Secure Cooperation of Untrusted Components Using a Strongly Typed Virtual Machine

Roland Wismüller and Damian Ludwig
University of Siegen, Germany
E-Mail: {roland.wismueller, damian.ludwig}@uni-siegen.de

***Abstract*—A growing number of computing systems, e.g., smart phones or web applications, allow to compose their software of components from untrusted sources. For security reasons, such a system should grant a component just the permissions it really requires, which implies that permissions must be sufficiently fine-grained. This leads to two questions: How to know and to specify the required permissions, and how to enforce access control in a flexible and efficient way? We present the design and implementation of a novel approach based on the object-capability paradigm with access control at the level of individual methods, which exploits two fundamental ideas: we simply use a component's published interface as a specification of its required permissions, and extend interfaces with optional methods, allowing to specify permissions that are not strictly necessary, but desired for a better service level. These ideas have been realized within a static type system, where interfaces specify both the availability of methods, as well as the permission to use them. In addition, we support deep attenuation of rights with automatic creation of membranes, where necessary. Thus, our access control mechanisms are easy to use and also efficient, since in most cases permissions can be checked when the component is deployed, rather than at run-time. Based on our type system, we have defined a secure intermediate representation, specified its semantics and sketched a correctness proof. The presented concepts have been implemented in a virtual machine called COSMA. When a component is loaded, COSMA type checks its intermediate representation and then compiles it into native machine code, thus enabling its execution with minimal run-time overhead. Thus, COSMA enables the secure, efficient, and flexible cooperation of untrusted software components.**

Keywords–Security; software component; type system; object-capability model; membrane; virtual machine.

I. Introduction

In today's computer based systems, the software environment is often composed of components developed by an open community. Prominent examples are web applications, and smart phones with their app stores. A major problem in such systems is the fact that the components' origin and therefore the components themselves may not be trusted [1] [2]. In order to ensure security in systems composed of untrusted components, the *Principle Of Least Authority* (POLA) should be obeyed, i.e., each component should receive just the permissions it needs to fulfill its intended purpose [3]. The term 'authority' denotes the effects, which a subject can cause. These effects can be restricted via permissions, which control the subject's ability to perform actions. An appealing and popular approach to implement POLA is the use of the object-capability model [4] [5], where unforgeable object references are used as a capability allowing to use the referenced object.

Based on the object-capability paradigm, several secure languages have been devised, such as the E language [4], Joe-E [6] or Emily [7]. In order to allow a fine grained access control at the level of individual methods, the programmer has to manually implement security-enforcing abstractions, e.g., membranes [5]. An inherent problem of language based approaches is that security is achieved mainly by restrictions of the source language and associated compile-time checks. Thus, they not only confine interactions between different components, but also limit the programmer's capabilities within a single component. A second drawback is that security can only be guaranteed, if all components are available in the form of source code, which in practice is infeasible for reasons of protecting intellectual property rights.

The second problem can be addressed by enforcing security at the level of a Virtual Machine (VM). However, existing VMs like Oviedo3 [8] only provide basic mechanisms for the management of access rights, i.e., adding and removing the permission to execute a method for a given object reference, and must check all these permissions at run-time. Thus, they are neither easy to use nor efficient.

To overcome the drawbacks of existing approaches, the goal of our work is to provide a VM that eliminates the shortcomings of existing capability systems and secure high-level languages, and addresses the special needs for the secure cooperation of untrusted components. In particular, it

- allows components to be distributed and deployed in binary form while still providing security,
- enables fine-grained access control without putting a relevant annotation or implementation burden on the components' programmers,
- does not restrict the code's expressiveness within a single component, and
- minimizes the number of required run-time checks by performing most checks when a component is deployed.

Our paper is organized as follows: We start with a discussion of related work in Section II, before in Section III, we present our security model and a component model, where each component specifies its minimal and desired permissions in a natural way using interfaces. We then outline the details of a type system that allows fine-grained access restrictions and optional methods (Section IV). In addition, we introduce the concepts and the implementation of a virtual machine based on a secure, strongly-typed byte code, which allows static type checking at deployment time and the automatic creation of membranes (Section V). We conclude the paper by giving an outlook to our future work (Section VI). The appendix contains the definition of the formal semantics of the virtual machine's instruction set and sketches a proof of its primary security property. This paper is an extended version of [1], which includes an elaborate discussion of our type system, the

automatic creation of membranes and the implementation of our virtual machine.

II. Related Work

Component-based software development has a long tradition in software engineering. Large software systems are built by composing smaller parts, which interact only via well-defined interfaces. Recent research on composition aspects focuses on how to specify the semantics of these interfaces, such that properties of the system can be derived from the properties of the individual components [9] [10] [11]. While typically components are assumed to be trustworthy, secure cooperation of untrusted components is gaining more and more attention, e.g., in the area of telecommunication systems [12] or web application [2]. In many cases, the security is based on the object-capability model.

Capability based protection mechanisms date back as early as the 1960-ies with, e.g., the IBM System/38 and the Hydra operating system as prominent examples [13]. A good introduction to the object-capability model and the principle of least authority is provided in [14]. The general properties of capability systems, as well as some common misconceptions about capabilities are clearly pointed out in [15]. The authors show that capabilities have strong advantages over access control lists and can support both confinement and revocation of access rights. More details about the object-capability paradigm as well as revocation and confinement are discussed in [3]. A detailed discussion of several common object-capability patterns, including membranes, together with a formal modeling and proofs has been presented by Murray [5].

The object-capability paradigm has been used as a basis for secure languages. A pioneer in this area is the work of Mark Samuel Miller [4] on the E language, which points out the prerequisites for secure languages: memory safety, object encapsulation, no ambient authority, no static mutable state, and an API without security leaks. In addition to these features, E provides capabilities for access control at the object level. Access control at the granularity of methods is also possible, but requires the programmer to explicitly implement security-enforcing abstractions like membranes. Based on E, Joe-E [6] restricts Java such that access to objects is only possible via capabilities that have been explicitly passed to a component. It also supports immutable interfaces allowing to implement secure plug-ins. Joe-E uses compile-time checking and secure libraries to disable insecure features of Java like, e.g., reflection and ambient authority. In a similar spirit, Emily [7] is a secure subset of OCaml. Emily does not have the notion of components, but dynamically grants permissions to launched applications via explicit use of the powerbox pattern. Maffeis et al. [2] specifically address the problem of mutual isolation of (third-party) web applications written in JavaScript. They define a language to be *authority safe*, if it satisfies the properties "only connectivity begets connectivity" and "no authority amplification", and formally show that authority safety implies isolation.

An implementation of capability-based access control at method granularity within a virtual machine is presented in [8] [16] [17]. Oviedo3 consists of an object oriented abstract machine and an accompanying operating system. It provides basic mechanisms for the management of access rights, i.e., adding and removing the permission to execute a method for a given object reference, and checks all these permissions at run-time.

A different area of research examines the use of type systems for enforcing security constraints. One goal that has been achieved is alias control, i.e., a means to restrict the sharing of references. Several different concepts have been developed, e.g., universes [18], ownership types [19] [20] [21] or confined types [22]. In [23], Philip Fong shows that these concepts can be used at the byte code level, enabling to enforce confined types at link time, while in [24] [25] the same author discusses how to formally model capabilities, such that confinement can be guaranteed. The common idea of these approaches is to augment class types, such that references to instances of these classes created in some context x cannot be passed to another context y.

There are also several proposals to assign more powerful and flexible security restrictions to types. As an example, both [26] and [27] allow the type system to enforce restrictions on information flow, similar to the Bell-LaPadula model. A powerful, but also complex capability type system is introduced in [28], which allows the programmer to define sequences of aliasing events that may occur to a reference type.

In [29], the authors present the idea of adding hidden capabilities to the interface description of components in order to separate protection definition from application code. Capabilities are implemented in the classical way using OS protection mechanisms. An approach to define the semantics of common type annotations used to specify access rights for, e.g., reading and writing objects is outlined in [30]. While the code can be checked statically in this approach, it does not support fine-grained access permissions for methods. Strategies for fine-grained access control with link-time enforcement have been developed in IsoMod [31]. Based on the idea that if a loaded module does not know the name of an entry point, it cannot call the corresponding service, IsoMod defines a powerful, but also rather complex policy definition language. A problem, which is inherited from the underlying Java VM is that there are no explicit interfaces between the modules, making it extremely hard to determine the minimal rights required by a module. In addition, only nominal typing is supported and rights cannot be attenuated at run-time. The Safe Language Kernel [32] employs a similar basic idea by introducing type capabilities, which are checked at link time. However, since the paper was never officially published the concept is not fully worked out and has not been integrated into a type system.

The above discussion shows that although there are some partial solutions for supporting the secure cooperation of untrusted components, there is no satisfying practical approach for this problem yet. The language-based approaches achieve security by imposing restrictions on the programming language, which also affects its expressiveness for programming within a single component. In addition, they require components to be distributed as source code, which is not desirable. Approaches based on the object-capability paradigm or on type systems for alias control in general just support access control at the object-level, but not at the method-level. Fine-grained access control either requires manually programmed security

abstractions or a run-time management and enforcement of method permissions implying a significant overhead. The more sophisticated type based approaches, like [28] result in a significant annotation burden for the programmer, which hinders their widespread usage.

The contribution of our work is to provide an approach that avoids the mentioned shortcomings by combining existing techniques with completely new ideas. Our solution is based on a VM executing a strongly typed intermediate language where types are used to represent permissions. A central idea is to specify fine-grained access control by just using standard interface definitions, thus avoiding complex code annotations. Since the VM performs a static type check, which in our approach also corresponds to a permission check, when a new component is loaded, most run-time checks can be avoided. If necessary, the VM automatically builds the required membranes for method-level access restrictions. The VM's underlying type system has been designed in such a way that it does not limit expressiveness within a component, but just restricts the permissions of object references passed from one component to another. While we did not implement annotations for alias control yet, this may easily be added in future.

III. Component and Security Model

A. Component Model

Our work is based on the established definition of a software component, as given by Szyperski: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*" [33]. We assume that components are distributed as compiled byte code for a VM, rather than source code. Their internal structure is not relevant, however, we require that a component defines a purely object oriented interface, i.e., to its environment it appears to be composed of classes. One of these classes, the *principal class*, is the component's entry point, i.e., when the component is loaded, an instance of the principal class is created and its constructor is invoked.

B. Security Model

In order to support qualified statements about security properties, we structure the computing system under consideration into a disjoint set of security domains that we call *contexts*. We assume that the system features a trusted computing base, which just consists of the VM and a minimal operating system (OS) kernel, and is referred to as OS context. Any other service available in the system is implemented by loading components from an external, untrusted source. When the OS loads a component, it creates a new context c and instantiates the component's principal class within this context. In turn, when a method of any object in context c instantiates a new object, the new object will also be enclosed in c. Thus, c comprises all objects created on behalf of the loaded component.

Our threat model now is as follows: Code executing within a context c may try to compromise the confidentiality, integrity, authenticity and availability of any data contained in or service provided by a context c' by accessing or invoking data or code in other contexts, including the OS context.

As a prerequisite, we assume that all interactions between different contexts are based on the object-capability paradigm, i.e., access to data or code is possible only via references that can be passed between contexts and act as capabilities. In addition, data can be accessed only by calling an object's methods. This implies that there is no ambient authority, thus, the OS kernel must also exhibit a purely object oriented interface.

The goal of our work now is to parry the depicted threat by enforcing the following property: Code of a component that has been loaded into a context c can only perform actions (i.e., execute methods on objects) in contexts different from c, which (1) have been explicitly documented by that component, and (2) have been explicitly permitted to context c.

Compared to the traditional object-capability model, there are two significant extensions. First, the granularity of access control is more fine grained, since our model controls the ability to execute certain actions on an object, rather than just the overall access to the object. Second, components include an explicit definition of their required permissions.

Listing 1. Calendar interface

```
principal class Calendar {
  interface Appointment {
    int startTime();
    int endTime();
    String subject();
    String notes();  // confidential notes
  }
  void createAppointment(...) { ... }
  // return next upcoming appointment
  Appointment getNextAppointment() { ... }
  ...
}
```

C. An Example

Consider a component `Calendar` that manages and provides appointments. In a Java-like language (where access modifiers have been omitted for brevity), this component may be defined as shown in Listing 1.

Now assume that there is a component C whose task is just to display the next upcoming appointment in some widget. In this situation, POLA requires that C can just get the next appointment, but not, e.g., create a new one. However, the standard object-capability model will allow C to invoke `createAppointment()` once it receives a reference to a `Calendar` object. This is true, even when the reference is passed using a restricted type (i.e., an interface type just containing `getNextAppointment()`), since virtually all popular object oriented languages will allow C to downcast the reference to the type `Calendar` again.

D. Fine Grained Access Control

In principle, an object-capability system can be extended with a more fine grained access control by supplementing

Let T_{pc} denote the type of the principal class of C,
$\mathcal{T}_{\mathrm{arg}}(T)$ the set of the argument types of all methods defined in a type T,
$\mathcal{T}_{\mathrm{res}}(T)$ the set of the result types of all methods defined in type T,
$\mathcal{T}_{\mathrm{in}}$ the set of all types of references (or values) that component C receives from its environment (required interface), and
$\mathcal{T}_{\mathrm{out}}$ the set of all types of references (or values) passed from component C to its environment (provided interface).

Then $(\mathcal{T}_{\mathrm{in}}, \mathcal{T}_{\mathrm{out}})$ is the least fixed point of the following equations:

$$\begin{aligned} \mathcal{T}_{\mathrm{in}} &= \mathcal{T}_{\mathrm{arg}}(T_{\mathrm{pc}}) \cup \mathcal{T}_{\mathrm{arg}}(\mathcal{T}_{\mathrm{out}}) \cup \mathcal{T}_{\mathrm{res}}(\mathcal{T}_{\mathrm{in}}) \\ \mathcal{T}_{\mathrm{out}} &= \{T_{\mathrm{pc}}\} \cup \mathcal{T}_{\mathrm{res}}(\mathcal{T}_{\mathrm{out}}) \cup \mathcal{T}_{\mathrm{arg}}(\mathcal{T}_{\mathrm{in}}) \end{aligned}$$

Figure 1. Computation of the provided and required interfaces of a component C.

each reference with a list of permissions, which allow or disallow the available methods. However, this approach, which is, e.g., implemented in Oviedo3 [8] [17] imposes a significant overhead both for storing the access rights in each reference and for checking them at run-time whenever the reference is used. In addition, it does not directly allow to restrict the permissions for references returned by calls to a permitted method. In our example, permissions included in the `Calendar` reference passed to C can prevent C from calling `createAppointment()`, but not from calling `notes()` on the appointment returned by `getNextAppointment()`.

A better solution is to use the membrane pattern [4] [5]: Instead of providing C with a reference to the real calendar, we create a membrane object, which acts similar to a proxy in the sense that it delegates method calls to the calendar object. The important difference is that the membrane will only provide a `getNextAppointment()` method. In addition, it can also wrap the returned appointment into another membrane that does not provide the `notes()` method. Thus, membranes can also support deep attenuation of rights.

However, the classical membrane pattern has two severe drawbacks: First, the manual creation of membranes to enforce a minimal set of permissions is a difficult and error prone task for large object systems, since the necessary membrane structure may be deeply nested or may even be recursive. Second, if permissions are attenuated in several steps (e.g., component A passes an attenuated version of the calendar to B, which passes a more attenuated version to C), membranes will be cascaded, thus sacrificing run-time performance due to multiple delegation. As we will show, both drawbacks can be avoided by automatically generating membranes, when necessary.

E. Specifying Required Permissions

Another general problem related to POLA is that the user of a component C must know the minimal permissions required by C in order to work properly. One of the central ideas of our work is to use the already available type definition of an object reference as a specification of the permissions that are requested (in the case of an input variable) or granted (in the case of an output variable) by this reference. This perception of type definitions is possible when the run-time system executing the code of a component does not allow any of its input references to be downcasted to a less restrictive type.

Now, given our purely object oriented component model, we can exactly determine the minimal permissions that a component C requires from its environment by determining the types of all references that C can receive. Vice versa, we also can identify the permissions that C grants to its environment from the types of all references that C returns. In order to increase the model's flexibility, we also allow optional methods (i.e., permissions) in interfaces. In this way, the type of a component C's principal class explicitly defines

- $\mathcal{T}_{\mathrm{in}}$: the minimum and maximum permissions that C requests from its environment, where C will use optional methods, if they are available, but does not require them for its correct operation, and
- $\mathcal{T}_{\mathrm{out}}$: the minimum and maximum permissions that C grants to its environment, where for each optional method, C may decide at run-time whether or not to provide it.

The set $\mathcal{T}_{\mathrm{in}}$ ($\mathcal{T}_{\mathrm{out}}$) is determined by recursively computing the types of all references that the component can receive from (pass to) its environment, as shown in Figure 1.

As an example, consider the calendar component in Listing 1. As the component has no input (we omitted the parameters of `createAppointment()` for simplicity), `Calendar` does not request any permissions from its environment, so $\mathcal{T}_{\mathrm{in}} = \emptyset$. However, it grants permission to use the appointment returned by `getNextAppointment()` via the `Appointment` interface, which results in $\mathcal{T}_{\mathrm{out}} = \{$`Calendar`, `Appointment`, `int`, `String`$\}$.

The calendar client displaying upcoming appointments may now have an interface similar to Listing 2.

This interface specifies the permissions the client needs from a `Provider`: it must be able to call the `getNextAppointment()` method, which returns an object of type `Event`. On an `Event`, the client must be able to call `startTime()` and `endTime()`, and it will use `subject()`, if available. Thus, for the calendar client component we have $\mathcal{T}_{\mathrm{out}} = \{$`CalendarClient`$\}$ and $\mathcal{T}_{\mathrm{in}} = \{$`Provider`, `Event`$\}$. Since we use structural typing for component interfaces, a reference to the `Calendar` component can be passed to `setProvider()`, as `Appointment` provides all the methods required by `Event`.

LISTING 2. CALENDAR CLIENT INTERFACE

```
principal class CalendarClient {
  interface Provider {
    Event getNextAppointment();
  }
  interface Event {
    int startTime();
    int endTime();
    optional String subject();
  }
  void displayEvents();
  void setProvider(Provider c);
}
```

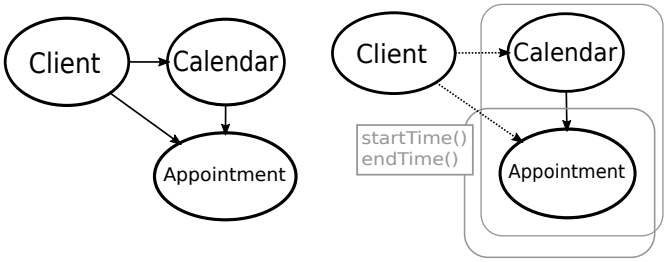

Figure 2. Full access to `Appointment` (left) versus restricted permissions (right): the client has access to `Calendar` only through a membrane. The membrane's `getNextAppointment()` method in turn returns a membrane for the `Appointment` object, which only allows two methods to be called.

In this example, the calendar client will not be able to call the `notes()` method on events received from any `Provider`, because it is not part of the `Event` interface. Formally, a component C can invoke method m on an object o from another component, only if o can be assigned to a reference of some type $T \in \mathcal{T}_{\text{in}}$, which allows to call m. Especially, a component can only execute the operations explicitly specified in its published interface. This means that everything the component can do is explicitly visible in its published interface, so the user can decide not to install the component or to only provide it with a reference to a restricted object where (some of) the optional permissions are not granted. Traditionally, this requires to manually program a membrane for the `Calendar` component, such that the object returned by `getNextAppointment()` does not have a `subject()` method (see Figure 2). In our model, the same effect can be achieved by just casting the `Calendar` reference to a more restricted interface, where the `subject()` method is missing.

In principle, if the `Calendar` component declared the `subject()` method in `Appointment` as optional, it also could decide at run-time whether or not to expose this method to the client invoking `getNextAppointment()`, e.g., based on some authentication procedure. However, we believe that this decision should usually be left to the user assembling the components.

Note that a component's published interface (what it pretends to do) may differ from its actually implemented interface, e.g., a component may try to call a method not declared in its published interface. However, because the component will always be *used* via its published interface, such a deviation will result in a type error when the component is loaded. We will present a detailed discussion of our type system in the next section.

TABLE I. INTENDED SEMANTICS OF A TYPE T.

Status of method m in type T	Assertion that the referenced object has method m	Permission to call method m
m is not in T	no	no
m is optional in T	no	yes
m is required in T	yes	yes

IV. TYPE SYSTEM

In order to ease the presentation of our type system, we make a few simplifying assumptions for the following discussion: First, we assume that components are written in a statically typed, object oriented language. This will usually be the case today and is also assumed by our current implementation. However, the only real requirement of our approach is just that the interface between a component and its environment is purely object oriented. Second, the presentation only covers object types, i.e., class and interface types, although our implementation of the type system also provides simple types and array types. Finally, we will assume the use of structural typing for interface types. An extension allowing a flexible mixture of structural and nominal typing is currently being developed (see Section VI).

A. Types as Permissions

A central idea of our type system is to interpret a component's type as a specification of access permissions at the granularity of single methods. In addition, we retain the traditional interpretation, which asserts that all objects of a given type will offer the methods specified by that type. We achieve both goals by using the concept of optional methods, as specified in Table I.

As the main goal of our type system is security, it must enforce the access restrictions defined by Table I in such a way that no component can amplify its rights by type conversions, i.e., downcasting. Whenever possible, we ensure this property statically, i.e., at the time a component is deployed, rather than by using run-time checks. In addition, we avoid delayed type failures: once a component C is deployed and a reference to C's primary object has successfully been assigned to a variable of some type I, all methods in I can be invoked without run-time type errors. Finally, the type system supports an easy attenuation of rights by just upcasting a reference, without the need to manually code a membrane.

B. Subtyping Rules

From Table I, we can immediately derive the subtype rules required for our type system: if we have two types S and T, which only differ in the status of a method m, then S is subtype of T, if and only if

- S contains m, but T does not (classical situation),

$$\text{S-REFL}\ \frac{}{S <: S} \qquad \text{S-TRANS}\ \frac{S <: U \qquad U <: T}{S <: T}$$

$$\text{S-RCD}\ \frac{\{l_i\ ^{i\in 1..n}\} \subseteq \{k_j\ ^{j\in 1..m}\} \quad k_j = l_i\ \Rightarrow\ (n_j \Rightarrow o_i) \wedge (S_j <: T_i)}{\{(k_j : S_j,\, n_j)\ ^{j\in 1..m}\} <: \{(l_i : T_i,\, o_i)\ ^{i\in 1..n}\}}$$

$$\text{S-ARROW}\ \frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

Figure 3. Subtyping rules

$$\text{CC-REFL}\ \frac{}{S \prec:_c S} \qquad \text{CC-TRANS}\ \frac{S \prec:_c U \qquad U \prec:_c T}{S \prec:_c T}$$

$$\text{CC-RCD}\ \frac{\{l_i\ ^{i\in 1..n} \mid \overline{o_i}\} \subseteq \{k_j\ ^{j\in 1..m}\} \quad k_j = l_i\ \Rightarrow\ (n_j \Rightarrow o_i) \wedge (S_j \prec:_c T_i)}{\{(k_j : S_j,\, n_j)\ ^{j\in 1..m}\} \prec:_c \{(l_i : T_i,\, o_i)\ ^{i\in 1..n}\}}$$

$$\text{CC-ARROW}\ \frac{T_1 \prec:_c S_1 \qquad S_2 \prec:_c T_2}{S_1 \to S_2 \prec:_c T_1 \to T_2}$$

Figure 4. Defining rules for $\prec:_c$. If $S \prec:_c T$, then $r : S$ can be assigned to $r' : T$ without the need for a type check at run-time.

$$\text{CM-REFL}\ \frac{}{S \prec:_m S} \qquad \text{CM-TRANS}\ \frac{S \prec:_m U \qquad U \prec:_m T}{S \prec:_m T}$$

$$\text{CM-RCD}\ \frac{\{l_i\ ^{i\in 1..n}\} \subseteq \{k_j\ ^{j\in 1..m}\} \quad k_j = l_i\ \Rightarrow\ S_j \prec:_m T_i}{\{(k_j : S_j,\, n_j)\ ^{j\in 1..m}\} \prec:_m \{(l_i : T_i,\, o_i)\ ^{i\in 1..n}\}}$$

$$\text{CM-ARROW}\ \frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 \prec:_m T_1 \to T_2}$$

Figure 5. Defining rules for $\prec:_m$. If $S \prec:_m T$, then $r : S$ can be assigned to $r' : T$ without the need to introduce a membrane.

$$\text{C-REFL}\ \frac{}{S \prec: S} \qquad \text{C-TRANS}\ \frac{S \prec: U \qquad U \prec: T}{S \prec: T}$$

$$\text{C-RCD}\ \frac{\{l_i\ ^{i\in 1..n} \mid \overline{o_i}\} \subseteq \{k_j\ ^{j\in 1..m}\} \quad k_j = l_i\ \Rightarrow\ S_j \prec: T_i}{\{(k_j : S_j,\, n_j)\ ^{j\in 1..m}\} \prec: \{(l_i : T_i,\, o_i)\ ^{i\in 1..n}\}}$$

$$\text{C-ARROW}\ \frac{T_1 \prec:_c S_1 \qquad S_2 \prec:_c T_2}{S_1 \to S_2 \prec: T_1 \to T_2}$$

Figure 6. Defining rules for $\prec:$. If $S \prec: T$, then $r : S$ can be assigned to $r' : T$.

- or m is required in S, but optional in T (since this means that every object implementing S also implements T).

The formal subtyping rules are shown in Figure 3. We model classes and interfaces as record types, whose members are functions. Functions have just one argument and return type, however, multiple argument and/or result values are possible, since the type can again be a record. The notation $\{(l_i : T_i,\, o_i)\ ^{i\in 1..n}\}$ denotes a type with n methods named l_i modeled as functions of type $T_i = T_i' \to T_i''$, where the Boolean value o_i indicates whether or not the method is optional. Compared to traditional type systems (see, e.g., [34]), the rule S-RCD representing structural subtyping is modified with an extension for optional methods: assume that $S = \{(k_j : S_j,\, n_j)\ ^{j\in 1..m}\}$ and $T = \{(l_i : T_i,\, o_i)\ ^{i\in 1..n}\}$, then for $S <: T$ we additionally require that for all common members m of S and T either m is not optional in S, or it is also optional in T.

C. Well-Typed Programs

For safety and security reasons, we may allow the VM to load a component only if the component's code is *well-typed*. According to Cardelli [35], this means that the code will not exhibit any unchecked run-time errors (although controlled exceptions are allowed). The main question in this context is: when can we allow to assign a reference from a variable r of type S to a variable r' of type T? Compared to traditional type systems, the important restriction here is that we must not allow r' to gain more permissions than r via downcasting.

Assume that there exists a method m that is optional in S, but required in T. Table I shows that there are no security concerns in this situation, since both S and T allow to call m. However, since T asserts that the referenced object has method m, we must check this condition at run-time when assigning r to r'. Thus, we can assign $r : S$ to $r' : T$ *without* a run-time type check, if and only if

- there is no optional method in S that is required in T,
- each required method of T is also present in S,
- each method of S can be assigned to its corresponding method in T without a run-time check, i.e., all its arguments and results can be assigned without check (this avoids delayed type failures).

Using the rules shown in Figure 4, we denote this situation as $S \prec:_c T$.

A different situation arises if there exists a method m that is optional in T, but is not present in S. In this case, Table I shows that T actually allows to call m (which may, however, result in a run-time error, if the referenced object o does not provide that method), while S does not. Thus, we actually can assign $r : S$ to $r' : T$ in a secure way, if after this assignment r' references an object that does *not* provide m. We ensure this by using a coercion semantics, where the result of the assignment is a reference to a membrane for o that does not provide method m. Vice versa, this means that we can assign $r : S$ to $r' : T$ *without* introducing a membrane, if and only if

- each method of T is also declared in S, and

- all methods of S can be assigned to the corresponding method of T without a need for a membrane.

This is formalized in Figure 5. Note the rule CM-ARROW, which states that when we assign a method $m : S_1 \to S_2$ to a method $m' : T_1 \to T_2$, and must perform a run-time check or introduce a membrane for the method's argument or its result (i.e., $T_1 \not<: S_1$ or $S_2 \not<: T_2$), we need to wrap m with some code performing these tasks when it is called at run-time. This is done by introducing a membrane for the object providing m.

We can summarize the above considerations into a single relation $\prec:$ defined by the rules in Figure 6. $S \prec: T$ denotes that an assignment from $r : S$ to $r' : T$ is (statically) type safe, if and only if

- each required method of T is also declared in S, and
- all methods of S can be assigned to the corresponding method of T.

The latter is only the case, if the argument and result can be assigned *without* a run-time type check. This is a deliberate decision in order to avoid delayed type failures, as mentioned in the second paragraph of Section IV-A.

D. Run-Time Actions

With the relations introduced above, a component's code is well-typed, if for each assignment from $r : S$ to $r' : T$ the condition $S \prec: T$ holds. However, when at run-time an assignment with $S \not\prec:_c T$ is about to execute, we need to perform an additional type check. Likewise, when $S \not\prec:_m T$, we need to introduce a membrane. If we must both create a membrane and perform a run-time check, the run-time check will be the last action. For the following detailed discussion, we assume that variable $r : S$ contains a reference to an object o of class C.

During the run-time type check, we must simply verify that a reference to o can be assigned to r' without any further actions, i.e., $C <: T$.

Introducing a membrane is more complex. The coercion semantics requires that after $r : S$ has been assigned to $r' : T$, r' refers to an object (i.e., the membrane) that actually has type T. This membrane must be computed from the types S and T, and the class C of the object o referenced by r. We perform this computation in two steps: First, from S and T we compute the *cast action* a that needs to be performed by the membrane. This can be done at the component's load-time. Second, we apply a to the actual object o at run-time in order to obtain the concrete membrane. The rules to compute the cast action are shown in Figure 7. Informally speaking, the cast action instructs the membrane to (recursively) retain only those methods that are allowed by both S and T. Thus, the application of a cast action a to an object o results in a membrane for o that (recursively) provides just these methods. This is formalized in Figure 8, where f_a is the actual run-time operation performed for the cast action a.

A special case occurs, if o already is a membrane (with cast action a') for some other object o'. In this case, the membranes will not be cascaded, but applying a to o will result in a merged membrane for o', reflecting both cast actions a and a', as shown in Figure 9. Informally, the merge operation corresponds to the (recursive) intersection of the allowed methods.

We define the set $\mathcal{CA}$ of cast actions inductively as follows:

1) $\top \in \mathcal{CA}$
   $\top$ denotes that the membrane does not need to perform any action, i.e., no need for a membrane.
2) $\forall_{i=1..n} : a_i \in \mathcal{CA} \;\Rightarrow\; \{l_i : a_i \;^{i=1..n}\} \in \mathcal{CA}$
   This means that the membrane (just) provides the methods l_i that perform the actions defined by a_i.
3) $a \in \mathcal{CA} \;\wedge\; b \in \mathcal{CA} \;\Rightarrow\; (a, b) \in \mathcal{CA}$
   This denotes that a method first applies the action a to its argument, then forwards the call to the real object, and finally applies the action b to the result.

For two types S and T, the cast actions $ca(S, T)$ that need to be performed when assigning $r : S$ to $r' : T$ are defined by the following rules:

$$\text{CA-NONE}\ \frac{S \prec:_m T}{ca(S,T) = \top}$$

$$\text{CA-RCD}\ \frac{\begin{array}{c} S = \{(k_j : S_j,\, n_j)\ ^{j\in 1..m}\} \\ T = \{(l_i : T_i,\, o_i)\ ^{i\in 1..n}\} \\ S \prec: T \\ S \not\prec:_m T \end{array}}{ca(S,T) = \{l_i : ca(S_j, T_i)\ ^{i\in 1..n,\ j\in 1..m,\ k_j = l_i}\}}$$

$$\text{CA-ARROW}\ \frac{\begin{array}{c} S = S_1 \to S_2 \\ T = T_1 \to T_2 \\ S \prec: T \\ S \not\prec:_m T \end{array}}{ca(S,T) = (ca(T_1, S_1), ca(S_2, T_2))}$$

Figure 7. Rules for the cast actions specifying the behavior of a membrane.

E. Downcasting

The type system outlined above achieves its security properties by strictly limiting downcast operations. However, this limitation is only necessary when code executing in a context x has a reference to an object in a different context y. If the reference points to an object o in the local context x, i.e., an object created by context x, there are no security issues at all. This is because x is able to retain the original (unrestricted) reference when it creates o, thus, it can not gain additional authority by downcasting any reference to o. So in order not to restrict the expressiveness of our type system, we should allow downcasting in this situation.

We achieve this by a simple extension: We distinguish between *local* types, which assert that references of this type will always point to objects contained in the local context, and *non-local* types, which do not provide such a guarantee. We then extend the subtyping rules from Figure 3, such that $S <: T$ does *not* hold if T is local, but S is non-local. It is still possible to assign from a variable r with a non-local type

$$
\begin{aligned}
f_a(null) &= null \\
f_\top(o) &= o \\
f_{\{k_j:a_j\ ^{j\in 1..m}\}}(\{l_i : m_i\ ^{i\in 1..n}\}) &= \{l_i : f_{a_j}(m_i)\ ^{i\in 1..n,\ j\in 1..m,\ k_j=l_i}\} \\
f_{(a,b)}(m) &= f_b \circ m \circ f_a
\end{aligned}
$$

Figure 8. Semantics of the generated membrane.

Assume that o is a membrane, i.e., $o = f_{\{k'_j:a'_j\ ^{j\in 1..m}\}}(o')$. Then

$$
\begin{aligned}
f_{\{k_j:a_j\ ^{j\in 1..m}\}}(o) &= f_{\{k_j:a_j\ ^{j\in 1..m}\}}(f_{\{k'_j:a'_j\ ^{j\in 1..m}\}}(o')) = (f_{\{k_j:a_j\ ^{j\in 1..m}\}} \circ f_{\{k'_j:a'_j\ ^{j\in 1..m}\}})(o') \\
&= f_{\mathrm{merge}(\{k_j:a_j\ ^{j\in 1..m}\},\{k'_j:a'_j\ ^{j\in 1..m}\})}(o')
\end{aligned}
$$

where the function merge is defined by

$$
\begin{aligned}
\mathrm{merge}(\top, a) &= a \\
\mathrm{merge}(a, \top) &= a \\
\mathrm{merge}(\{l_i : a_i\ ^{i\in 1..n}\}, \{k_j : b_j\ ^{j\in 1..m}\}) &= \{l_i : \mathrm{merge}(a_i, b_j)\ ^{i\in 1..n,\ j\in 1..m,\ k_j=l_i}\} \\
\mathrm{merge}((a_1, b_1), (a_2, b_2)) &= (\mathrm{merge}(a_2, a_1), \mathrm{merge}(b_1, b_2))
\end{aligned}
$$

Figure 9. Fusion of cascaded membranes.

S to a variable r' with local type T, provided that a run-time check is performed to ensure that r actually refers to an object in the local context. With this extension, we can allow all the traditional downcast operations when the source type is local.

F. An Example

Assume that we have the types `Calendar` and `Appointment` from Section III, as well as the three restricted interfaces shown in Listing 3.

LISTING 3. RESTRICTED CALENDAR PROVIDER INTERFACES

```
interface Provider1 {
  Event1 getNextAppointment();
}
interface Event1 {
  int startTime();
  int endTime();
}

interface Provider2 {
  Event2 getNextAppointment();
}
interface Event2 {
  int startTime();
  int endTime();
  optional String subject();
}

interface Provider3 {
  Event3 getNextAppointment();
}
interface Event3 {
  int startTime();
  int endTime();
  String subject();
}
```

In this example, the type `Calendar` can be converted to `Provider1` without any precautions, since `Calendar` $<:$ `Provider1` (Rules S-RCD and S-ARROW in Figure 3). Note that in this case the restricted access permissions, like the fact that it is not allowed to call `subject()` on the object returned by `getNextAppointment()`, are statically enforced by the type `Provider1` without a need for any run-time checks.

A further conversion from `Provider1` to `Provider2` is also allowed, since `Provider1` $\prec:$ `Provider2` (Rule C-RCD in Figure 6). However, we have `Provider1` $\not\prec:_m$ `Provider2` (Rules CM-RCD and CM-ARROW in Figure 5), since the members of `Event2` are a not a subset of the members of `Event1`. Thus, at run-time a membrane for the `Calendar` object is created that wraps the method `getNextAppointment()`, such that its return value is wrapped into a second membrane that does not have a method `subject()` (c.f. Figure 2). Now, the access restrictions are enforced by the membrane.

Our type system does not allow a conversion from `Provider2` to `Provider3`. Although this conversion would be safe if the reference actually points to an object of type `Calendar`, allowing it could result in unexpected run-time errors: A component that has a reference of type `Provider3` would assume that `getNextAppointment()` always returns an object that provides the method `subject()`. However, since `subject()` is declared as optional in `Event2`, the implementation of `getNextAppointment()` in the referenced object may actually return a reference that does not allow to call this method.

V. COSMA

The *Component Oriented Secure Machine Architecture* (COSMA) is a secure VM based on the outlined component model and type system. It comes with a specification for an object oriented byte code, called *Component Intermediate Language*. As outlined in Figure 10, the structure of this byte code

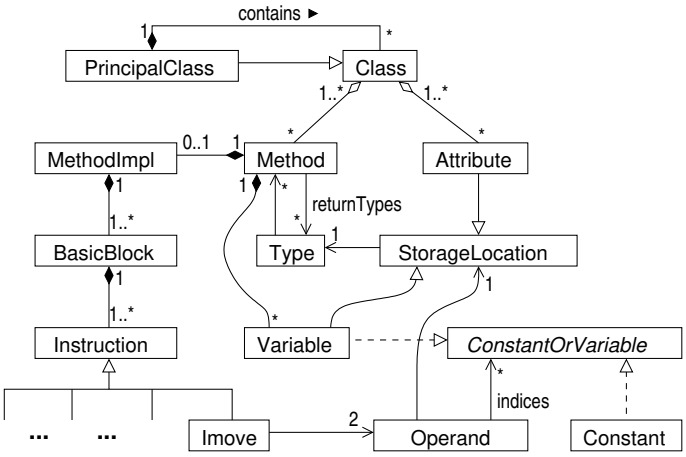

Figure 10. Simplified structure of a COSMA program

reflects that of a component: The entry point for a component's code always is its principal class, which logically contains all other classes. Method implementations are structured into basic blocks, which are sequences of instructions (cf. Figure 11). Basic blocks within the local method are the only admissible targets of branch instructions. Instructions do not allow direct access to the memory. Instead, they use typed operands to access abstract storage locations. There is also no visible call stack, but a high-level method call instruction, where lists of operands are passed for arguments and results. This ensures that a malicious program cannot forge references (e.g., by abusing an untyped stack), which is the major requirement for a secure object-capability system. Since the byte code does not contain any names except the obligatory method names in interfaces, it also protects the component developer's intellectual property rights.

We need a secured byte code, since secure high-level languages "*can still be attacked from below*" [36]. In order to prevent such attacks, we must use "*computers on which only capability-secure programs are allowed*" [36]. Thus, new programs can only be loaded into COSMA as components represented in our byte code.

When a component is deployed into the VM, it is associated with a new context that serves as a trust (or protection) unit. Within this context, the component's principal class is instantiated, and a reference (capability) to this *principal object* is returned and gets casted to the component's published interface. Initially, this reference is the only way to interact with the component. When an object in a context c creates another object, the new object also is associated with c. Thus, a context comprises all objects that are (transitively) created by the principal object of a loaded component. COSMA ensures that references can point to objects in a different context, only if they have a non-local type and therefore are subject to the security restrictions outlined in Section IV. References with local types can only point to objects in the local context. Thus, we do not restrict the code's expressiveness within a component.

During deployment, a component's complete byte code is checked for consistency, which includes type checking. Since the byte code does not allow any untyped data accesses, this

`call r m args res`
: Calls method m on the reference r passing $args$ as arguments and res as write-back operands.

`cjmp src nz block`
: Jumps to $block$ if $src \neq 0$ ($nz = \texttt{true}$) or if $src = 0$ ($nz = \texttt{false}$).

`chktype r t dst`
: Tests whether r has type t and stores the result in dst.

`inv r id args`
: Invokes a method whose name is stored in the string referenced by operand id on the reference r and passes $args$ as the arguments.

`jmp b`
: Jumps to block b.

`load const dst`
: Loads the constant $const$ into dst.

`mov src dst`
: Assigns src to dst.

`new c dst`
: Creates a new object of the given class c and stores the reference in dst.

`op lhs rhs op dst`
: Calculates $lhs\ o_{op}\ rhs$ (with $o_{op} \in \{+, -, *, /, \texttt{mod}, ...\}$) and stores the result in dst.

`ret r`
: Returns the given operands to the caller.

`test lhs rhs op dst`
: Tests for $lhs\ o_{op}\ rhs$ (with $o_{op} \in \{=, \neq, <, \leq, >, \geq\}$) and writes the result to dst.

Figure 11. Instructions supported by the virtual machine (without array-specific instructions).

can be done on a per-instruction basis, without a need for a complex verification of instruction sequences, as it is necessary, e.g., in Java byte-code [37]. Based on the type information available in the component's code, COSMA automatically generates the code for all required membranes, relieving the programmer from this burden. At run-time, membranes are automatically inserted via coercion semantics when security relevant downcasting is performed. Thus, security constraints are enforced mainly statically, leaving only a few run-time checks.

The first component that is deployed into the VM can receive a reference to a *native kernel object* as the first parameter of its constructor. This object offers basic operating system services (system calls), which are not expressible through the instruction set for security reasons. As described in Section III-B, the kernel object is part of the trusted computing base and built into COSMA. At the moment, we have implemented only a few indispensable services, thus, the reference has the interface type shown in Listing 4 (the type `Any` used here is a special type, which can be assigned to any reference type, using a run-time type check; the type `String` is currently implemented as an integer array).

LISTING 4. KERNEL INTERFACE

```
interface Kernel {
  // load component IR and
  // return principal object
  Any loadComponent(String filename);
  // print string to stdout
  void print(String msg);
  // read string from stdin
  String scan();
  // create a thread
  Thread createThread(Runnable r);
}
interface Thread {
  void start();
  void join();
}
interface Runnable {
  void run();
}
```

The initial component can share the reference to the kernel object with other components, while all the rules from Section IV still apply. In particular, utilization of the operating system can be restricted and checked at load time as if it were a normal object.

The basic idea is that the first component, wrapping the functionality of the native kernel object, provides additional operating system services with a cleaner and more powerful interface.

A. Implementation

We have implemented two variants of the COSMA virtual machine: A pure interpreter as a reference implementation of the machine's semantics (see Appendix), and a more realistic version that uses an ahead-of-time (AOT) compiler to translate a component's intermediate representation (IR) to native machine code at load time.

The AOT version of COSMA consists of two processes: One of them implements the loader and type checker, which are programmed in Java, while the other one executes the generated native code of all loaded components. When COSMA starts, the loader reads the IR of the initial component, performs type and consistency checks for each element of the IR, and then compiles the IR into C code, which is then compiled into a shared library using the GNU C compiler. This library is then dynamically loaded and linked into the native execution process using the dynamic linker (i.e., `dlopen()` in Linux systems).

The constructor of the initial component's principal class receives a reference to the kernel object described above. When the kernel's `loadComponent()` method is invoked, a synchronous request is sent to the loader process (using interprocess communication via UNIX pipes), which will then load the component's IR as described above.

When the native code executes an assignment, it may be necessary to perform a run-time type check and/or to introduce a membrane. Since a naïve implementation of these operations would require a recursive traversal of the involved type definitions, which is prohibitively expensive, we follow a different strategy. In the native code, each type is just represented as a globally unique integer number, which is assigned by the loader when it first sees the type. The loader also precomputes the relation $<:$ into a hash table, so that a type-check at run-time just requires a hash table lookup. The hash table is extended incrementally whenever a new component is loaded.

Furthermore, the loader precomputes the information that is needed to create the native code for the classes of all membranes that may be required at run-time. In Section IV-D, we have shown that when a reference r pointing to an object o of class C is converted from type S to T, the required membrane is determined in two steps:

1) compute the cast actions $ca(S,T)$ from S and T,
2) generate the membrane for o from $ca(S,T)$ and C.

In order to generate all membrane classes that may be required at run-time, we execute a simple data flow analysis that determines for each statement s performing a relevant type cast from S to T the set of all classes C, such that a reference to an object of class C may reach s. We then generate the code for the required membrane class from $ca(S,T)$ and C. Since the instances of these membrane classes may again need to be wrapped by a membrane, resulting in new membrane classes, we perform a fixed point iteration, which stops when the set of membrane classes does not change any more. The termination of this fixed point iteration is actually guaranteed by the fact that we avoid cascading of membranes by fusing them as outlined in Fig 9. As with the subtype relation, we incrementally expand the set of generated membrane classes whenever a new component is loaded.

Although the creation of all possibly required membrane classes at load-time requires additional time and storage when a component is loaded, it makes the creation of membranes at run-time extremely fast: We just need a hash table lookup (with the type number of the object's class as the key) to find the membrane class that must be instantiated.

B. Performance Considerations

Since in the majority of cases, the necessary access restrictions are enforced statically by COSMA's type system, we can achieve fine grained access control between components with minimal run-time overhead. This overhead, as compared to traditional designs for object oriented virtual machines, arises from the following three sources:

- The IR does not have a fixed format, like traditional byte codes, and therefore cannot be directly interpreted with comparable efficiently. However, as modern virtual machines are based on just-in-time or ahead-of-time compilation techniques, an efficient direct interpretation of the IR is no longer a necessity. If really required, a simple ahead-of-time compiler could easily transform the IR into a fixed format byte code at load time.
- Method calls on component interfaces require a more expensive dispatch mechanism, since we use a modified form of structural typing instead of nominal

typing. However, there is a number of established methods to minimize this overhead [38] [39].

- Finally, we need an additional method call for each method invoked via a membrane. Ignoring possible low-level optimizations, this effectively doubles the time required for an (empty) method call. However, membranes are only introduced for component interfaces that include optional methods, thus, this situation will not occur at high frequencies. We also will investigate the benefit of dynamically testing when membranes can safely be removed again. In the example in Section IV-F, the membrane introduced when the reference to the `Calendar` object was converted from `Provider1` to `Provider2` can be removed again when the reference is converted back to `Provider1`. The problem is that we need to trade the time required for this run-time test when assigning a reference against the time saved when calling methods via this reference. Thus, this optimization requires a more elaborate analysis during the ahead-of-time compilation, which is part of our future work.

So the only unavoidable run-time overhead of our approach is due to the introduction of membranes. However, membranes are only needed when a decision on the granted permissions should be possible at run-time, which imperatively implies that also the permission checks must be performed at run-time.

In summary, the implementation of COSMA proves that using our intermediate representation and type system, we can efficiently execute components while offering a high degree of protection by enforcing fine grained access permissions.

VI. Conclusion and Future Work

In this paper, we proposed a novel concept for the secure cooperation of untrusted components. This involves a component model, where each component declares its required and granted permissions via a self-explanatory public interface. This interface can then be used to connect it to other components. Components are distributed in the form of a secure byte code with high-level instructions that preserves typing information, but still protects intellectual property rights. The corresponding VM implements a type system ensuring that a component cannot gain more permissions than those explicitly defined in its public interface. Type checking is done at deployment time, with some additional run-time checks, where necessary. Coercion semantics is used to automatically insert membranes.

At present we have a fully operational implementation of the type system and the VM, as well as a compiler translating a Java-like language into our byte code. The implementation is freely available at the COSMA website [40]. A formal specification of the type system, including subtyping and coercion, is also available, along with the semantics of the implemented instructions and a formal proof that no instruction sequence can amplify a component's permissions (see Appendix).

In the current implementation all components are executed by the same VM, thus, security of network connections is not an issue. In the future, the model can be extended to distributed systems using remote method invocation, provided that the communication link between the VMs uses a secure protocol ensuring authentication and integrity.

We are currently working on an extension of our type system that integrates structural and nominal typing into a unified framework by explicitly considering the required semantics of methods. In this way, we enable a flexible and safe reuse of components and at the same time fulfill all the subtyping desiderata outlined in [41] (i.e., flexible assignment of responsibility, modular extensibility of the subtyping relation, unique name introduction and traceability) without the idiosyncrasies of that approach, especially avoiding non-transitiveness of the subtype relation.

We are also working on a more complete operating system that meets the special requirements of safe and secure component-based software. This includes in particular memory management for strongly interacting components, such as paging and garbage collection. The latter is especially interesting in combination with fault tolerance and error recovery: If an error occurs in one component, the effects for all other components interacting with it must be as small as possible. To this end, we are looking for a solution to make reloading or replacing a component mostly transparent to its users.

Based on our current work, we will investigate the performance of our VM in more detail, comparing it against plain Java, in order to assess the costs for run-time checks and the indirection caused by the use of membranes. Other topics, which we will address in future are the integration of alias control (cf. Section II), error and exception handling, mechanisms for the revocation of permissions and optimizations such as the removal of unnecessary membranes discussed in Section V-B. Our ultimate goal is to provide a complete programming system, consisting of a VM, an OS and a high-level language compiler, which can be used to develop and deploy component-based software in an easy, secure and efficient way.

References

[1] R. Wismüller and D. Ludwig, "Secure Cooperation of Untrusted Components," in Twelfth Intl. Conf. on Emerging Security Information, Systems and Technologies (SECURWARE 2018). Venice, Italy: IARIA, Sep. 2018, pp. 103–107.

[2] S. Maffeis, J. C. Mitchell, and A. Taly, "Object Capabilities and Isolation of Untrusted Web Applications," in Proc. of IEEE Symp. Security and Privacy. Oakland, CA, USA: IEEE, May 2010, pp. 125–140.

[3] M. S. Miller and J. S. Shapiro, "Paradigm Regained: Abstraction Mechanisms for Access Control," in Advances in Computing Science - ASIAN 2003. Progamming Languages and Distributed Computation, ser. LNCS, vol. 2896. Springer, 2003, pp. 224–242.

[4] M. S. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. Thesis, Johns Hopkins University, Baltimore, Maryland, May 2006.

[5] T. Murray, "Analysing object-capability security," in Proc. of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security, Pittsburgh, PA, USA, Jun. 2008, pp. 177–194.

[6] A. Mettler, D. Wagner, and T. Close, "Joe-E: A Security-Oriented Subset of Java," in Network and Distributed Systems Symposium. Internet Society, Jan. 2010, pp. 357–374.

[7] M. Stiegler, "Emily: A High Performance Language for Enabling Secure Cooperation," in Fifth Intl. Conf. on Creating, Connecting and Collaborating through Computing C5'07. Kyoto, Japan: IEEE, Jan. 2007, pp. 163–169.

[8] D. A. Gutierrez, F. T. Martínez, F. A. García, M. A. D. Fondón, R. I. Castanedo, and J. M. C. Lovelle, "An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System," in Object-Oriented Technology ECOOP, Workshop Reader, ser. LNCS, vol. 1357. Jyväskylä, Finland: Springer, Jun. 1997, pp. 537–544.

[9] R. P. e Silva and R. T. Price, "Component Interface Pattern," in Proc. 9th Conf. on Pattern Language of Programs, Monticello, IL, USA, Sep. 2002. [Online]. Available: http://hillside.net/plop/plop2002/final/plop2002_rpsilva0_1.pdf [last access: 17.05.2019]

[10] S. Mouelhi, K. Agrou, S. Chouali, and H. Mountassir, "Object-Oriented Component-Based Design using Behavioral Contracts: Application to Railway Systems," in Proc. 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CompArch '15). Montreal, QC, Canada: ACM, May 2015, pp. 49–58.

[11] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition," Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, USA, Technical Report CMU/SEI-2000-TR-008, ESC-TR-2000-007, May 2000. [Online]. Available: https://resources.sei.cmu.edu/asset_files/TechnicalReport/2000_005_001_13715.pdf [last access: 17.05.2019]

[12] J. Andronick, D. Greenaway, and K. Elphinstone, "Towards proving security in the presence of large untrusted components," in Proc. 5th Intl. Workshop on System Software Verification, Vancouver, BC, Canada, Oct. 2010. [Online]. Available: https://www.usenix.org/legacy/events/ssv10/tech/full_papers/Andronick.pdf [last access: 17.05.2019]

[13] H. M. Levy, Capability-Based Computer Systems. Digital Press, 1984. [Online]. Available: http://homes.cs.washington.edu/∼levy/capabook [last access: 17.05.2019]

[14] M. S. Miller, B. Tulloh, and J. S. Shapiro, "The Structure of Authority: Why Security Is not a Separable Concern," in Proc. 2nd Intl. Conf. on Multiparadigm Programming in Mozart/Oz. Charleroi, Belgium: Springer, 2004, pp. 2–20.

[15] M. S. Miller, K. P. Yee, and J. Shapiro, "Capability Myths Demolished," Systems Research Laboratory, Johns Hopkins University, Technical Report SRL2003-02, 2003. [Online]. Available: http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf [last access: 17.05.2019]

[16] M. A. D. Fondon, D. A. Gutierrez, L. T. Martinez, F. A. Garcia, and J. M. C. Lovelle, "Capability-based protection for integral object-oriented systems," in Proc. Computer Software and Applications Conf. COMPSAC '98. Vienna, Austria: IEEE, Aug. 1998, pp. 344–349.

[17] M. A. D. Fondon, D. A. Gutierrez, A. G. M. Sánchez, F. A. García, F. T. Martínez, and J. M. C. Lovelle, "Integrating capabilities into the object model to protect distributed object systems," in Proc. Intl. Symp. on Distributed Objects and Applications. Edinburgh, GB: IEEE, Sep. 1999, pp. 374–383. [Online]. Available: http://dx.doi.org/10.1109/DOA.1999.794067 [last access: 17.05.2019]

[18] P. Müller and A. Poetzsch-Heffter, "Universes: A type system for controlling representation exposure," in Programming Languages and Fundamentals of Programming, A. Poetzsch-Heffter and J. Meyer, Eds. Fernuniversität Hagen, 1999, pp. 131–140, Technical Report 263.

[19] W. Dietl and P. Müller, "Ownership Type Systems and Dependent Classes," in Intl. Workshop on Foundations of Object-Oriented Languages (FOOL'08), San Francisco, CA, USA, Jan. 2008.

[20] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," in Proc. 13th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98), Vancouver, Canada, Oct. 1998, pp. 48–64.

[21] S. Balzer, T. Gross, and P. Müller, "Selective ownership: Combining object and type hierarchies for flexible sharing," in Foundations of Object-Oriented Languages (FOOL), J. Boyland, Ed., 2012.

[22] B. Bokowski and J. Vitek, "Confined Types," in Proc. 14th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99), Denver, CO, USA, Nov. 1999, pp. 82–96.

[23] P. W. L. Fong, "Link-Time Enforcement of Confined Types for JVM Bytecode," in Proc. 3rd Annual Conf. on Privacy, Security and Trust (PST'05), St. Andrews, Canada, Oct. 2005, pp. 191–202.

[24] ——, "Discretionary capability confinement," in Proc. 11th European Symposium On Research In Computer Security (ESORICS'06), ser. LNCS, vol. 4189. Hamburg, Germany: Springer, Sep. 2006, pp. 127–144.

[25] ——, "Discretionary Capability Confinement," International Journal of Information Security, vol. 7, no. 2, Apr. 2008, pp. 137–154.

[26] D. Volpano and G. Smith, "A Type-Based Approach to Program Security," in TAPSOFT '97: Theory and Practice of Software Development, ser. LNCS, M. Bidoit and M. Dauchet, Eds., vol. 1214. Springer, 1997, pp. 607–621.

[27] A. Gollamudi and S. Chong, "Automatic Enforcement of Expressive Security Policies using Enclaves," in Proc. 2016 ACM SIGPLAN Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16), Amsterdam, Netherlands, Nov. 2016, pp. 494–513.

[28] P. W. L. Fong and C. Zhang, "Capabilities as alias control: Secure cooperation in dynamically extensible systems," Dept. of Computer Science, Univ. of Regina, Regina, Canada, Technical Report CS-2004-3, Apr. 2004.

[29] D. Hagimont, J. Mossière, X. R. de Pina, and F. Saunier, "Hidden Software Capabilities," in Proc. 16th Intl. Conf. on Distributed Computing Systems (ICDCS '96), Hong Kong, May 1996, pp. 282–289.

[30] J. Boyland, J. Noble, and W. Retert, "Capabilities for Sharing - A Generalisation of Uniqueness and Read-Only," in 15th European Conf. on Object-Oriented Programming (ECOOP '01), Budapest, Hungary, Jun. 2001, pp. 2–27.

[31] P. W. L. Fong and S. Orr, "A Module System for Isolating Untrusted Software Extensions," in Proc. 22nd Annual Computer Security Applications Conf. (ACSAC'06), Miami Beach, Florida, USA, Dec. 2006, pp. 203–212.

[32] C. Hawblitzel, C. C. Chang, G. Czajkowski, D. Hu, and T. von Eicken, "SLK: A Capability System Based on Safe Language Technology," Cornell University, Technical Report, Mar. 1997. [Online]. Available: http://www.cs.cornell.edu/slk/papers/slk.pdf [last access: 17.05.2019]

[33] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd ed. Boston, MA, USA: Addison-Wesley, 2002.

[34] B. C. Pierce, Types and programming languages. MIT Press, 2002.

[35] L. Cardelli, "Typeful Programming," in Formal Description of Programming Concepts, E. Neuhold and M. Paul, Eds. Springer, 1991, pp. 431–507.

[36] M. Stiegler, "The E Language in a Walnut," 2000. [Online]. Available: http://www.skyhunter.com/marcs/ewalnut.html [last access: 17.05.2019]

[37] X. Leroy, "Java bytecode verification: Algorithms and formalizations," Journal of Automated Reasoning, vol. 30, no. 3, May 2003, pp. 235–269. [Online]. Available: https://doi.org/10.1023/A:1025055424017 [last access: 17.05.2019]

[38] A. M. Schiffman and L. P. Deutsch, "Efficient Implementation of the Smalltalk-80 System," in Proc. 11th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL '84), 1984, pp. 297–302.

[39] J. Gil and I. Maman, "Whiteoak: Introducing Structural Typing into Java," in Proc. 23rd ACM SIGPLAN Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08), Nashville, TN, USA, Oct. 2008, pp. 73–90.

[40] "COSMA – A Virtual Machine Supporting the Secure Cooperation of Untrusted Components." [Online]. Available: https://www.bs.informatik.uni-siegen.de/forschung/cosma [last access: 28.05.2019]

[41] K. Ostermann, "Nominal and Structural Subtyping in Component-Based Programming," Journal of Object Technology, vol. 7, no. 1, Jan. 2008. [Online]. Available: http://www.jot.fm/issues/issue_2008_01/article4 [last access: 17.05.2019]

APPENDIX

In this section we present the semantics of COSMA's instruction set. The auxiliary rules we use here can be found in Figure 12. We also provide a proof sketch that the semantics successfully prevents an amplification of access rights.

A. Basic Definitions

According to the structure of the intermediate representation, we use the following (abstract) types (cf. Figure 10):

- V for instances of `Variable`
- F for instances of `Attribute`
- $L = V \cup F$ for storage locations
- B for instances of `BasicBlock`
- T for instances of `Type`
  - $\tau_{ref} \in T$ is a reference type
  - $\tau_{obj} \in T$ is an object type
  - $\tau_{val} \in T$ is a value type
- Id for method names
- $\mathbb{N}, \mathbb{Z}$ as usual
- $\mathbb{B}$ for Boolean values
- $Impl : (\text{name} : Id) \times (\text{var} : V^*) \times (\text{nargs} : \mathbb{N}) \times (\text{resT} : T^*) \times (\text{blocks} : B^*) \times (\text{clazz} : Class)$ for method implementations
- $Decl : (\text{name} : Id) \times (\text{opt} : \mathbb{B}) \times (\text{argT} : T^*) \times (\text{resT} : T^*)$ for method declarations
- $Class : (\text{pub} : Impl^*) \times (\text{priv} : Impl^*) \times (\text{attr} : F^*) \times (\text{child} : Class^*)$ for classes and class types
- $Iface : (\text{methods} : Decl^*)$ for interface types.
- $\mathcal{CA}$ for type cast actions
- RT for generic run-time values
  - Val for numeric run-time values
  - $Obj : (\text{methods} : Impl^*) \times (\text{config} : F^*)$ for objects
  - $Mem : (\text{obj} : Obj) \times (\text{actions} : (Id \times \mathcal{CA}^*)^*)$ for membranes
- Γ for the type environment

A *Frame* is defined as a tuple $(\text{obj} : Obj) \times (\text{method} : Impl) \times (\text{block} : \mathbb{N}) \times (\text{pc} : \mathbb{N}) \times (\text{var} : V^*) \times (\text{res} : L^*) \times (\text{resT} : T^*) \times (\text{mem} : f_A(\text{obj}))$, where obj represents the current object, $method$ the currently executing method with the basic block number $block$ and its program counter pc. The current values of the method's variables are stored in var, and res is a list of operands to write back the method's results to the caller. $resT$ holds the result types expected by the caller. If the current method was called on a membrane, a reference to the membrane is stored in mem. For simplification we assume that the last instruction in each basic block is either a return statement or an unconditional jump into the following basic block, i.e., we do not need to model overflows of the program counter. A *state* S is a stack of frames. We write $s :: t$ to split S into the topmost frame s and the tail t.

$BlockIndex(m, b) = i$ holds, if there exists exactly one basic block in m's implementation that *is* b and this block is the i-th block in m.

$$\textsc{BlockIndex}\ \frac{m : Impl \quad b : B \quad \exists_1 i \geq 0 : m.blocks_i = b}{BlockIndex(m, b) = i}$$

To lookup a method implementation for a method named id inside an object o:

$$\textsc{LookUp}\ \frac{o : Obj \quad \exists_1 m \in o.methods : m.name = id}{lookup(id, o) = m}$$

Object creation follows the INSTANTIATE rule:

$$\textsc{Instantiate}\ \frac{class : Class \quad \langle class.pub \cup class.priv, class.attr \rangle = o}{Instantiate(class) = o}$$

Figure 12. Semantic functions and auxiliary rules

All semantic rules are formulated as a state transition of one stack configuration to another. Possible effects of these transitions are:

1) the top-most frame is changed,
2) a new frame is pushed onto the stack,
3) the top-most frame is removed from the stack, or
4) any meaningful combination of that.

If a frame is changed, we only write down the differences in the transition. Everything remaining unchanged is not shown. If for example the program counter is increased, but everything else is not touched, we just write $s[pc \leftarrow pc + 1] :: t$. For new frames we assume that the program counter pc and the block number $block$ are set to 0 and refrain from writing this explicitly.

B. Constants, Branching and Arithmetic Operations

In this subsection we present the semantics for simple instructions, such as conditional and unconditional jumps, loading constants and arithmetic operations and relations.

$$\textsc{Jmp}\ \frac{BlockIndex(s.method, b) = blk}{s :: t \xrightarrow{\texttt{jmp b}} s[block \leftarrow blk, pc \leftarrow 0] :: t}$$

JMP jumps to the given block and resets the program counter. It holds, iff the given block is part of the current method.

$$\textsc{Cjmp-T}\ \frac{BlockIndex(s.method, b) = blk \quad v \rightarrow val : Val \quad (v \neq 0) = nz}{s :: t \xrightarrow{\texttt{cjmp v nz b}} s[block \leftarrow blk, pc \leftarrow 0] :: t}$$

CJMP-T jumps to the given block and resets the program counter. It holds, if:

1) the block is part of the current method,

2) v contains a value val (e.g., integer),
3) $val \neq 0$ and nz is set to true (jump on non-zero), or $val = 0$ and nz is false.

$$\textsc{Cjmp-F}\ \frac{\begin{array}{c} BlockIndex(s.method, b) = blk \\ v \rightarrow val : Val \\ (v \neq 0) \neq nz \end{array}}{s :: t \xrightarrow{\texttt{cjmp v nz b}} s[pc \leftarrow pc + 1] :: t}$$

CJMP-F increments the program counter. It holds, iff CJMP-T fails only on the third premise.

$$\textsc{Load}\ \frac{const : \mathbb{Z} \quad \Gamma \vdash dst : \tau_{ref} \Rightarrow const = 0}{s :: t \xrightarrow{\texttt{load const dst}} s[pc \leftarrow pc + 1, dst \leftarrow const] :: t}$$

LOAD loads the constant $const$ into the operand dst and increases the program counter. It holds for every dst having a value type. If dst has a reference type, $const$ must be 0 ("*null*").

$$\textsc{Op}\ \frac{\begin{array}{c} lhs \rightarrow val_1 : Val \\ rhs \rightarrow val_2 : Val \\ \Gamma \vdash dst : \tau_{val} \\ f \in \{\div, -, \texttt{mod}, \cdot, +\} \end{array}}{\begin{array}{c} s :: t \xrightarrow{\texttt{op lhs rhs f dst}} \\ s[pc \leftarrow pc + 1, dst \leftarrow val_1 \circ_f val_2] :: t \end{array}}$$

$$\textsc{Test}\ \frac{\begin{array}{c} lhs \rightarrow val_1 : Val \\ rhs \rightarrow val_2 : Val \\ \Gamma \vdash dst : \tau_{val} \\ f \in \{=, \neq, <, \geq, >, \leq\} \end{array}}{\begin{array}{c} s :: t \xrightarrow{\texttt{test lhs rhs f dst}} \\ s[pc \leftarrow pc + 1, dst \leftarrow val_1 \circ_f val_2] :: t \end{array}}$$

OP and TEST increase the program counter and calculate $val_1 \circ_f val_2$ to store it in the given operand dst. Both rules hold, if:

1) both source operands point to values,
2) the target operand has a value type and
3) the operator is valid.

C. Object Creation and Assignments

$$\textsc{New}\ \frac{\begin{array}{c} c : Class \\ \Gamma \vdash c : \tau_1 \\ \Gamma \vdash dst : \tau_2 \\ Instantiate(c) = o \\ f_{ca(\tau_1, \tau_2)}(o) = o\prime \end{array}}{s :: t \xrightarrow{\texttt{new c dst}} s[pc \leftarrow pc + 1, dst \leftarrow o\prime] :: t}$$

NEW increases the program counter and stores a reference to a newly created object (or to a membrane for that object). It holds, if:

1) the given class has type τ_1,
2) the target operand has type τ_2, and
3) an object o is created from the given class and
4) can be assigned to the target operand.

$$\textsc{Mov}\ \frac{\begin{array}{c} \Gamma \vdash src : \tau_1 \\ \Gamma \vdash dst : \tau_2 \\ src \rightarrow o : RT \\ f_{ca(\tau_1, \tau_2)}(o) = o\prime \end{array}}{s :: t \xrightarrow{\texttt{mov src dst}} s[pc \leftarrow pc + 1, dst \leftarrow o\prime] :: t}$$

MOV increases the program counter and assigns src to dst, performing the necessary cast actions. It holds, if src is initialized, and o is assignable to the target operand.

D. Calling Methods and Returning Results

$$\textsc{ChkType}\ \frac{\begin{array}{c} \Gamma \vdash r : \tau_{ref} \\ t : T \\ \tau_{ref} \prec: T = \text{assignable} : \mathbb{B} \end{array}}{\begin{array}{c} s :: t \xrightarrow{\texttt{chktype r t dst}} \\ s[pc \leftarrow pc + 1, dst \leftarrow \text{assignable}] :: t \end{array}}$$

CHKTYPE increases the program counter and writes **True** to dst, iff r is assignable to the given type t.

$$\textsc{Call}\ \frac{\begin{array}{c} r \rightarrow o : Obj \ \wedge\ \Gamma \vdash r : \tau \ \wedge\ \tau : Iface \\ m : Decl \\ m \in \tau.methods \\ lookup(m.name, o) = m\prime : Impl \\ args = (a_1, \ldots, a_n) \ \wedge\ \Gamma \vdash a_i : \tau_i \\ n = m\prime.nargs \\ (f_{ca(\tau_1, m.argT_1)}(a_1), \ldots, f_{ca(\tau_n, m.argT_n)}(a_n)) = Var\prime \\ m.resT : T^k \ \wedge\ res : L^k \end{array}}{\begin{array}{c} s :: t \xrightarrow{\texttt{call r m args res}} \\ [o, m\prime, Var\prime, res, m.resT, \bot] :: s :: t \end{array}}$$

CALL creates a new frame containing the target object, the called method implementation and the method's list of variables including the parameters. The frames also carries the result operands and the result types, as expected by the caller. The rule holds, if:

1) r points to an object o and has type τ,
2) m is a method declaration and available in τ,
3) o's class type has an implementation of m, and
4) the given arguments can be casted to the formal parameter types of m,
5) the number of formal result types matches the given write-back operands.

$$\text{CALL-M}\ \frac{\begin{array}{c} r \to o\prime = f_A(o)\ \wedge\ o : Obj\ \wedge\ \Gamma \vdash r : \tau \\ m : Decl \\ m \in \tau.methods \\ lookup(m.name, o) = m\prime : Impl \\ args = (a_1, \ldots, a_n)\ \wedge\ \Gamma \vdash a_i : \tau_i \\ n = m\prime.nargs \\ (f_{ca(\tau_1, m.argT_1)}(a_1), \ldots, f_{ca(\tau_n, m.argT_n)}(a_n)) = Var\prime \\ m.resT : T^k\ \wedge\ res : L^k \\ (m.name, (A_1, \ldots, A_n)) \in o\prime.actions \\ (f_{A_1}(Var\prime_1), \ldots, f_{A_n}(Var\prime_n)) = Var\prime\prime \end{array}}{\begin{array}{c} s :: t \xrightarrow{\texttt{call r m args res}} \\ [o, m\prime, Var\prime\prime, res, m.resT, o\prime] :: s :: t \end{array}}$$

CALL-M does the same as CALL, but additionally adds the membrane to the new frame. The rule holds, if:

1) r points to a membrane $o\prime$ for object o,
2) CALL would hold for $r \mapsto o$, and
3) $o\prime$ holds method actions for m that can be applied to the casted parameters.

$$\text{INV-M}\ \frac{\begin{array}{c} r \to o\prime = f_A(o)\ \wedge o : Obj\ \wedge\ \Gamma \vdash r : Any \\ id \to name : Id \\ lookup(name, o) = m : Impl \\ args = (a_1, \ldots, a_n), \Gamma \vdash a_i : Any \\ n = m.nargs \\ m.resT : () \\ (m.name, (A_1, \ldots, A_n)) \in o\prime.actions \\ (f_{A_1}(a_1), \ldots, f_{A_n}(a_n)) = Var\prime \end{array}}{s :: t \xrightarrow{\texttt{inv r id args}} [o, m\prime, Var\prime, (), (), o\prime] :: s :: t}$$

INV-M creates a new frame containing the target object and the called method implementation, as well as the given arguments. Since `inv` is only allowed on references of type `Any`, the reference always points to a membrane, which is added to the new frame. Result operands and result types are not set, since the instruction does not support results. The rule holds, if

1) r r has type *Any* and points to a membrane $o\prime$ for o,
2) o has an implementation for a method named *name*,
3) all arguments have type *Any*,
4) $o\prime$ holds method actions for the called method, which can be applied to the arguments.

$$\text{RET}\ \frac{\begin{array}{c} m : Impl\ \wedge\ m.resT : T^n \\ r : L^n\ \wedge\ \Gamma \vdash r_i : \tau_i\ \wedge\ r_i \mapsto o_i : RT \\ res : L^n\ \wedge\ \Gamma \vdash res_i : \eta_i \\ resT : T^n \\ f_{ca(resT_i, \eta_i)}(f_{ca(\tau_i, m.resT_i)}(r_i)) = \mathcal{V}_i \end{array}}{\begin{array}{c} [\ldots, m, \ldots, res, resT, \bot] :: s :: t \xrightarrow{\texttt{ret r}} \\ s[pc \leftarrow pc + 1, res_i \leftarrow \mathcal{V}_i] :: t \end{array}}$$

RET removes the top-most frame (the callee frame) from the stack and increases the program counter in the caller frame. The operand list res in the callee frame references storage locations available in the caller frame. Writing to these operands changes the caller frame, which is the intended behavior. The instruction performs two typecasts. First, the actual results r_i must be casted to the formal result type of the method implementation $m.resT_i$. Finally, we need to apply a cast action, casting from the interface's result type $resT_i$ to the target operand's type η_i.

$$\text{RET-M}\ \frac{\begin{array}{c} m : Impl\ \wedge\ m.resT : T^n \\ r : L^n\ \wedge\ \Gamma \vdash r_i : \tau_i\ \wedge\ r_i \mapsto o_i : RT \\ res : L^n\ \wedge\ \Gamma \vdash res_i : \eta_i \\ (m.id \times (A_1, \ldots, A_n)) \in o\prime.actions \\ resT : T^n \\ f_{ca(resT_i, \eta_i)}(f_{A_i}(f_{ca(\tau_i, m.resT_i)}(r_i))) = \mathcal{V}_i \end{array}}{\begin{array}{c} [\ldots, m, \ldots, res, resT, o\prime] :: s :: t \xrightarrow{\texttt{ret r}} \\ s[pc \leftarrow pc + 1, res_i \leftarrow \mathcal{V}_i] :: t \end{array}}$$

In case the method was invoked through a membrane as in RET-M, an additional cast is needed between the two type casts mentioned in RET. This cast addresses differences between the method implementation's formal result types and interface's result types.

E. Proof Sketch: No Amplification of Rights

Assumption 1. *X is a context, and $o \in X$ is an object providing method m. $X\prime$ is a different context ($X\prime \neq X$) holding references that point to o or a membrane for o. For all references $r \in X\prime$ pointing to o or a membrane for o, we assume that m is not callable.*

Notation: *For a method m and an interface type T we write $m \in T \Leftrightarrow (m : U, \eta) \in T$. For a reference r we write $m \in r \Leftrightarrow \Gamma \vdash r : T \wedge m \in T$. For a membrane mem we write $m \in mem \Leftrightarrow (m.name, ca) \in mem.actions$. For cast actions $ca(S,T)$ we write $m \in ca(S,T) \Leftrightarrow (m : A) \in ca(S,T)$.*

With this notation, the assumption can be formalized as

$$\forall r \in X\prime : \begin{cases} \Gamma \vdash r : T\ \wedge\ m \notin T & r \mapsto o \\ m \notin membrane(o) & r \mapsto membrane(o) \end{cases}$$

Observation 1. *For assignments from type $S = \{(k_j : S_j, \eta_j)^{j \in 1\ldots z}\}$ to type $T = \{(l_i : T_i, \sigma_i)^{i \in 1\ldots n}\}$ we have three cases:*

1) $S \not\prec: T$
   $\Rightarrow$ *assignment is not possible* $\Rightarrow$ *no need to consider*
2) $S \prec:_m T$
   $\Rightarrow \{l_i{}^{\ i \in 1\ldots n}\} \subseteq \{k_j{}^{\ j \in 1\ldots z}\}\ \wedge\ ca(S,T) = \top$
3) $S \prec: T\ \wedge\ S \not\prec:_m T$
   $\Rightarrow ca(S,T) = \{l_i : ca(S_j, T_i)^{\ i \in 1\ldots n,\ j \in 1\ldots z,\ k_j = l_i}\}$

Theorem 1. *There is no sequence of instructions executed in context $X\prime$ involving any $r \in X\prime$ that results in successfully calling m on o, unless m is called by some method in a different context $X\prime\prime$ or $X\prime\prime$ returns a reference to o allowing m.*

Proof: COSMA supports 14 instructions of which four (`mov`, `call`, `ret`, `inv`) are able to create or copy object

references. The `new`-instruction is of no interest. We perform an induction over the length ℓ of the instruction sequence. For $\ell = 0$, m is not callable on any reference $r \in X\prime$ by assumption (induction hypothesis).

Instruction: `mov r r'`
Let $r \in X\prime$ be a reference of type S and $r\prime$ a new reference of type T.

1) $S \prec:_m T$
   a) $r \mapsto o \overset{\text{Ass}}{\Rightarrow} m \notin S \Rightarrow m \notin T \overset{\text{Mov}}{\Rightarrow} r\prime = r \Rightarrow m \notin r\prime$
   b) $r \mapsto o\prime = f_{ca(S\prime,T\prime)}(o) \overset{\text{Ass}}{\Rightarrow} m \notin ca(S\prime,T\prime) \overset{\text{Mov}}{\Rightarrow} r\prime \mapsto o\prime \Rightarrow m \notin r\prime$
2) $S \prec: T \wedge S \not\prec:_m T$
   a) $r \mapsto o \overset{\text{Ass}}{\Rightarrow} m \notin S \Rightarrow m \notin ca(S,T) \Rightarrow m \notin f_{ca(S,T)}(o) \overset{\text{Mov}}{\Rightarrow} m \notin r\prime$
   b) $r \mapsto o\prime = f_{ca(S\prime,T\prime)}(o) \overset{\text{Ass}}{\Rightarrow} m \notin ca(S\prime,T\prime) \overset{\text{Mov}}{\Rightarrow} r\prime = f_{ca(S,T)}(f_{ca(S\prime,T\prime)}(o)) = f_{\text{merge}(ca(S,T),ca(S\prime,T\prime))}(o) \overset{\text{merge}}{\Rightarrow} m \notin r\prime$

The second case needs more explanation: If r has type S and points directly to o, S has no m (induction hypothesis). Then, m is not part of $ca(S,T)$ following `CA-RCD`. By applying this cast action to o, m is not callable on the resulting membrane.

If r points to a membrane, following the induction hypothesis m is not callable on this membrane. But the membrane results from applying a type cast action $ca(S\prime,T\prime)$ to the object o, so m was not a part of that type cast action. When merging this cast action with any other type cast action, m cannot be part of the result. Therefore it is also not callable through $r\prime$.

Instruction: `call ref m args res`
Let method *foo* accept an argument of type U and $S \prec: U$. We assume that *foo* is callable through some reference *ref* and the actual implementation accepts arguments of type T, with $U \prec: T$. Let $r\prime$ be the name of a parameter in *foo*. The instruction pushes a new frame onto the stack so that the stack will look like this: $[\dots, Var\prime = (r\prime, \dots), \dots] :: s :: t$.

- $m = (foo, opt, argT = (\dots, U, \dots), resT = (\dots))$
- $args = (\dots, r, \dots)$, $res = (\dots)$, $ref \mapsto obj \vee ref \mapsto f_A(obj)$
- $ref \mapsto obj \Rightarrow U \prec:_m T \Rightarrow ca(U,T) = \top$

Again, we look at the different cases:

1) $S \prec:_m U \wedge U \prec:_m T \Rightarrow S \prec:_m T$
   a) $r \mapsto o \overset{\text{Ass}}{\Rightarrow} m \notin S \Rightarrow m \notin T \overset{\text{Call}}{\Rightarrow} r\prime \mapsto f_\top(o) = o \Rightarrow m \notin r\prime$
   b) $r \mapsto f_{ca(S\prime,T\prime)}(o) \overset{\text{Ass}}{\Rightarrow} m \notin ca(S\prime,T\prime) \overset{\text{Call}}{\Rightarrow} r\prime \mapsto f_\top(f_{ca(S\prime,T\prime)}(o)) = f_{ca(S\prime,T\prime)}(o) \Rightarrow m \notin r\prime$
2) $S \not\prec:_m U \vee U \not\prec:_m T \Rightarrow S \not\prec:_m T$
   a) $r \mapsto o \overset{\text{Ass}}{\Rightarrow} m \notin S \Rightarrow m \notin ca(S,U) \Rightarrow m \notin f_{ca(S,U)}(o)$
   $\Rightarrow m \notin f_{ca(U,T)}(f_{ca(S,U)}(o)) \overset{\text{Call}}{\Rightarrow} m \notin r\prime$
   b) $r \mapsto o\prime = f_{ca(S\prime,T\prime)}(o) \overset{\text{Ass}}{\Rightarrow} m \notin ca(S\prime,T\prime)$
   $\overset{\text{Call}}{\Rightarrow} r\prime = f_{ca(U,T)}(f_{ca(S,U)}(f_{ca(S\prime,T\prime)}(o)))$
   $\overset{\text{merge}}{\Rightarrow} m \notin r\prime$

The proof for method calls performed on a membrane follows directly from Call-Mem and merge.

Instruction: `inv x mth r`
The `inv`-instruction accepts only arguments of type Any. In order to pass r as an argument to *mth* it needs to be casted it to this special type. This always introduces a membrane $f_{ca(S,Any)}(r)$ allowing exactly the methods that are callable through r, because $S \not\prec:_m Any$. This membrane can then be assigned to a reference $r\prime$ of the methods formal parameter type T with just a type check ($Any \prec:_m T \wedge Any \not\prec:_c T$).

$$m \notin r \overset{\text{Inv-M}}{\Rightarrow} m \notin r\prime$$

Instruction: `ret`
Let *bar* be the current method with formal return type $(\dots, U, \dots)$ that was invoked through an interface $bar : X \mapsto (\dots, V, \dots)$. Let $s\prime :: s :: t$ be a snapshot of the system's state right before the execution of the `return`-instruction.

$$\begin{aligned} s\prime = [\ & obj, \\ & bar, \\ & Var\prime = (\dots, r, \dots), \\ & ResT\prime = (\dots, V, \dots), \\ & Res\prime = (\dots, r\prime \dots), \\ & \bot\ ] \end{aligned}$$

Let $\Gamma \vdash r :: S \wedge \Gamma \vdash r\prime :: T$ and $S \prec: U$.

- $S \prec: U \Rightarrow ca(S,U) \neq \bot$
- $V \prec: T \Rightarrow ca(V,T) \neq \bot$
- $U <: V \Rightarrow ca(U,V) = \top$ (follows directly from CM-Arrow)

Again, look at the different cases:

1) $S \prec:_m U \wedge V \prec:_m T \Rightarrow S \prec:_m T$
   a) $r \mapsto o \overset{\text{Ass}}{\Rightarrow} m \notin S \Rightarrow m \notin T \Rightarrow m \notin r\prime$
   b) $r \mapsto f_{ca(S\prime,T\prime)}(o) \overset{\text{Ass}}{\Rightarrow} m \notin ca(S\prime,T\prime) \overset{\text{Ret}}{\Rightarrow} r\prime \mapsto f_{ca(S\prime,T\prime)}(o) \Rightarrow m \notin r\prime$
2) $S \not\prec:_m U \vee V \not\prec:_m T \Rightarrow S \not\prec:_m T$
   a) $r \mapsto o \overset{\text{Ass}}{\Rightarrow} m \notin S$
   $\Rightarrow m \notin \text{merge}(\text{merge}(ca(S,U),\top), ca(V,T))$
   $\overset{\text{Ret}}{\Rightarrow} r\prime \mapsto f_{\text{merge}(\text{merge}(ca(S,U),\top),ca(V,T))}(o)$
   $\Rightarrow m \notin r\prime$
   b) $r \mapsto f_{M=ca(S\prime,T\prime)}(o) \overset{\text{Ass}}{\Rightarrow} m \notin r$
   $\Rightarrow m \notin ca(S\prime,T\prime)$
   $\overset{\text{Ret}}{\Rightarrow} r\prime = f_Y(f_M(o))$ with
   $Y = \text{merge}(\text{merge}(ca(S,U),\top), ca(V,T))$
   $\Rightarrow r\prime = f_{\text{merge}(Y,M)}(o)$
   $\Rightarrow m \notin r\prime$

The proof for returning from a method called on a membrane follows directly from Ret-M and merge.

■