Secure Cooperation of Untrusted Components

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 suggest 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 which are not strictly necessary, but desired for a better service level. These ideas can be 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.**

***Keywords*—Software-components; security; typesystems.**

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 component's sources and thus, the components themselves may not be trusted [1]. 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 [2]. 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 [3,4], where unforgeable object references are used as a capability allowing to use the referenced object.

A good introduction to the object capability model and POLA is provided in [5]. The general properties of capability systems, as well as some common misconceptions about capabilities are pointed out in [6], where the authors also show that capabilities have strong advantages over access control lists and can support both confinement and revocation. Murray [4] discusses several common object capability patterns, including membranes, which allow a deep attenuation of rights.

Based on the object-capability paradigm, several secure languages have been devised. A pioneer in this area is the work of Mark Miller [3] 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 method level access control, but requires the programmer to manually implement security-enforcing abstractions, like membranes. Based on E, Joe-E [7] restricts Java such that access to objects is only possible via capabilities that have been explicitly passed to a component. Joe-E also supports immutable interfaces allowing to implement secure plug-ins. It 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 [8] is a secure subset of OCaml, whereas Maffeis et al. [1] specifically address the problem of mutual isolation of (third-party) web applications written in JavaScript. These language-based approaches share two fundamental problems: Since they restrict the programming language, they not only confine interactions between components, but also limit the programmer's capabilities within a component. Another drawback is that security can only be guaranteed, if all components are distributed at the source code level, which in practice is infeasible for reasons of protecting intellectual property rights.

A feasible solution for the second problem is the use of a Virtual Machine (VM) that enforces security. An example for such an approach is Oviedo3, which includes a secure VM implementing capability-based access control at the granularity of methods [9]–[11]. However, Oviedo3 only provides basic mechanisms for the management of access rights, i.e., adding and removing the permission to execute a single method for a single object reference, and must check all these permissions at run-time. Thus, Oviedo3 is neither easy to use nor efficient.

To overcome the drawbacks of existing approaches, the goal of our work is to provide a VM that

- 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,
- minimizes the number of required run-time checks by performing most checks when a component is deployed.

In this paper, we suggest an easy to use approach 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 Section II, we present a component model, where each component specifies its minimal and desired permissions in a natural way using interfaces. We then outline the basics of

 ISBN: 978-1-61208-661-3

a type system that allows fine-grained access restrictions and optional methods (Section III). Finally, we introduce concepts for a virtual machine and a secure, strongly-typed byte code, that allows static type checking at deployment time and the automatic creation of membranes (Section IV). We conclude the paper by giving an outlook to our future work (Section V).

II. 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*" [12]. 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 starting point for defining the component's interface.

Under these conditions, at run-time a component can be viewed as a collection of objects. Thus, secure interaction between components can be implemented by an extended object capability model, where the type of a reference imposes additional access restrictions.

In this run-time model, we can exactly determine the interface that a component C requires from its environment (by determining the types of all references and values that C can receive), as well as the interface it provides to the environment (i.e., the types of all references and values that C returns) by just examining the type of C's principal class. Now, a central idea of our approach is to view these interfaces also as a specification of the required (requested) and provided (granted) permissions of a component. E.g., if a method m is in C's required interface, then C requires the permission to invoke m. As an extension, we also allow optional methods in component interfaces. In this way, the type of the principal class explicitly defines

- $\mathcal{T}_{\text{in}}$: 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
- $\mathcal{T}_{\text{out}}$: 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.

As an example, consider a calendar component that holds objects implementing an interface `Appointment`. Users can create new appointments or get a list of all stored ones. The public interface of this component could look like shown in Listing 1 (assuming `String` is a built-in type).

As the component has no input (we omitted the parameters of `createAppointment()` for simplicity), `Calendar` does not request any permissions from its environment, so $\mathcal{T}_{\text{in}} = \emptyset$. In contrast, it grants permission to use the stored appointments via the `Appointment` interface, which results in $\mathcal{T}_{\text{out}} = \{\texttt{Calendar}, \texttt{Appointment}\}$.

Listing 1. Calendar interface

```
component interface Calendar {
  interface Appointment {
    int startTime();
    int endTime();
    String location();
    String subject();
  }
  void createAppointment(...);
  Appointment[] getAppointments();
}
```

A calendar client displaying the appointments stored in a calendar may have a component interface similar to Listing 2.

Listing 2. Calendar client interface

```
component interface CalendarClient {
  interface CalendarProvider {
    Event[] getAppointments();
  }
  interface Event {
    int startTime();
    int endTime();
    optional String subject();
  }
  void displayEvents();
  void setProvider(CalendarProvider c);
}
```

This interface specifies the permissions the client needs from a `CalendarProvider`: it must be able to call the `getAppointments()` method, which returns an array of objects 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 $\mathcal{T}_{\text{out}} = \{\texttt{CalendarClient}\}$ and $\mathcal{T}_{\text{in}} = \{\texttt{CalendarProvider}, \texttt{Event}\}$. 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`.

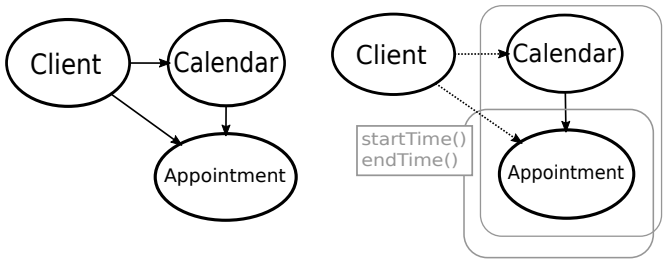

Figure 1. Full access to `Appointment` (left) versus restricted permissions (right).

In this example, the calendar client will not be able to

TABLE I. INTENDED SEMANTICS OF COMPONENT INTERFACE TYPES.

Status of method m in interface type T	Assertion that 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 T	yes	yes

call the `location()` method on events received from any `CalendarProvider`, 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 $T \in \mathcal{T}_{\text{in}}(C)$, 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 calendar object. Traditionally, this requires to manually program a membrane for the `Calendar` component, such that the objects returned by `getAppointments()` do not have a `subject()` method (see Figure 1). 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 Figure 1, the client has access to `Calendar` through a membrane. The calendar membrane's `getAppointments()` method in turn returns membranes for the `Appointment` objects, that only allow two methods to be called.

In principle, if the `Calendar` component declared the `subject()` method in `Appointment` as optional, it also could decide at runtime whether or not to expose this method to the client invoking `getAppointments()`, based, e.g., on some authentication procedure. However, we believe that this decision should 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 at run-time. We will briefly present our type system in the next section.

III. TYPE SYSTEM

As outlined before, we interpret a component's interface type as a specification of access permissions for methods. In addition, we retain the traditional interpretation, which asserts that all objects implementing the interface will offer the specified methods. We achieve both goals by using optional methods, as shown in Table I.

As the main goal of our type system is security, it must enforce the access restrictions given in Table I in such a way that no component can amplify its rights by type conversions, i.e., down-casting. 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 component interface type I, all methods in I can be invoked without type errors. Finally, the type system supports an easy attenuation of rights by just up-casting a reference, without the need to manually code a membrane.

For safety and security reasons, we allow the VM to load a component, only if the component's code is *well-typed*. According to Cardelli [13], 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, when at least one of these types is a component interface type? The important restriction here is that we must not allow r' to gain more permissions than r via down-casting.

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'. 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 requred 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 run-time check, i.e., all its arguments and results can be assigned without check (this avoids delayed type failures).

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 (if the referenced object o provides that method), while S does not. Thus, we actually can assign $r : S$ to $r' : T$, 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 type systems enables the construction of a secure VM, which can decide at deployment time for which assignments in a component's code a run-time check is required and/or a membrane must be introduced.

IV. COSMA

The *Component Oriented Secure Machine Architecture* (COSMA) is a secure VM based on the outlined type system. It comes with a specification for an object oriented byte code, called *Component Intermediate Language*. The structure of this byte code 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. Such a block is a sequence of instructions and is the only admissible target of a branch

 ISBN: 978-1-61208-661-3

instruction. 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 for component 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*" [14]. In order to prevent such attacks, we must use "*computers on which only capability-secure programs are allowed*" [14]. 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 X creates another object, the new object also is associated with X. 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 component interface type and thus are subject to the security restrictions outlined in Section III. References with "normal" class or interface types are also supported, but 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 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 [15]. 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 permissions are "casted away". Thus security constraints are enforced mainly statically, leaving only a few run-time checks.

V. Conclusion and Future Work

In this paper, we propose a new 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 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 mentioned 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 minimalistic language into our byte code. 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.

In the current implementation all components are executed by the same VM, thus, security of the 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 another compiler for a more mature, Java-like programming language, that enables us to execute more realistic programs. This will allow us to compare our implementation to other approaches. Especially, we will evaluate its performance against plain Java, so we can assess the costs for the run-time checks and the indirection caused by the use of membranes. Our long term goal is to provide a complete programming system that can be used to develop and deploy component-based software in an easy and secure way.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] M. S. Miller, K. P. Yee, and J. Shapiro, "Capability Myths Demolished," Systems Research Laboratory, Johns Hopkins University, Technical Report SRL2003-02, 2003, http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf [retrieved: 7, 2018].

[7] 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.

[8] 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.

[9] D. A. Gutierrez *et al.*, "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. Jyvskyl, Finland: Springer, Jun. 1997, pp. 537–544.

 ISBN: 978-1-61208-661-3

[10] 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 Conference COMPSAC '98*. Vienna, Austria: IEEE, Aug. 1998, pp. 344–349.

[11] M. A. D. Fondon *et al.*, "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

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

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

[14] M. Stiegler, "The E Language in a Walnut," 2000, http://www.skyhunter.com/marcs/ewalnut.html [accessed: 7, 2018].

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

 ISBN: 978-1-61208-661-3