast_0 a$ denotes a non-linear capability of type int that cannot be dereferenced (this is in effect just an address). The reason $\text{int}^\ast_0 a$ is kept is for performing address operations and checks, as these non-spatial manipulations require no separation-logic resources.

Note that we cannot just discard the pointer $\text{int}^\ast_0 a$ and perform address operations and checks on the reified resource $n$ directly, as there is not necessarily a one-to-one connection between addresses and resources for them. The function $f$ could create multiple aliases for $a$, e.g. by 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

<table>
<thead>
<tr>
<th>Verified Component</th>
<th>Outcall Stub</th>
<th>Context Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void f(int* a)</td>
<td></td>
<td>void g(int* a)</td>
</tr>
<tr>
<td>//@pre n: a \rightarrow [0]</td>
<td></td>
<td>//@pre n: a \rightarrow [0]</td>
</tr>
<tr>
<td>//@post n: a \rightarrow [1] {</td>
<td></td>
<td>//@post n: a \rightarrow [0]</td>
</tr>
<tr>
<td>g(a); a[0] = 1 }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int* f(int<em>0 a,int</em> n) {</td>
<td></td>
<td>int* gstub(int<em>0 a,int</em> n)</td>
</tr>
<tr>
<td>n = gstub(a,n); n[0] = 1; return n }</td>
<td></td>
<td>n = g(a,n); guard(n != null) guard(length(n) == 1); guard(a == addr(n)); guard(n[0] == 0); return n</td>
</tr>
</tbody>
</table>

Fig. 2. Motivating example: a verified function $f$ interacts with an untrusted context function $g$.
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, avoids this mismatch.

Now that the need for linear capabilities has been established, we can use this example to illustrate the intuition behind the main two aspects of our compiler, namely the concept of stubs and the separation-logic-proof-directedness.

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 were 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.

The introduction of linear capabilities to represent reified source-level resources, combined with the fact that the capabilities need to have the correct contents when passed to the context as resources, 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 conclusion, 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.

In the sense that stubs enforce contracts at trust boundaries, they can be seen as a part of separation-logic-proof-directed compilation. The reification of separation logic resources described above is proof-directed as well, making proof-directedness the main theme of our compiler.

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, \( id_{\log} \) a logical variable, \( id_{\prog} \) 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 \( \text{symbol}[\text{parameter}] \) is used for parameterized symbols. The concrete notation for separation logic annotations and assertions is inspired by the VeriFast tool [Jacobs et al. 2010].

Both the source and target language (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.
Types To simplify types, the type bool is embedded in the type int, where 0 is true and k ≠ 0 is false. The target type τ* 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* is a regular non-linear heap pointer. A type (τ, ..., τ), representing length-n (n ≥ 0) tuples is present for all three cases. Pointers and arrays are seen as identical types τ* in our formalization, for simplicity's sake.
The target language has an extra type $\tau_n^{*0}$ of length-0 non-linear capabilities, used to store the compiled version of the source-level permissionless program variables (hence the use of $\tau_n$). The separation logic also employs a unique type list$_r$, 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$_{prog}$, which appear in source programs, and expressions exp over them on the one hand and logical (or symbolic) proof-only variables id$_{log}$, 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 exp$_{texp}$ additionally contain a function addr, which returns the $\tau_n^{*}$-type address of a $\tau_n^{*}$-type value, and a function length, which returns the length of the region addressed by a linear capability $\tau_n^*$. 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$_r$-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.

**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 assertions. Conditional assertions of the form exp ? assert express that assert only needs to hold if exp == true can be derived. Lastly, two types of assertions represent spatial resources: array and range resources. Array resources exp$_1 \mapsto_{\tau_n} exp$_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$_1 \mapsto_{\tau_n} [exp$_2$, . . . , exp$_k]$, which desugars to exp $\mapsto_{\tau_n} l * \overline{l[i]} = \overline{exp$_2^l * length(l) = k$ and was already demonstrated in figure 2.

Range resources [assert | exp$_1 \leq id_{log} < exp$_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$_{log}$. Range resources can be nested and will be useful for the back-translation in section 6.

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 language 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. Firstly, it returns a linear capability, not a regular pointer (since it essentially creates a target-level reified resource). Secondly, it guarantees a fresh heap location for the allocated variable. This avoid reuse of locations after free; if any newly allocated location could have been previously used by the context, it could have kept a non-linear 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, we do not consider free (see section 7 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. 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 \( n \) in two at a given index \( \text{sexp} \). For example, \(/@\text{split} \, n[1] \) 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 \( n : [\text{assert} \mid 0 \leq i < 3] \), the same \(/@\text{split} \, n[1] \) creates \( n' : [\text{assert} \mid 0 \leq i < 1] \) and \( n'' : [\text{assert} \mid 1 \leq i < 3] \). The source join statement is the inverse of split and merges two adjacent resources into one, e.g. both previous sets of resources \( n' \) and \( n'' \) are merged into \( n \) by \(/@\text{join} \, n' \, n'' \). 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 \(/@\text{collect} \, n' \cdot n'' \). Notice again how only the top-level resources is named. The proof tool has to decide, within the possibilities, which exact range resource to create. Conversely, the statement \(/@\text{flatten} \, n \) 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 a flatten and collect statement 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 \(/@\text{collect} \, n' \cdot n'' \) could be realized by the compiled code \( \tau* \, n; n[0] = n'; n[1] = n'' \).

**Functions** Two classes of functions exist: implemented functions with a function declaration and body, and context functions: declarations denoting the assumptions that a component makes about the function. For simplicity, tuple return types \((\tau^+)\) exist in both source and target language and every function has to end in a single return statement return \( \text{exp}^+ \). 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 \( \text{id}_{\text{log}} \), not program variables \( \text{id}_{\text{prog}} \).

**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 \( \text{id} \) forms an entire program.

### 3.2 Source Language Proofs

As explained, our compiler is separation-logic-proof-directed, i.e., 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 axioms that describe how to construct such proofs starting from the source code (section 4). **Hoare triples** are the building blocks of separation logic axioms [Reynolds 2002].
Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code

We will often use this notation in our formalization 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 \( \gamma \) and give rise to the extended Hoare triple notation \( \{ P \}_\gamma \ c \ { Q \}_\gamma \), stating that if \( (P, \gamma) \) holds, then either \( c \) diverges or \( (Q, \gamma') \) 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:

\[
P ::= \text{assert} \quad \text{(Symbolic Heap)} \quad c ::= \text{sstm} \mid \text{sstm; return sexp} \quad \text{(Source Code)} \\
\gamma ::= \{ \text{id}_{\text{prog}} : \text{exp} \}^* \quad \text{(Environment)} \quad \text{triple ::= \{ P \}_\gamma \ c \ { Q \}_\gamma} \quad \text{(Hoare Triple)}
\]

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

The two parts, \( P \) and \( \gamma \), of separation logic states have the following meaning:

- The environment \( \gamma \) maps program variables \( \text{id}_{\text{prog}} \) to expressions \( \text{exp} \) over logical, proof-level variables \( \text{id}_{\text{log}} \). The environment \( \gamma \) hence relates the program domain to the logical domain.

- The symbolic heap \( P \) is a \( s \)-separated list of assertions representing the symbolic program state. It is of the same form \text{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 \( s \), be it (part of) a function (body), a component or a program, the notation \( \vdash s \) represents a specific, valid separation logic proof tree for \( s \). This proof \( \vdash s \) 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:

\[
\frac{\text{Precondition}}{\langle \bar{s}, h \rangle \mid \bar{c} \leftrightarrow \langle \bar{s}', h' \rangle \mid \bar{c}' \quad \text{(RuleName)}}
\]

The small-step operational semantics \( \leftrightarrow \) transform a program state \( \langle \bar{s}, h \rangle \mid \bar{c} \) into a state \( \langle \bar{s}', h' \rangle \mid \bar{c}' \), where \( \bar{s} \) is the list of stack frames, \( h \) the heap, and \( \bar{c} \) a list of partly-executed function bodies, separated by return statements, where the sequence \( \bar{c} \) corresponds to the sequence of stack frames \( \bar{s} \). A function call creates a new stack frame and accompanying executing function body and adds them to \( \bar{s} \) and \( \bar{c} \) 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.

The memory model for both source and target languages is location-based, i.e. addresses are pairs \( (l, i) \) of an opaque location and an index \( i \). A malloc statement that allocates \( k \) units of type \( \tau \) creates a new location \( l \) in the heap, populated with default values (out of simplicity considerations) for type \( \tau \) at indexes \( 0 \) through \( k - 1 \). Separately malloced variables are hence logically separate.

Source pointer values of type \( \tau_{*i} \) follow the heap memory model and are hence denoted as couples \( (l, i) \). Target-level linear capabilities \( \tau_{*i} \), on the other hand, are denoted as \( [a, b] \), 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 \( [a, b] \) hence always points to index \( a \). Target-level length-0 capabilities are represented by \( l_0 \) 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 not for the target and the two differ markedly in two ways. Firstly where the built-in target-level functions join and split are concerned. Ghost statements in the source language only have an effect on the separation logic proof and are hence skipped, whereas they are reified and manipulate linear capabilities in the target. Ghost statements and their reification were already discussed in section 3.1.

Secondly when linearity of capabilities is involved. When a linear capability \( l[a, b] \) is copied, the original value is set to null. We call this process linear capability erasure. For example, if a tuple variable \( x \) has \( t_1^{[a_1, b_1]}, t_2^{[a_2, b_2]} \) as its stack value, an assignment \( y = x.2 \) of the second coordinate of \( x \) will erase this component, and \( x \) will have the stack value \( (t_1^{[a_1, b_1]}, \text{null}) \) afterwards. Erasure happens analogously when reading from the heap (where the erasure now happens in the heap), or when passing arguments to functions (this includes split and join). Additionally, whenever a single linear capability is used multiple times in any of the previous cases, the operational semantics get stuck. For example, using the same \( x \) from before, the assignment \( y = (x.2, x.2) \) gets stuck, as otherwise a linear capability would be duplicated.

4 AXIOMS AND COMPILATION BY EXAMPLE

This section introduces the separation logic axioms that constitute separation logic proof trees. Because our compiler is separation-logic-proof-directed, the compilation rules directly derive from these axioms, so we present both simultaneously. A relevant selection from both is spread over figures 6, 9, 10 and 11. The separation logic axioms and the compilation rules are obtained by respectively ignoring and not ignoring all green text. Compilation of a separation logic proof \( \vdash s \) to target code \( t \) is denoted \( \vdash s \Rightarrow t \). Other judgments appearing in these figures are explained as needed below.

We illustrate the rules using the running example in figure 4. Since our compiler is separation-logic-proof directed, it receives a proof of the verified component in figure 4 as input. For simplicity, we avoid foreach loops and range resources and stick to arrays of statically-known size, and we use the fixed-size array resource syntax discussed in section 3.1. The axioms and compilation rules will still be presented in their general form.

By \( f \)'s contract in figure 4, 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 \( \text{add1} \) to add the 1. The contract of \( \text{add1} \) specifies that it takes a pointer \( a \) and a resource \( m \) to access a one-element array corresponding to \( a \). It returns the value \( \text{result} \), 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 \( \text{result} \) is a privileged name, used to denote a function’s return value(s) in its postcondition (\( \text{result} \) is used in case of multiple return values). Based on the value of \( a \)'s first element \( c \), \( f \) either calls \( \text{add1} \) 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 \( \text{add1} \).

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

4.1 Structural Rules

Structural rules are axioms 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 \( \text{ConseqPost}, \text{Frame}, \text{Seq} \) and \( \text{If} \), presented in figure 6.
Fig. 4. Illustrating example: conditionally add 1 to the second element of a length-2 array or its negation.

```c
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
}
```

Fig. 5. Separation logic proof of the function given in the illustrating example.

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

    int m0; int a, m1, m2; m1, m2 = split(m1);
    if c == 0
        then int r1; res,r1 = add1comp(a+1,m2)
    else int r0 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
}
```

```c
int add1comp(int a, int m)
{
    int res; int c; c = m[0];
    int m1; int m2; m1, m2 = split(m1);
    if c == 0
        then int r1; res,r1 = add1comp(a+1,m2)
    else int r0 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
}
```

```c
{ m : a → int [c, a1] } \{ a.a \}
int res;
{ m : a → int [c, a1] } \{ a.a || res0 \}
int c; c = a[0];
//@split m[1];
{ m1 : a → int [c] } \{ a.a || res0 || c : c \} + m2 : a + 1 → int [a1] \{ a.a || res0 || c : c \}
if c == 0
    then
        { m2 : a + 1 → int [a1] * c != 0 \{ (a.a || res0) \} b: int a1;
    { m2 : a + 1 → int [a1] } \{ a + 1 || res0 || b: null || a1 \}
        a1 = (a+1)[0];
    b = malloc(1 * sizeof(int));
    { c != 0 \{ (res0 || b: null || a1) \} a1 }
    b[0] = -a1;
    { c != 0 + l : y → int [0] \{ (res0 || b: y || a1) \} a1 }
    res = add1(b);
    { c != 0 + x : y → int [−a1] \{ (res0 || b: y) \} x == (c == 0 ? a1 + 1 : −a1 + 1),
    return res
    { result == (c == 0 ? a1 + 1 : −a1 + 1) },
```
They are the elementary building blocks of the symbolic execution in figure 5 and the workhorses of the separation-logic driven compiler. The rules are named after the source statements they create a proof for, i.e., `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 9.

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 5 and the workhorses of the separation-logic driven compiler. The rules are named after the source statements they create a proof for, i.e., `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 9.

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

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 ⊢ 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 resources, particularly fabricating conditional resources of the form `false ? assert`. Both these operations will influence the compiled code `p`, but are omitted for brevity.

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 5 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 e.g. 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.
In the following descriptions of the inference rules, $\text{sexp}_\gamma$ denotes $\text{sexp}$ where each program variable $id_{\text{prog}}$ is substituted by $\gamma(id_{\text{prog}})$ (implicitly requiring that $id_{\text{prog}} \in \text{dom}(\gamma)$). Also note that compilation is the identity for expressions $\text{sexp}$, since the variables contained in $\text{sexp}$ are automatically converted to address capabilities by the compilation.

The auxiliary judgments $\tau_s \rightsquigarrow_{\text{CompileType}} \tau_t$, which compiles a source type $\tau_s$ to the corresponding target type $\tau_t$, and $\tau_s \rightsquigarrow_{\text{def}} v$, which returns the default value $v$ for the type $\tau_s$, are first defined in figures 7 and 8 respectively.

The remainder of this section consists of a discussion of the depicted basic separation logic axioms and compilation rules. Separation logic axioms are illustrated by referencing lines from figure 5. Compilation rules are illustrated using a combination of source code lines from figure 5 and lines from the compiled verified function $f_{\text{comp}}$ in figure 4.

The $\text{MALLOC}$ axiom assigns a fresh logical variable $id_{\text{log}}$ to $id_{\text{prog}}$ in $\gamma$ and creates a new array resource $n$, consisting of the default values $v$ repeated $\text{sexp}_\gamma$ times. In the corresponding $\text{MALLOC}$ compilation rule, the variable $\tau' * n$ is declared and assigned the malloced linear capability, i.e. the resource corresponding to $id_{\text{prog}}$ 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 axiom). As $id_{\text{prog}}$ is itself merely a permissionless address on the target level, we assign it using the $\text{addr}$ function, maintaining the correspondence between $a$ and $n$ from the separation logic proof. The $\text{MALLOC}$ axiom is demonstrated on lines 18-20, where a new resource $l$ is created. Lines 18-20 are compiled to lines 7-8 in $f_{\text{comp}}$.

The $\text{SPLIT}$ and $\text{JOIN}$ rules for both array resources and range resources are dual, with the difference that $\text{SPLIT}$ has to check whether the given splitting index to split on is in bounds, whereas $\text{JOIN}$ has to check memory adjacency of the two resources it is given. Given the similarities, only the array version of the $\text{SPLIT}$ rule is detailed in figure 9. In the $\text{SPLIT}$ compilation rule, the compiled code performs the target-level split operation on the corresponding reified linear capability, mirroring the split that happens on resources in the separation logic proof. The $\text{SPLIT}$ axiom is demonstrated on lines 5-7, where resource $m$ is split. Lines 5-7 are compiled to line 3 in $f_{\text{comp}}$.

The axioms $\text{VARDECL}$ and $\text{VARASGN}$ prove variable declaration and assignment, respectively. Their compilation rules are straightforward, apart from the change in type in $\text{VARDECL}$. The $\text{VARDECL}$ axiom is demonstrated on e.g. lines 1-3, which compile to the first declaration on line 2 in $f_{\text{comp}}$.

The axioms $\text{ARRAYMUT}$ and $\text{ARRAYLKUP}$ are very similar, so figure 9 only shows the former. The mutation of source arrays is again compiled to the same action on the corresponding reified resource. Both axioms are demonstrated on lines 20-22 and lines 16-18, respectively. Line 17 contains $(a + 1)[0]$ and not $a[1]$ because $(a + 1)_\gamma$ is equal to the logical address of the resource $m_\gamma$ and $a_\gamma$ is not, and both axioms enforce this address equality for simplicity. These lines respectively compile to lines 9 and the end of line 6 in $f_{\text{comp}}$.
\[\begin{align*}
\tau \xrightarrow{\text{def}} \nu & \quad \tau \xrightarrow{\text{CompileType}} \tau' \quad n, \text{id}_{\text{log}} \text{ fresh} \\
\text{id}_{\text{prog}} \in \text{dom}(\gamma) & \quad \gamma' = \gamma[\text{id}_{\text{prog}} : \text{id}_{\text{log}}] \\
\text{(ALLOC)} & \\
\{ \text{sexp}_y > 0 \}_\gamma & \quad \text{id}_{\text{prog}} = \text{malloc}(\text{sexp}_x \ast \text{sizeof}(\tau)) \\
\{ n : \text{id}_{\text{log}} \rightarrow_r \text{repeat}(\text{sexp}_y, \nu) \}_\gamma & \quad \tau' \ast \nu; \ n = \text{malloc}(\text{sexp}_x \ast \text{sizeof}(\tau)); \\
\text{id}_{\text{prog}} = \text{addr}(n) & \quad \tau' \ast n; \ n' = \text{malloc}(\text{sexp}_x \ast \text{sizeof}(\tau)); \\
\text{id}_{\text{prog}} \notin \text{dom}(\gamma) & \quad \tau' \xrightarrow{\text{def}} \nu \\
\text{id}_{\text{prog}} \in \text{dom}(\gamma) & \quad \gamma' = \gamma[\text{id}_{\text{prog}} : \nu] \\
\text{(VARDECL)} & \\
\{ y \}_\gamma \text{id}_{\text{prog}} \{ y \}_\gamma & \quad \tau' \xrightarrow{\text{id}_{\text{prog}}} \\
\{ y \}_\gamma \text{id}_{\text{prog}} = \text{sexp} \{ y \}_\gamma & \quad \tau' \xrightarrow{\text{id}_{\text{prog}}} \\
\text{(VARASGN)} & \\
\{ y \}_\gamma \text{id}_{\text{prog}} = \text{sexp} & \quad \gamma' = \gamma[\text{id}_{\text{prog}} : \text{sexp}_y] \\
\text{(RETURN)} & \\
\{ y \}_\gamma \text{stm} \{ Q \}_y & \quad \gamma' \xrightarrow{\text{p}} \quad \text{CN}(Q) = \overline{\text{n}} \\
\{ y \}_\gamma \text{stm}; \text{return} \overline{\text{sexp}} & \quad \gamma' \xrightarrow{\text{p}} \quad \text{CN}(Q) = \overline{\text{n}} \\
\text{(FAPP)} & \\
\{ \text{PRE}[\text{subst}_{\text{pre}}] \}_y \overline{\text{id}} = f(\overline{\text{sexp}}) \{ \text{POST}[\text{subst}_{\text{post}}] \}_y & \quad \gamma' \xrightarrow{\overline{\text{n}}} \overline{\text{n}}; \ \{ \overline{\text{id}}, \overline{\text{n}} \} = \text{fcomp}(\overline{\text{sexp}}, \overline{\text{m}})
\end{align*}\]

Fig. 9. Basic separation logic axioms that can be extended to compilation rules

The Guard rule adds the asserted conditions to the symbolic heap and compilation is the identity. Function application FApp is the most intricate basic axiom. The variable \(\Sigma\) denotes a componentwise 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 \text{PRE} and \text{POST}, the concrete logical expressions \(\overline{\text{sexp}}_y\) used in the function call are substituted for the arguments \(\overline{\text{id}}_{\text{arg}}\), instantiating the callee’s contract with the caller-provided arguments. Additionally, in the caller’s postcondition, the privileged \(\overline{\text{id}}_{\text{res}}\) variables are substituted with fresh logical variables \(\overline{\text{id}}_{\text{res}}\), linked to \(\overline{\text{id}}\) in \(\gamma'\). Fresh resource names \(\overline{\text{n}}\) are required to avoid name clashes.

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

The FAPP axiom is illustrated on lines 10-12 and lines 22-24. For lines 10-12 for example, \( \text{[subst}_{\text{pre}}] == [a \mapsto a + 1] \) and \( \text{[subst}_{\text{pre}}] == [a \mapsto a + 1][\text{result} \mapsto x] \), where \( [\text{res} : x] \) 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 \( \text{CONSEQ} \) on line 12). Lines 10-12 are hence compiled to line 5 in \( f_{\text{comp}} \), where \( \text{sum}_{\text{stub}} \) denotes the outcall stub for \( \text{add1} \).

The \text{RETURN} axiom forms a special case because return is not a source statement; it appears exactly once at the end of each function body. Since \( \text{SEQ} \) can only sequence source statements, the \text{RETURN} axiom has to manually construct a new proof from a previous one. Conceptually, however, \text{RETURN} is a basic rule. Given a proof of \( \text{stm} \), the \text{RETURN} axiom introduces the privileged 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 \( \text{CONSEQ} \) 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 axiom and are compiled to line 11 in \( f_{\text{comp}} \). No returned variables are added because \( f \) leaks its resources. Notice that line 27 follows from \text{RETURN’s} postcondition \( \{x \equiv (c \equiv 0 ? a_1 + 1 : -a_1 + 1) \land \text{result} \equiv x\} \) by the \( \text{CONSEQ} \) 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 \( \text{ImplFVerif}, \text{ContFVerif}, \text{CompVerif} \) and \( \text{ProgVerif} \), presented in figure 10 and discussed below.

First, we have the axioms for implemented functions \( \text{ImplFVerif} \) and context functions \( \text{ContFVerif} \). The main difference between both axioms is that \( \text{ContFVerif} \) expects to reference any function body, whereas \( \text{ImplFVerif} \) requires a proof of the function body to construct a function proof \( \vdash \). The precondition environment of this proof is \( [id_{\text{arg}} : id_{\text{arg}}] \), since our separation logic contract preconditions always implicitly map the function arguments \( id_{\text{arg}} \) to logical variables of the same names \( id_{\text{arg}} \). This environment is non-coincidentally the starting environment in figure 5. This initial environment explains how we can allow function contracts to be entirely logical assertions, but still reference function arguments \( id_{\text{arg}} \).

The corresponding compilation rules both use an auxiliary judgment \( sf\text{func} \cong_{\text{Decl}} tf\text{func} \) that compiles a function declaration \( sf\text{func} \) to a declaration \( tf\text{func} \) by reifying all resources in the given pre- and postcondition into arguments and return values respectively, and compiling existing argument and return types using \( \cong_{\text{CompileType}} \). The rule \( \text{ImplFVerif} \) also changes any implemented function \( f \)’s name to \( f_{\text{comp}} \) during compilation, again for stub-related reasons explained in section 4.4. The proof of figure 5 suffices to construct a proof \( \vdash \) of \( f \) using \( \text{ImplFVerif} \), which can then be compiled to the target-level function \( f_{\text{comp}} \) in figure 4. The declaration of \( \text{add1} \), on the other hand, is compiled to the declaration of \( \text{add1} \) using \( \text{ContFVerif} \).

The component verification rule \( \text{CompVerif} \) allows verification and compilation of entire components. A component \( s\text{comp} \) has a proof \( \vdash s\text{comp} \) if it is well-formed (\( \vdash_{\text{WF}} s\text{comp} \), this includes some restrictions mentioned in section 4.4) and if all its implemented and (exported and imported)
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, i.e., 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 11. The Incall compilation rule that generates the incall stubs for exported functions is an analogous but simpler version of Outcall and hence not detailed.

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, i.e., functions that are imported or exported by any module, boundary functions.

As an example, given the proofs of \( f \) and \( \text{add1} \) constructed in the previous paragraph, the CompVerif axiom proves the source component in figure 4, which has \( f \) as an internal function, an empty export list and the declaration of \( \text{add1} \) as the import list. The source component is compiled to the target component in the bottom row of figure 4, where \( \text{add1 stub} \) is the outcall stub resulting from the application of \( \sim \outcall \) on \( \text{add1} \). 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 \( \sim \incall \) 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 \( \text{sprog} \) has a proof \( \vdash \text{sprog} \) 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.
No **Outcall** or **Incall** separation logic axiom 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 $\text{csfunc} \rightsquigarrow_{\text{Outcall}} \text{ctfunc}$, $\text{stub}_{\text{outcall}}$ that both compiles a context function $\text{csfunc}$ to $\text{ctfunc}$ using Cont$\text{FVerif}$ and generates an outcall stub $\text{stub}_{\text{outcall}}$ 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 $n : \text{exp} \mapsto [\text{exp}_1, \ldots, \text{exp}_p]$ 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 $\text{PRE}$ and $\text{POST}$ are separable into a spatial heap $\overline{m} : \text{PRE}_s$ or $\overline{n} : \text{POST}_s$, consisting of a sequence of separating-conjunction separated fixed-size array resources (whose names $\overline{m}$ and $\overline{n}$ we externalize in our notation), and a pure heap $\text{PRE}_p$ or $\text{POST}_p$, consisting of pure conditions. The **Outcall** rule will handily use 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 $\text{PRE}_s$ and $\text{POST}_s$ and the set of names $\text{result}$ must be mutually distinct and cannot contain duplicates. This makes any otherwise implicit conditions in $\text{PRE}_s$ and $\text{POST}_s$ explicit in $\text{PRE}_p$ and $\text{POST}_p$, 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 $\text{add1}$ in figure 4 is linearized to:

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

Lastly, we require boundary functions to not introduce any new logical variables (except for $\text{result}$) in their spatial heaps $\text{PRE}_p$ and $\text{POST}_p$. This will make the constraints in $\text{PRE}_p$ and $\text{POST}_p$ 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 $a$, $a^{\text{pre}}$, $a_1^{\text{pre}}$, $a^{\text{post}}$, $a_1^{\text{post}}$ and $\text{result}$ in the compiled code. The restrictions imposed by these last two assumptions are included as part of the component well-formedness $\text{WF}_{\text{CompVerif}}$.

An outcall stub then needs to generate code in order to check any constraints present in the untrusted function’s postcondition $\text{POST}$.

We first investigate what information from $\overline{m} : \text{PRE}_s$ and $\overline{n} : \text{POST}_s$ we need to reify to be able to check $\text{POST}$. Both $\overline{m} : \text{PRE}_s$ and $\overline{n} : \text{POST}_s$ are linear and hence use fresh variable names that can reappear in conditions in $\text{POST}_p$. These variables hence need to be reified, i.e. declared and assigned the right values, so they can be used when reifying $\text{POST}_p$’s conditions. Additionally, any constraints present in the linear spatial heap $\overline{n} : \text{POST}_s$ 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 these checks need to be performed by a guard statement for each $n$.

We need a way to reify the aforementioned checks $\text{check}$, declarations $\text{decl}$ and assignments $\text{assign}$ for a given spatial assertion $\text{assert}_s$ (i.e. $\overline{m} : \text{PRE}_s$ or $\overline{n} : \text{POST}_s$). This is the function of the auxiliary
Therefore, no declarations or assignments will be reified; only checks. Since outcall stubs only check declarations and assignments will be reified; only checks. Since outcall stubs only check the function call to _f_, since the reified resources _m_ might be altered by _f_. Postcondition-related declarations _d_post_, assignments _a_post_ and checks _c_post_ and _c_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 _add1comp_ for _add1_ in figure 4 gives an example of the generated stub _stub_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 $d_{\text{pre}}$ can be found on line 2 and the assignments $a_{\text{pre}}$ on line 3. The declarations $d_{\text{post}}$ are generated on line 7, assignments $a_{\text{post}}$ on line 8 and checks $c_{\text{post}}$ and $c_{\text{post'}}$ 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. In this section, notions relating to the source and target language are typeset in green and pink respectively. The first subsection 5.1 formally defines both directions of full abstraction. Subsection 5.2 provides some high-level proof intuitions, highlighting the need for a back-translation.

5.1 Definition

Terms $x$ and $x'$ are contextually equivalent, denoted $x \equiv_{\text{ctx}} x'$, if $\forall C : C[x] \Downarrow \iff C[x'] \Downarrow$ 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 $\mathcal{C}_s$ or $\mathcal{C}_t$, which is a sequence of components, and a main function identifier $//@main = id$, identifying the main function to execute when starting program execution. A context is hence denoted $(\mathcal{C}_s, id)$ and an entire program $\mathcal{P}(x)//@main = id$. In our source language, the notion of plugging from the contextual equivalence definition above also requires (next to the usual well-formedness constraints) that given the source component proof $\vdash_s$ and the context $(\mathcal{C}_s, id)$, a program proof $\vdash \mathcal{C}_s[s]//@main = id$ exists.

Full abstraction is then defined as the reflection and preservation of contextual equivalence $\equiv_{\text{ctx}}$ [Abadi 1999]. Given source components $s$ and $s'$ and target components $t$ and $t'$, we have that compilation is fully abstract iff $\vdash_s \rightsquigarrow t \land \vdash_s' \rightsquigarrow t'$ implies that $(t \equiv_{\text{ctx}} t' \iff s \equiv_{\text{ctx}} s')$. This statement depends on the chosen proofs $\vdash$ of $s$ and $s'$, 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. This is, however, not a problem; since our compiler does not alter control flow, it is easy to prove that it preserves divergence and stuck-ness, as expected.

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:

\[
\begin{align*}
\forall s, s', t, t'. \vdash s \rightsquigarrow t \land \vdash s' \rightsquigarrow t' \Rightarrow (t \equiv_{\text{ctx}} t' \Rightarrow s \equiv_{\text{ctx}} s') & \quad \text{(Correctness)} \\
\forall s, s', t, t'. \vdash s \rightsquigarrow t \land \vdash s' \rightsquigarrow t' \Rightarrow (t \equiv_{\text{ctx}} t' \Leftarrow s \equiv_{\text{ctx}} s') & \quad \text{(Security)}
\end{align*}
\]

5.2 High-Level Proof Approach

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

$\vdash \mathcal{C}_s[s]//@main = id \rightsquigarrow \mathcal{C}_t[t]//@main = id \Rightarrow (\mathcal{C}_s[s]//@main = id \Downarrow \iff \mathcal{C}_t[t]//@main = id \Downarrow)$

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

$\mathcal{C}_s[s]//@main = id \Downarrow \iff \mathcal{C}_t[t]//@main = id \Downarrow \iff \mathcal{C}_t[t']//@main = id \Downarrow \iff \mathcal{C}_s[s']//@main = id \Downarrow$. 

The equi-termination of verified code and its compilation is proven using an auxiliary \textit{simulation relation}, that relates executing source and target code [see e.g. Chlipala 2017].

We would like to repeat the above process for the \textsc{Security} direction, i.e. prove $t \simeq_{\text{ctx}} t'$ through some form of equi-termination between source and target code. To prove $t \simeq_{\text{ctx}} t'$, we need to prove (by definition) that $\mathcal{C}_t[/@main = id] \Downarrow \Leftrightarrow \mathcal{C}_t'[/@main = id] \Downarrow$, given any target context $(\mathcal{C}_t, id)$ and source component proofs $\triangleright s$ and $\triangleright s'$. There is, however, one problem: since we start from a target context $(\mathcal{C}_t, id)$ rather than a source context $(\mathcal{C}_s, id)$, we cannot use our compiler to construct an equi-terminating source program for us. Simply inverting the compiler is impossible, since its range is not the entire 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 $(\mathcal{C}_t, id)$ and a source component proof $\triangleright s$. This target-to-source code transformation is called the \textit{back-translation}, denoted $(\mathcal{C}_t, id), \triangleright s \rightsquigarrow \mathcal{C}_s[//@main = id]$, and is a standard tool in full abstraction proofs. The proof $\triangleright s$ is necessary because the back-translated context $(\mathcal{C}_t, id)$ needs to result in a sound program proof $\triangleright \mathcal{C}_s[//@main = id]$ (since $s \simeq_{\text{ctx}} s'$ requires verified code).

With such a back-translation, we can repeat the process used to prove \textsc{Correctness}. If we can prove that target code and its back-translation equi-terminate, i.e. if it holds for any contexts $(\mathcal{C}_s, id)$ and $(\mathcal{C}_t, id)$ and any components $s$ and $t$ (with $\triangleright s \rightsquigarrow t$) that:

$$ (\mathcal{C}_t, id), \triangleright s \rightsquigarrow h \triangleright \mathcal{C}_s[s]//@main = id \implies (\mathcal{C}_s[s]//@main = id \Downarrow \Leftrightarrow \mathcal{C}_t[t]//@main = id \Downarrow) \tag{1} $$

Then $t \simeq_{\text{ctx}} t'$. The reasoning is analogous to the one for \textsc{Correctness} above. Equi-termination of target code and its back-translation is again proven using a (different) \textit{simulation relation}.

6 PROVING SECURITY: THE BACK-TRANSLATION

Similarly to how section 4 introduced compilation using figure 4, this section will introduce the back-translation by means of an example, presented in figure 12, that builds further on the compilation example. The goal of this section is to illustrate how the back-translation of this example satisfies one specific instance of equation (1) in the previous section, thereby illustrating the main ideas behind our general back-translation.

The verification of source function $f$ from figure 4 acts as our verified component $\triangleright s$ and hence $f_{\text{comp}}$ and $add1_{\text{comp}}$ together act as our target component $t$, where $\triangleright s \rightsquigarrow t$ by construction. Our target context $(\mathcal{C}_t, id)$ then consists of the target function $add1$'s implementation in figure 12 and any choice for the main function $id$, e.g. $f$. Notice how $f_{\text{comp}}$, $add1_{\text{comp}}$ and $(\mathcal{C}_t, id)$ together form a sound target-program, as required by the definition of contextual equivalence $t \simeq_{\text{ctx}} t'$ in section 5.2. To prove this specific instance of (1), we back-translate the example context $(\mathcal{C}_t, id)$ to a context $(\mathcal{C}_s, id)$ such that $\triangleright \mathcal{C}_s[s]//@main = id$ is a valid separation logic proof, and $\mathcal{C}_s[s]//@main = id$ and $\mathcal{C}_t[t]//@main = id$ equi-terminate.

The implementation of the context function $add1$ in figure 12 upholds the source-level contract of $add1$ in figure 4 that the verified component $\triangleright s$ expects of it: 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. Note that $(\mathcal{C}_t, id)$ is \textit{not} required to behave properly like this! It might also add 2 instead, change the contents of the resource $m$, etc., effectively ignoring $f$'s expectations and causing execution to get stuck at the guard statements of the outcall stub $add1_{\text{comp}}$. This extra freedom of the target context to misbehave 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 programs that are incapable of breaking $\triangleright s$’s verification guarantees.

We first define the back-translation of types and expressions. Target types that can result from compilation of source types, i.e. int, length-0 capabilities $\tau_0$ and tuples $(\tau_1^*)$, are 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 let us explain how we represent linear capabilities
in the source language and hence back-translate all types in the next paragraph. After that, we discuss simulating linearity in order to preserve equi-termination.

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 be able to extract both, the target language contains built-in addr and length functions. The source language does not have (or need) these functions, since its pointers are of the form $(l, i)$, are non-linear and do not have length information built in. To solve this information discrepancy between both pointer memory models, we introduce a fat pointer scheme to back-translate linear capabilities. We back-translate each linear capability of type $\tau^*$ to a pair $(\tau^*, \text{int})$. The first element $\tau^*$ is a pointer to the $\tau^*$-typed contents of the recursively back-translated capability that simultaneously represents the pointer’s address, since $\tau^*$ is non-linear and allows pointer arithmetic. The second int element is the externalized length of the capability. The null-pointer null is back-translated to the fat pointer $(\text{null}, 0)$. In conclusion, we can define a judgment $\tau \sim_{\text{InvCompileType}}^\tau$, dual to $\tau \sim_{\text{CompileType}}^\tau$ from section 4.2, that recursively back-translates target types as follows:

\[
\begin{array}{c}
\tau \sim_{\text{CompileType}}^\tau' \\
(\text{INVERTCOMPILATION}) \hline \\
\tau' \sim_{\text{InvCompileType}}^\tau \\
\end{array}
\]

Unfortunately, the back-translated fat pointers will not automatically behave linearly. The problem of simulating the linear behavior, as described in section 3.3, of target-language capabilities still has to be solved. Extra statements have to be added during the back-translation to imitate the target-language erasure of capabilities. When e.g. back-translating the assignment $x = n[2]$ with $n$ of target type int*, an erasure assignment $n[2] = (\text{null}, 0)$ has to be added in the source (remember that $(\text{null}, 0)$ is the fat null-pointer). 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. When e.g. back-translating the assignment $x = (n, n)$, with $n$ as before, the source language needs to add a statement guard(false) to ensure equi-termination. For simplicity, our example implementation of add1 avoids both these scenarios.

The back-translation of expressions $\text{texp}$, denoted $\text{texp}_{bt}$, is now easy to define, since target and source expressions only differ in the addr and length functions mentioned before, which we created our fat pointer scheme for. The back-translation hence maps $\text{addr}(\text{texp})$ to $\text{texp}_{bt, 1}$ (the address is the first part of the fat pointer), $\text{length}(\text{texp})$ to $\text{texp}_{bt, 2}$ (the length is the second part of the fat pointer) and $\text{null}$ to $(\text{null}, 0)$ (as mentioned before). All other cases are the identity.

With these prerequisites, we can examine the back-translation $\text{add1}_{bt}$ of add1. The end goal is to understand figure 13: the back-translation equivalent of figure 5, which shows a symbolic execution for the body of $\text{add1}_{bt}$. The notation $\sim$ is used to denote an unchanged symbolic heap.

A first question is what the separation-logic contract of the back-translated function $\text{add1}_{bt}$ should be. The desired contract is the one $t \sim s$ expects for add1 in figure 4, so that the resulting source program $\text{C}_{s}[s]//@main = \text{id}$ 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 $t \sim s$, there are no restrictions on their contract whatsoever. The solution is to employ the most general contract possible. Given that the back-translation tries to mimic the target language, this means that the contract needs to at least simulate the linear capabilities in the target language as separation logic resources in the source language. For the precondition, each back-translated function $f$ has a description of the minimal resources required to exactly match the permissions $f$ receives from linear capabilities at call-time in the target language. Analogously, the postcondition describes the most general
separation logic resources required to match the permissions associated with the returned linear capabilities in the target language. We call this type of contract a *universal contract* and define it in the next paragraph. Lines 2-3 of \texttt{addI}_{\texttt{id}}\texttt{b} in figure 12 show the universal contract for this function.

Universal contracts \texttt{univ\_contr}_{\gamma}(\tau_i) are associated with back-translated program variables \texttt{id}_{\texttt{prog}} and indexed by their target type \tau_i, since this type determines the target-language permissions associated with the variable, which the universal contract represents as separation logic assertions. Universal contracts are separation logic assertions and hence cannot take program variables like \texttt{id}_{\texttt{prog}} as a direct argument. Instead, they are applied to the variable’s logical-domain interpretation \(\gamma(\texttt{id}_{\texttt{prog}})\). In other words, the universal contract for a program variable \texttt{id}_{\texttt{prog}} of type \tau_i is given by \texttt{univ\_contr}_{\gamma}(\gamma(\texttt{id}_{\texttt{prog}})).

We now present a simplified definition for universal contracts, omitting the case for nested capabilities, since we will not need it for our example:

\begin{definition}
\texttt{univ\_contr}_{\gamma}(\exp) =
\begin{align*}
\text{true} & \quad \text{if } \tau_1 = \text{int or } \tau_1 = \tau_2 \ast 0 \\
\text{univ\_contr}_{\tau_1, \ldots, \tau_k}(\exp) & = \text{univ\_contr}_{\tau_1}(\exp.1) \ast \ldots \ast \text{univ\_contr}_{\tau_k}(\exp.k) \\
\text{exp} & = \text{null}(0) \quad \text{if } n : [\exp.1 + i \mapsto \tau_i, l[i] \mid 0 \leq i < \text{length}(l)] \ast \text{length}(l) = \text{exp.2}
\end{align*}
\end{definition}

The case for target-capabilities \(\tau_1\ast\) is the only non-trivial one. It states that a linear capability is either the fat null-pointer \texttt{null}(0), or that we have a range resource that allows us to access each element of the capability (without knowing anything about each element’s value), where \exp.1 is the capability’s address and \exp.2 its length. The fat-pointer scheme appears in the logic, because the \(\tau_1\ast\)-typed value \texttt{id}_{\texttt{prog}} is back-translated to a fat pointer \texttt{id}_{\texttt{prog}} in the source.

The reason for using a range-expression resource rather than a variable-length array resource above, is related to the omitted nested linear capabilities. Permissions for them are impossible to represent using flat array resources. Using range-expressions entails that whenever we require an array resource to satisfy the precondition of a separation-logic axiom in the back-translated code, we need to convert the range resource from the universal contract to an array resource for applying the axiom, and back afterwards. This is exactly what happens in lines 6-9 of \texttt{addI}'s back-translation \texttt{addI}_{\texttt{id}}\texttt{b}; the resource we have for the fat source pointer \texttt{m} is a range-expression, but we need an array resource to apply \texttt{ARRAYLKUP} on line 6. Lines 7-13 in figure 13 exemplify the conversion between an array and a range resource in both directions. A length-1 range resource \texttt{m}_{\text{chunk}}^0\text{flat} (not explicitly shown) is split from \texttt{m}_{\text{chunk}}^0\text{chunk} and flattened to the array resource \texttt{m}_{\text{chunk}}^0\text{flat} on lines 7-9. The axiom \texttt{ARRAYLKUP} can now be applied on lines 9-11. Afterwards, the resource is recollected to reobtain the range resource \texttt{m}_{\text{chunk}}^0\text{chunk} and rejoined to the rest of the range resource on lines 11-13.

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 postcondition 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 \texttt{tstm} declares variables \texttt{V}_{\texttt{tstm}} and variables \texttt{V}_{\texttt{pre}} were previously declared and if \texttt{tstm} \leadsto_b \texttt{sstm} holds, then \texttt{sstm} has as contract (omitting type subscripts) \(\{\texttt{univ\_contr}(\gamma_{\texttt{pre}}(V_{\texttt{pre}}))\}_{\gamma_{\texttt{pre}}}\texttt{sstm} \{\texttt{univ\_contr}(\gamma_{\texttt{post}}(V_{\texttt{pre}} \cup V_{\texttt{tstm}}))\}_{\gamma_{\texttt{post}}}\). The three statements (including return) of \texttt{addI} are back-translated to the three blocks on lines 1-4, 4-14 and 14-16 in the proof of figure 13, showcasing this block-level 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; a back-translated function’s precondition is hence $\text{univ\_contr}(id)$, as demonstrated on line 1 of figure 13. The postcondition is an exception, since the caller only cares about the privileged $result$ variables and resources over them. The universal postcondition is hence $\text{univ\_contr}(result)$, as demonstrated on line 17 of figure 13. This postcondition is achieved through **CONSEQUENCE** 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. As an example, we investigate the back-translation of the array lookup on line 2 in $add1$, i.e. the block on lines 4-14 in figure 13. First, to apply the $\text{ARRAYLKUP}$ separation-logic axiom, 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. Since the non-functional universal contract does not provide us such guarantees, lines 4-7 add these conditions and derive the necessary preconditions for $\text{ARRAYLKUP}$. Failing such a guard statement would make the back-translated program get stuck, but the target operational semantics would also get stuck in these faulty cases, preserving equi-termination. Second, the core axiom $\text{ARRAYLKUP}$ is applied on lines 9-11. Finally, on lines 13-14, **CONSEQUENCE** is applied to forget the information added through the guard statements and the array lookup again, making the postcondition universal again. The back-translations of other statements follows a structure similar to the one described here for array lookups.

We can now back-translate regular target functions, but the resulting universal contracts do not match the concrete contracts that the source context $\vdash s$ expects, e.g. the contract for $add1$ in figure 4 does not match the universal contract of $add1_{bt}$ in figure 12. When $f$ performs an outcall to $add1$, this does not directly correspond to an invocation of $add1_{bt}$: the representation of the back-translated resource needs to be constructed and its universal contract satisfied from $f$’s concrete precondition. When $add1$ returns control to $f$ afterwards, on the other hand, we need to find a way to go from $add1_{bt}$’s universal postcondition to $f$’s concrete expectations for $add1$’s postcondition. We need a way to tie the knot and link both contracts.

When we had a similar mismatch between expectations on trust boundaries during compilation, we introduced stubs to enforce contracts on the target level. Enforcing a contract is exactly what we need to do here, but now on the source level. We therefore reexamine the outcall stub $add1_{comp}$ (repeated in figure 12) that $f$ generates for $add1$, based on the contract it expects $add1$ to uphold. This outcall stub contains reified checks (i.e. the guard statements on line 6 and lines 9-10) for all conditions present in the postcondition of $add1$. If we can in some way back-translate the outcall stub $add1_{comp}$ and place it between $f$ and $add1$, it will connect both function’s arguments and pre- and postconditions and use guard statements to add the missing concrete conditions to $add1_{bt}$’s universal postcondition. This back-translated outcall stub, called $add1$ to match the name used by $f$, is shown in the top left corner of figure 12. It mostly consists of straightforward back-translations of the individual statements of $add1_{comp}$, with a few caveats, caused by the fact that $f$ is a regular verified source function, whereas $add1_{comp}$ is a back-translated function that mimics a target function. The non-obvious aspects are the following:

- **The proof that $add1$ has a valid contract needs to be constructed, to prove soundness of the back-translation.** Unlike in 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 means that only the middle, core operation phase for each back-translated block is kept.
- **The function $f$ is unaware of the back-translated reified resource $m$, whereas $add1_{comp}$ 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 $m$ in the precondition.
• The function $f$ uses array resources, whereas $add1_{\text{comp}}$ uses range resources. Lines 7 and 11 in $add1$ switch between both representations, similarly to what happened on lines 7-13 of figure 13.
• The back-translated guard statements also guarantee equi-termination between source and target language when $add1_{\text{bt}}$ misbehaves, mirroring the guards in the target-level outcall stub $add1_{\text{comp}}$.

Since 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 (1) will indeed hold, as we set out to illustrate at the start of this section.

<table>
<thead>
<tr>
<th>Outcall Stub</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int $add1$((int*) a)</td>
<td>$\text{univ}<em>\text{contr}</em>{\text{int}_0}(x) = \text{true}$</td>
</tr>
<tr>
<td>2 //@pre m: a $\rightarrow_{\text{int}} [a_1]$</td>
<td>$\text{univ}<em>\text{contr}</em>{\text{int}_0}(x) = \text{true}$</td>
</tr>
<tr>
<td>3 //@post n: a $\rightarrow_{\text{int}} [a_1]$ + result $== a_1 + 1$ {</td>
<td>$\text{univ}<em>\text{contr}</em>{\text{int}_0}(x) = (x !=(\text{null}, 0))$ ?</td>
</tr>
<tr>
<td>4 (int*,int) m; m = (a,1);</td>
<td>$(n: [x.1 + i \rightarrow_{\text{int}} I[I] \mid 0 \leq i &lt; \text{length}(I)]$</td>
</tr>
<tr>
<td>5 int* a$\text{pre}$; int a$\text{post}$</td>
<td>$\ast \text{length}(I) := x.2)$</td>
</tr>
<tr>
<td>6 a$\text{pre} = m.1$; a$\text{post} = m.2[0]$;</td>
<td></td>
</tr>
<tr>
<td>7 //@flatten m</td>
<td></td>
</tr>
<tr>
<td>8 int result; (int*,int) n;</td>
<td></td>
</tr>
<tr>
<td>9 (result,n) = $add1_{\text{bt}}$((a,m));</td>
<td></td>
</tr>
<tr>
<td>10 guard(n != (null,0)); guard(n.2 == 1);</td>
<td></td>
</tr>
<tr>
<td>11 //@collect $\text{bt}$chunk;</td>
<td></td>
</tr>
<tr>
<td>12 int$\text{post}$; int a$\text{post}$</td>
<td></td>
</tr>
<tr>
<td>13 a$\text{post} = n.1$; a$\text{post} = n.2[0]$;</td>
<td></td>
</tr>
<tr>
<td>14 guard(result $== a_1 + 1$);</td>
<td></td>
</tr>
<tr>
<td>15 guard(a$\text{post} = a$); guard(a$\text{post} = a_1$)</td>
<td></td>
</tr>
<tr>
<td>16 return (result,n) }</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (int,int) $add1_{\text{comp}}$((int*,int* m)</td>
</tr>
<tr>
<td>2 int* a$\text{pre}$; int a$\text{post}$;</td>
</tr>
<tr>
<td>3 a$\text{pre} = addr(m)$; a$\text{post} = m[0]$;</td>
</tr>
<tr>
<td>4 int result; int* n;</td>
</tr>
<tr>
<td>5 (result,n) = $add1$(a,m);</td>
</tr>
<tr>
<td>6 guard(n !=null); guard(length(n) $== 1$);</td>
</tr>
<tr>
<td>7 int* a$\text{post}$, int a$\text{post}$</td>
</tr>
<tr>
<td>8 a$\text{post} = addr(n)$; a$\text{post} = n[0]$;</td>
</tr>
<tr>
<td>9 guard(result $== a_1 + 1$);</td>
</tr>
<tr>
<td>10 guard(a$\text{post} = a$); guard(a$\text{post} = a_1$)</td>
</tr>
<tr>
<td>11 return (result,n) }</td>
</tr>
</tbody>
</table>

Fig. 12. Illustrating example: back-translating a context that implements $add1$.

7 DISCUSSION AND FUTURE WORK

This section provides more detail on two challenges in making our compiler more broadly applicable.

7.1 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 efforts are hence still required by third-party developers to produce code that follows our target-level calling convention. Two scenarios are possible.
We now discuss some ideas for extensions in these three directions in order. We do believe all valuable in practice, as many large codebases contain small, isolated components whose security that the authority to deallocate a block of memory needs to be represented separately from the free by relying on a notion of capability range resources. We believe our approach can be extended to a more general form of predicates, suggested extensions to be within reach.

In this paper, we support only two kinds of spatial predicates, describing array and resources. In particular, the source language had a simple type system and featured restrictions on the form of boundary contracts. In the introduction, this paper contains but the first steps towards a practically applicable secure code 

7.2 Extending the Source Language

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.

Secondly, 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 password manager with plugins as 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 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 [Berdine et al. 2007; Distefano et al. 2006], Infer [Calcagno et al. 2015] or SL/Ayer [Berdine et al. 2011], or on a kind of universal contract for unverified code in terms of a pure predicate similar to the lowlevel 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.

7.2 Extending the Source Language

A second direction we want to extend our work in, is to extend the compiler itself. As mentioned in the introduction, this paper contains but the first steps towards 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, i.e., 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

Fig. 13. Separation logic proof of the body of \texttt{addHt} from the the back-translation example
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. We believe the type system will scale in parallel with the introduction of resources to represent more complex permissions, e.g. 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.

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 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 [Chisnall et al. 2015; Watson et al. 2015], and the SAFE machine developed within the CRASH/SAFE project [de Amorim et al. 2016, 2015]. 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. 2018].

To formalize secure compilation, we use the property of fully abstract compilation [Abadi 1999], like many previous results [e.g., Abadi and Plotkin 2012; Devriese et al. 2016; Fournet et al. 2013; New et al. 2016; Patrignani et al. 2015; Skorstengaard et al. 2019]. We refer to Patrignani et al. [2019] for an overview of the field. Recent research has investigated other formal characterisations of secure compilation: robust safety preservation [Patrignani and Garg 2018; Swasey et al. 2017], trace-preserving compilation [Patrignani and 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 (RrHP). Technically, our backtranslation and its use of universal contracts is reminiscent of the use of universal types and universal embeddings in previous work [Devriese et al. 2016; New 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 has 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].
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 runtime 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.

9 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. Thomas Van Strydonck holds a PhD Fellowship of the Research Foundation - Flanders (FWO).

REFERENCES


