# Type Capabilities for Object-Oriented Programming Languages

Xi Wu[1(✉)], Yi Lu[2], Patrick A. Meiring[1], Ian J. Hayes[1], and Larissa A. Meinicke[1]

[1] School of ITEE, The University of Queensland, Brisbane 4072, Australia
{xi.wu,p.meiring,l.meinicke}@uq.edu.au, Ian.Hayes@itee.uq.edu.au
[2] Oracle Labs, Brisbane 4000, Australia
yi.x.lu@oracle.com

**Abstract.** Capabilities are used to control access to system resources. In modern programming languages that execute code with different levels of trust in the same process, the propagation of such capabilities must be controlled so that they cannot unintentionally be obtained by unauthorised code. In this paper, we present a statically-checked type system for object-oriented programming languages which guarantees that capabilities are restricted to authorised code. Capabilities are regarded as types that are granted to code based on a user-defined policy file (similar to that used by Java). In order to provide a finer-grained access control, the type system supports parameterised capabilities to more precisely identify system resources. The approach is illustrated using file-access examples.

**Keywords:** Capability-based security · Access control
Authorisation · Parameterisation · Programming language

## 1 Introduction

The concept of capability-based security [5,16], in which a capability is regarded as a communicable and unforgeable token of authority, has been used in operating systems. A process inside the system, which possesses a capability, is authorised to use the referenced object according to the operations that are specified on that capability. In this model, the acquisition of capabilities is limited by authorisation at the process-level, and forgery is prevented by storing capabilities in a memory region protected from direct application writes. Capabilities can be shared, but only through operating system APIs, which can enforce the correct passing of capabilities based on the Principle Of Least Privilege (POLP) [15]. In operating systems, processes are mostly isolated (i.e., run in different memory spaces and can only communicate via restricted channels), and so it is relatively straight-forward to ensure that capabilities are not leaked to unauthorised processes.

The goal of our work is provide access control at the programming-language level using a capability-based approach. However, although capabilities may also be used at the application (i.e. programming language) level to control access to resources, their use in this context is complicated by the fact that both trusted and untrusted code may be executing *within* the same process, and so it is necessary to control the flow of capabilities within the same process itself. This is challenging because of the use of shared memory and pointers, and the level of interaction between trusted and untrusted code. In this context, *language-based security* [7,14] approaches may be used to prevent vulnerabilities that are not addressed by process-based access control at the operating system level.

One of the main approaches to handling capabilities in programming languages is the object capability model. It was first proposed by Dennis and Horn [1] and is currently supported by secure programming languages such as E [11], Joe-E [9,10] and Caja [12,17]. In this model, a capability is regarded as a reference to an object, which may be used to invoke operations on that object. Such capabilities can only be obtained through a pre-existing chain of references. It provides modularity in code design and ensures reliable encapsulation in code implementation. However, this references-as-capabilities model does not provide an explicit authorisation mechanism or enforce security guarantees.

Java [3] is an object-oriented programming language. It has an access control model for guarding access to resources which relies on programmer discipline to insert security checks, which are then performed at runtime [2,8]. It makes use of a capability-like notion for access to some resources. For example, the class *FileOutputStream* in the Java Class Library (JCL) is like a capability to write to a file in the sense that permission-checking is performed in the constructor of the class. After the class has been instantiated, no further permission checks are required to use the operations of the class, like the *write* method. The Java access control model provides an approach to prevent confused deputy attacks [4] (e.g., unauthorised code accesses security-sensitive code by calling authorised code). However, it is not sufficient to track the propagation of capabilities, which means that Java does not guarantee that capabilities are not obtained and used by unauthorised code.

Capability-based access to Java resources was proposed recently by Hayes et al. [6] with the aim of preventing security flaws as well as tightening security management for access to resources both within JCL and Java applications. In this work, a capability can be viewed as an object with a restricted interface, which contains a set of operations that can be invoked by holders of the capability. In other words, a capability encapsulates what one can do with a resource. For example, a capability *OutCap* with a method *write* for output access to a stream is declared as follows:

```
capability OutCap { void write (int b); }
```

Access to this capability is restricted to code that has a corresponding permission, e.g. permission *write*. The philosophy behind capabilities is that code can only access a resource if it is given an explicit capability to do so: no other access is permitted. Once a capability is created, it has a more restrictive dynamic type than its implementing class and access to the full facilities of the implementing class (e.g., via down casting) is precluded. Thus, classes implementing capabilities are not directly accessible to users and hence cannot be overridden. In this way, only capabilities are open to exploit by untrusted code.

In the original approach proposed by Hayes et al. [6], no solutions were proposed for controlling the propagation of capabilities. The example in Listing 1 demonstrates how this can lead to capabilities escaping to unauthorised code. In the listing, the class *AuthorisedCode* is assumed to have the permissions required to use the file access capability (*FileAccessCap*), and to write to output streams (*OutCap*), while the class *UnauthorisedCode* does not. Because *UnauthorisedCode* does not have the permission to write to streams it cannot directly request the capability *OutCap*. However, this does not prevent the authorised code passing an instance of this capability to the unauthorised code as a parameter in a method call.

**Listing 1.** Capabilities may escape to unauthorised code

```java
public class AuthorisedCode {
    public static void main (String[] args) throws Exception {
        FileAccessCap fileAccess = new RandomAccessFileManager();
        UnauthorisedCode uc = new UnauthorisedCode();
        OutCap out = fileAccess.requestOutCap (filename);
        uc.use(out);
    }
}
public class UnauthorisedCode {
    public void use (OutCap out){
        out.write(temp);
    }
}
```

In practice, permissions granted to a class are parameterised using the targets on which a certain action is allowed. For example, a class that has the permission to write to files may either have: unlimited access to modify any file on the system (denoted "*"); access to modify only files in a particular directory (e.g. "dir/*"); or only a particular file, (e.g. "dir/a.txt") etc. In the original Capability model proposed in [6], there was no mechanism to limit a capability to be used on a particular target. For example, in Listing 1, either the capability *OutCap* is granted to a class, or it is not. There is no way to restrict *OutCap* to only be used to write to a particular file.

***Contributions.*** In this paper, our aim is to adapt capabilities to object-oriented programming languages in a way that (i) controls their propagation, and (ii) allows them to be parameterised in a way that limits their use to particular targets, so that they more closely correspond to the fine-grained permissions that are typically granted to classes.

We use the term "type capabilities" to analogize the term "object capabilities" that restrict capabilities at runtime. The key insight of our work is that, by providing explicit code-level authorisation via a user-defined policy file, we enforce a security guarantee at compile time that capabilities can only be obtained by authorised code. The main contributions are summarized as follows:

**Table 1.** Syntax of a Java-like language with parameterised capabilities

| | | |
|---|---|---|
| $CB$ | $::= \textbf{capability}\ cb(\widetilde{n})\ \textbf{extends}\ \overline{cb(\widetilde{n})}\ \{\overline{dec}\}$ | $(capabilities)$ |
| $C$ | $::= \textbf{class}\ \mathrm{c}(\widetilde{n})\ \textbf{extends}\ \mathrm{c}(\widetilde{n})\ \textbf{implements}\ \overline{cb(\widetilde{n})}\ \{\overline{\tau\, f}; \overline{M}\}$ | $(classes)$ |
| $M$ | $::= dec\{s\}$ | $(methods)$ |
| $dec$ | $::= m(\overline{\tau\, x})$ | $(declarations)$ |
| $s$ | $::= x = e \mid x.f = x \mid s;\ s \mid x.m(\overline{x}) \mid \textbf{if}\ x\ \textbf{then}\ s\ \textbf{else}\ s$ | $(statements)$ |
| e | $::= \mathrm{x} \mid \mathrm{x.f} \mid \textbf{new}\ c(\widetilde{\sigma}) \mid (\tau)\, e$ | $(expressions)$ |
| $\tau$ | $::= \textbf{int} \mid c(\widetilde{\sigma}) \mid cb(\widetilde{\sigma})$ | $(types)$ |
| $\sigma$ | $::= n \mid \kappa$ | $(parameters)$ |

∗ We present a type system to enforce the proper use of capabilities by type checking. Capabilities are regarded as types so that we can control the propagation of capabilities by controlling the visibility of their types.
∗ We provide a security guarantee statically at compile time, reducing the possibility of errors in code as well as runtime overhead. In particular, we guarantee that a method on an object can only be invoked if: (1) the static type of that object is granted to the calling class, and (2) the runtime type of the object is a subtype of its static type.
∗ We introduce capability types that are parameterised by strings, denoting the targets on which they can be used. It provides a finer-grained access control and identifies system resources more precisely.

***Organization.*** Section 2 gives the abstract syntax of parameterised capabilities for a Java-like core language. In Sect. 3, we illustrate how to enforce the proper use of capabilities statically by a type system and apply our approach on an example of Java file access. Section 4 presents the big-step operational semantics as well as the subject reduction theorem with a security guarantee before we conclude our paper and point out some future directions in Sect. 5.

## 2 A Java-Like Language with Parameterised Capabilities

Built on the model of capabilities described by Hayes et al. [6], a Java-like core language with parameterised capabilities is shown in Table 1. We choose a minimal set of features that still gives a Java-like feel to the language, i.e., classes, capabilities, inheritance, instance methods and fields, method override, dynamic dispatch and object creation.

In the syntax, the metavariables $cb$ and $c$ range over capability names and class names respectively; $f$ and $m$ range over field names and method names; $x$ ranges over variables, $n$ ranges over final string variables as type parameter names and $\kappa$ stands for string literals. Names for capabilities, classes, fields and variables are unique in their corresponding defining scopes. For simplicity, we use the notation $\overline{x}$ as a shorthand for the sequence $x_1; ...; x_n$, in which $n$ stands for the length of the sequence and we use semicolon to denote the concatenation of sequences. A sequence can be empty.

A *capability* $CB$, defined by a new keyword capability, consists of a set of method declarations and it may extend other capabilities. A *class* $C$ is composed of a sequence of fields $\overline{f}$ as well as a sequence of methods $\overline{M}$. We abbreviate sequences of pairs as $\overline{\tau\, f}$ for $\tau_1\, f_1; ...; \tau_n\, f_n$. A class has one super class and may implement a sequence of capabilities. Both capabilities and classes can be parameterised by a string parameter, which limits the targets that these capabilities or classes can be used on. The notation $\widetilde{n}$ (a sequence containing zero or one element) represents that the parameter $n$ is optional: if the parameter is absent then the capabilities or classes can be used on any target.

A *method* $M$ is a declaration *dec*, representing the method signature, followed by a method body $s$. A method *declaration dec* with the form of $m(\overline{\tau\ x})$ contains the method name $m$ as well as a list of parameters with types. We assume methods are not overloaded, that is, they are distinguished via their names rather than their signatures.

A *statement* $s$ is distinguished from an *expression* since it does not contain return values. It can be an assignment $x = e$, a field assignment $x.f = x$, a sequential composition $s; s$, a conditional choice **if** $x$ **then** $s$ **else** $s$ or a parameterised method invocation $x.m(\overline{x})$. An *expression* $e$ can be a variable $x$, a class field $x.f$ or a creation expression **new** $c(\widetilde{\sigma})$, which creates a new object of class $c$ with a type parameter $\widetilde{\sigma}$. It can also be a type cast $(\tau)\ e$, which stands for casting the type of the expression $e$ into type $\tau$. A *type* $\tau$ can be an integer **int**, a class type $c(\widetilde{\sigma})$ or a capability type $cb(\widetilde{\sigma})$. $\sigma$ is a string type parameter, which may be a final string variable or a string literal.

A *program* $P$ is a triple $(CT, CBT, s)$ of a class table, a capability table and a statement used as the program entry point. A class table $CT$ is a mapping from class names to class declarations. Similarly, a capability table $CBT$ is a mapping from capability names to capability declarations. For simplicity, the semantic rules in Sects. 3 and 4 are written with respect to a fixed program $P$ including a fixed class table $CT$ and a fixed capability table $CBT$. We assume that for every class $c$ (including class **Object**) appearing in $CT$, we have $c \in dom(CT)$ and we simply write"**class** $c(\widetilde{n})$..." to abbreviate $CT(c) =$ **class** $c(\widetilde{n})$.... Likewise, for every capability $cb$ appearing in $CBT$, we have $cb \in dom(CBT)$ and we use "**capability** $cb(\widetilde{n})$..." to abbreviate $CBT(cb) =$ **capability** $cb(\widetilde{n})$....

***Example.*** The parameterised capability for output access to a stream is given in Listing 2, as well as its implementing class and an application class.

Classes implementing a parameterised capability are also parameterised with the same *string* variable. Class $OutCapImp(n)$ in Listing 2 implements the capability $OutCap(n)$, which provides a method to write a file. We assume the implementing class always has at least one constructor (maybe by default) taking $n$ as its parameter, hence the instantiation of the class is restricted to the specific target file name. For example, in Listing 2, the class *Application* instantiates

**Listing 2.** The parameterised capability for file output stream

```
capability OutCap(n){ write(int b) }

class OutCapImp(n) implements OutCap(n){
    OutCapImp(n) { ... }; // open the file on path 'n'
    write(int b) { ... }
}
class Application{
    main() {
        OutCap("dir/A.txt") out = new OutCapImp("dir/A.txt")
    }
}
```

*OutCap* with the string ***"dir/A.txt"***, restricting the instance *out* to only write to the text file "A.txt" under the directory *dir*.

## 3 Static Semantics

In this section, we give a set of inference rules to formalize the static semantics of our type system. Based on a user-defined policy file, we control type visibility to avoid capabilities escaping to unauthorised code, and to restrict the targets that these capabilities can be used on.

**Table 2.** Subtyping rules

$$\tau <: \tau \qquad \frac{cb'_i(\widetilde{n}'_i) \in \overline{cb'(\widetilde{n}')} \quad \textbf{capability}\ cb(\widetilde{n})\ \textbf{extends}\ \overline{cb'(\widetilde{n}')}\ \{...\} \quad |\widetilde{\sigma}| = |\widetilde{n}| \quad \widetilde{\sigma}'_i = \mathrm{truncate}(\widetilde{\sigma}, |\widetilde{n}'_i|)}{cb(\widetilde{\sigma}) <: cb'_i(\widetilde{\sigma}'_i)}$$

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \qquad \frac{c \neq \textbf{Object} \quad \textbf{class}\ c(\widetilde{n})\ \textbf{extends}\ c'(\widetilde{n}')\ ...\ \{...\} \quad |\widetilde{\sigma}| = |\widetilde{n}| \quad \widetilde{\sigma}' = \mathrm{truncate}(\widetilde{\sigma}, |\widetilde{n}'|)}{c(\widetilde{\sigma}) <: c'(\widetilde{\sigma}')}$$

$$\frac{cb_i(\widetilde{n}_i) \in \overline{cb(\widetilde{n})} \quad \textbf{class}\ c(\widetilde{n})\ \textbf{extends}\ c'(\widetilde{n}')\ \textbf{implements}\ \overline{cb(\widetilde{n})}\ \{...\} \quad |\widetilde{\sigma}| = |\widetilde{n}| \quad \widetilde{\sigma}_i = \mathrm{truncate}(\widetilde{\sigma}, |\widetilde{n}_i|)}{c(\widetilde{\sigma}) <: cb_i(\widetilde{\sigma}_i)}$$

### 3.1 Subtyping Rules and Look up Functions

Subtyping rules are given in Table 2. They include the reflexive and transitive closure of the direct subclass (and subcapability) relations. If a class $c(\widetilde{n})$ implements a capability $cb(\widetilde{n})$, for all instantiations $\widetilde{\sigma}$ of parameter $\widetilde{n}$, $c(\widetilde{\sigma})$ is also a

**Table 3.** Look up functions on fields and methods

$$\text{fields}(\textbf{Object}) = \bullet \qquad \text{methods}(\textbf{Object}) = \bullet \qquad \text{methodsigs}(\textbf{Object}) = \bullet$$

$$\frac{\textbf{class}\ c(\widetilde{n})\ \textbf{extends}\ c'(\widetilde{n}')\ \{\overline{\tau_0\, f};\ \overline{m(\overline{\tau\, x})\{s\}}\} \quad c \neq \textbf{Object} \quad \widetilde{\sigma}' = \text{truncate}(\widetilde{\sigma}, |\widetilde{n}'|)}{\begin{array}{c}\text{fields}(c(\widetilde{\sigma})) = \text{fields}(c'(\widetilde{\sigma}')) \oplus \{\overline{f \mapsto \tau_{0}{}_{[\widetilde{n \backslash \sigma}]}}\} \\ \text{methods}(c(\widetilde{\sigma})) = \text{methods}(c'(\widetilde{\sigma}')) \oplus \{\overline{m \mapsto (c(\widetilde{\sigma}), (\overline{\tau_{[\widetilde{n \backslash \sigma}]}\ x})\{s_{[\widetilde{n \backslash \sigma}]}\})}\} \\ \text{methodsigs}(c(\widetilde{\sigma})) = \text{methodsigs}(c'(\widetilde{\sigma}')) \oplus \{\overline{m \mapsto (\overline{\tau_{[\widetilde{n \backslash \sigma}]}})}\}\end{array}}$$

$$\frac{\textbf{capability}\ cb(\widetilde{n})\ \textbf{extends}\ \overline{cb'(\widetilde{n}')}\ \{\overline{m(\overline{\tau\, x})}\} \quad \overline{\widetilde{\sigma}' = \text{truncate}(\widetilde{\sigma}, |\widetilde{n}'|)}}{\text{methodsigs}(cb(\widetilde{\sigma})) = \overline{\text{methodsigs}(cb'(\widetilde{\sigma}')) \oplus} \{\overline{m \mapsto (\overline{\tau_{[\widetilde{n \backslash \sigma}]}})}\}}$$

subtype of $cb(\widetilde{\sigma})$. Here, the sequence of the substitution value $\widetilde{\sigma}$ has the same length as the one of the string parameter $\widetilde{n}$ (denoted as $|\widetilde{\sigma}| = |\widetilde{n}|$). The rules use the function *truncate*, which shortens a sequence to the given length, to generalise instantiation to cases where classes (or capabilities) extend other classes (or capabilities) with fewer (i.e., zero) type parameters.

Table 3 gives the look up functions for accessing field and method definitions and declarations. The function *fields* is used to look up all field definitions (as a mapping from field names to types) in a class, including any field inherited from its superclass(es). The functions *methodsigs* and *methods* return mappings from method names to the method signatures and declarations (respectively) of methods in a type. Specifically, the function *methods* provides a tuple for each method, which is composed of the class type ($c(\widetilde{\sigma})$) that defined the method body as well as the method definition (of form $(\overline{\tau\ x})\{s\}$). We use the operator $\oplus$ to denote the addition of two mappings, where elements in the right-hand side mapping override (take precedence over) elements in the left-hand side mapping. The notation $t_{[\widetilde{n \backslash \sigma}]}$ denotes the substitution of any reference to type parameter $n$ for $\sigma$ within the preceding term $t$.

## 3.2 Well-Formedness and Typing Rules

A user-defined policy file $\Sigma$ is a mapping from a class $c$ (or a capability $cb$) to the set $\mathcal{G}$ of permissions (i.e., well-formed capabilities and well-formed classes) granted to that class (or capability). The transitive closure $\mathcal{G}^+$ of the set $\mathcal{G}$ can be found in Definition 1. It is defined with respect to $\preceq$, which is a partial order relation on strings. For example, "dir/A.txt" $\preceq$ "dir/*" and "dir/*" $\preceq$ "*".

**Definition 1 (Transitive Closure of $\mathcal{G}$).** *For the permission set $\mathcal{G}$ of a class (or a capability), class $c$, capability $cb$, types $\tau$ and $\tau'$, string literals $\kappa$ and $\kappa'$, and type parameter name $n$, the transitive closure of $\mathcal{G}$, denoted as $\mathcal{G}^+$, is defined as follows: (1) if $\tau \in \mathcal{G}$, then we have $\tau \in \mathcal{G}^+$; (2) if $\tau \in \mathcal{G}$ and $\tau <: \tau'$, then*

**Table 4.** Well-formedness rules for program, capabilities and classes

All rules are shown with respect to a fixed program $P = (CT, CBT, s)$.

$\boxed{\text{Program}}$

$$\frac{\mathcal{G} \vdash s \quad \forall\, c \in \mathrm{dom}(CT) \cdot \Sigma \vdash CT(c) \quad \forall\, cb \in \mathrm{dom}(CBT) \cdot \Sigma \vdash CBT(cb)}{\Sigma \vdash P}$$

$\boxed{\text{Capabilities}}$

$$\frac{\overline{\boldsymbol{\tau \in (\Sigma(cb))^+}}}{\Sigma\; cb(\widetilde{n}) \vdash m(\overline{\tau\, x})}$$

$$\frac{\begin{array}{c}\overline{\Sigma\; cb(\widetilde{n}) \vdash m_i(\overline{\tau\, x})} \quad \mathrm{referencedtypevars}(CBT(cb)) \subseteq \{\widetilde{n}\} \\ \left(\begin{array}{c}\forall\, cb_j'(\widetilde{n_j}') \in \overline{cb'(\widetilde{n}')}, \forall\, m_k \in \mathrm{dom}(\mathrm{methodsigs}(cb_j'(\widetilde{n_j}'))) \cdot \\ \textbf{capability}\, cb_j'(\widetilde{n_j}'') \ldots \wedge \boldsymbol{\widetilde{n_j}' = \mathbf{truncate}(\widetilde{n}, |\widetilde{n_j}''|)} \wedge cb_j'(\widetilde{n_j}') \not<: cb(\widetilde{n}) \wedge \\ \mathrm{methodsigs}(cb_j'(\widetilde{n_j}'))(m_k) = \mathrm{methodsigs}(cb(\widetilde{n}))(m_k)\end{array}\right)\end{array}}{\Sigma \vdash \textbf{capability}\; cb(\widetilde{n})\; \textbf{extends}\; \overline{cb'(\widetilde{n}')}\{\overline{m_i(\overline{\tau\, x})}\}}$$

$\boxed{\text{Classes}}$

$$\frac{\overline{\boldsymbol{\tau \in (\Sigma(c))^+}} \quad \Sigma(c)\; (\overline{\tau\, x}) \vdash s}{\Sigma\; c(\widetilde{n}) \vdash m(\overline{\tau\, x})\{s\}}$$

$$\frac{\begin{array}{c}\overline{\boldsymbol{\tau_0 \in (\Sigma(c))^+}} \quad \overline{\Sigma\; c(\widetilde{n}) \vdash m_j(\overline{\tau\, x})\{s\}} \quad \mathrm{referencedtypevars}(CT(c)) \subseteq \{\widetilde{n}\} \\ c \neq \textbf{Object} \quad \textbf{class}\; c'(\widetilde{n}''') \ldots \quad \boldsymbol{\widetilde{n}' = \mathbf{truncate}(\widetilde{n}, |\widetilde{n}'''|)} \quad c'(\widetilde{n}') \not<: c(\widetilde{n}) \\ \forall\, m_q \in \mathrm{dom}(\mathrm{methodsigs}(c'(\widetilde{n}'))) \cdot \mathrm{methodsigs}(c'(\widetilde{n}'))(m_q) = \mathrm{methodsigs}(c(\widetilde{n}))(m_q) \\ \forall\, f_q \in \mathrm{dom}(\mathrm{fields}(c'(\widetilde{n}'))) \cdot \mathrm{fields}(c'(\widetilde{n}'))(f_q) = \mathrm{fields}(c(\widetilde{n}))(f_q) \\ \left(\begin{array}{c}\forall\, cb_k(\widetilde{n_k}'') \in \overline{cb(\widetilde{n}'')}, \forall\, m_o \in \mathrm{dom}(\mathrm{methodsigs}(cb_k(\widetilde{n_k}''))) \cdot \\ \textbf{capability}\; cb_k(\widetilde{n_k}^i) \wedge \boldsymbol{\widetilde{n_k}'' = \mathbf{truncate}(\widetilde{n}, |\widetilde{n_k}^i|)} \wedge cb_k(\widetilde{n_k}'') \not<: c(\widetilde{n}) \wedge \\ \mathrm{methodsigs}(cb_k(\widetilde{n_k}''))(m_o) = \mathrm{methodsigs}(c(\widetilde{n}))(m_o)\end{array}\right)\end{array}}{\Sigma \vdash \textbf{class}\; c(\widetilde{n})\; \textbf{extends}\; c'(\widetilde{n}')\; \textbf{implements}\; \overline{cb(\widetilde{n}'')}\{\overline{\tau_0\, f};\; \overline{m_j(\overline{\tau\, x})\{s\}}\}}$$

*we have $\tau' \in \mathcal{G}^+$; (3) if $c(\kappa) \in \mathcal{G}$ and $\kappa' \preceq \kappa$, then we have $c(\kappa') \in \mathcal{G}^+$; (4) if $cb(\kappa) \in \mathcal{G}$ and $\kappa' \preceq \kappa$, then we have $cb(\kappa') \in \mathcal{G}^+$; (5) if $c(\text{“*”}) \in \mathcal{G}$, then we have $c(n) \in \mathcal{G}^+$; (6) if $cb(\text{“*”}) \in \mathcal{G}$, then we have $cb(n) \in \mathcal{G}^+$.*

***Example.*** Let $\mathcal{G} = \{OutCap(\text{“dir/*”})\}$, then we have $OutCap(\text{“dir/*”}) \in \mathcal{G}^+$. Because the relation on strings “dir/A.txt” $\preceq$ “dir/*” is satisfied, according to Definition 1, we have that $OutCap(\text{“dir/A.txt”}) \in \mathcal{G}^+$. Intuitively, if the user allows the code to write any file in the directory *dir* through $OutCap$, then it implicitly allows the code to write the specific text file *A.txt* in that directory.

A typing environment $\Gamma$ is a finite sequence of bindings $x : \tau$ of variables to types. For the variables in the domain of $\Gamma$, $\Gamma(x)$ is the type bound to the variable $x$. The typing judgement for an expression is of the form $\mathcal{G}\ \Gamma \vdash e : \tau$, which means the expression $e$ with type $\tau$ is well-formed in the typing environment $\Gamma$, according to the permission set $\mathcal{G}$ granted to the current executing class. The type judgement for a statement is of the form $\mathcal{G}\ \Gamma \vdash s$, which is used for checking whether a statement $s$ is well-formed or not according to the permission set $\mathcal{G}$.

Well-formedness rules for program, capabilities and classes are shown in Table 4. A program $P$, composed of classes and capabilities, is well-formed based on the user-defined policy file $\Sigma$ (denoted as $\Sigma \vdash P$) only if all classes and capabilities are well-formed (denoted as $\forall\ c \in \text{dom}(CT) \cdot \Sigma \vdash CT(c)$ and $\forall\ cb \in \text{dom}(CBT) \cdot \Sigma \vdash CBT(cb)$ respectively), as well as the entry point statement of the program is well-formed (denoted as $\mathcal{G} \vdash s$, and $\mathcal{G}$ stands for the permission set granted to the class containing the entry point statement).

The other two group rules in Table 4 are used for checking the well-formedness of capabilities and classes respectively. Traditional well-formedness checking considers that statements of the method body are well-formed, signatures of overriding methods are compatible and there are no cycles in the transitive closure of extension relations. It also checks that the only type variable referenced inside a class or capability is the class or capability parameter $n$ (e.g., through the function *referencedtypevars*). Besides these traditional criteria, we add the following additional criteria (highlighted in **bold**) which state a class (or capability) is well-formed only if:

$*$ types of parameters in all method signatures are granted;
$*$ types of all fields in the class are granted;
$*$ capability parameters (or class parameters) should remain the same in extension (or implementation) relations.

If a type $\tau$ is granted to a capability $cb$ (or a class $c$) based on the user-defined policy file $\Sigma$, then we have $\tau \in \Sigma(cb)$ (or $\tau \in \Sigma(c)$).

The typing rules for expressions and statements are shown in Table 5. As before, we highlight our additions in **bold**. The first group of rules are used for expressions. We can obtain the types of variables directly from the typing environment $\Gamma$ according to the first rule (VAR) and look up the types of fields using rule (FID). Types of variables and fields are granted to the current executing class if they are given in the set $\mathcal{G}$, which stands for the set of permissions granted to the current executing class based on the user-defined policy file. Note that we leave the situation that the type of the expression is a subtype of a variable or a field to be covered by the subsumption rule (SUB). The rule (NEW) for the object creation may create an object with the parameter $\widetilde{\sigma}$ to instantiate the type parameter. The derived type of the expression should be the same as the class type (i.e., $c(\widetilde{\sigma})$). The last rule (CAST) in the first group for expressions is used to deal with the type casting in the object-oriented programming languages, which allows an expression to be cast to a granted subtype.

The next group of typing rules in Table 5 covers the rules for statements. Rule (AGN) and rule (FIDAGN) for variable assignment and field assignment are typed by ensuring that the derived type of the expression is the same as the type of the variable $x$ or the field $f$. For the rule (IF), if the variable $x$ has the type **int**, and statements $s_1$ and $s_2$ are well-formed under the typing environment

**Table 5.** Typing rules for expressions and statements

All rules are shown with respect to a fixed program $P = (CT, CBT, s)$.

$\boxed{\text{Expressions}}$

$$\text{(VAR)}\ \frac{\Gamma(x) = \tau \quad \boldsymbol{\tau \in \mathcal{G}^+}}{\mathcal{G}\,\Gamma \vdash x : \tau} \qquad \text{(SUB)}\ \frac{\mathcal{G}\,\Gamma \vdash e : \tau' \quad \tau' <: \tau \quad \boldsymbol{\tau \in \mathcal{G}^+}}{\mathcal{G}\,\Gamma \vdash e : \tau}$$

$$\text{(NEW)}\ \frac{\boldsymbol{c(\widetilde{\sigma}) \in \mathcal{G}^+}}{\mathcal{G}\,\Gamma \vdash \textbf{new}\ c(\widetilde{\sigma}) : c(\widetilde{\sigma})} \qquad \text{(CAST)}\ \frac{\mathcal{G}\,\Gamma \vdash e : \tau' \quad \boldsymbol{\tau \in \mathcal{G}^+}}{\mathcal{G}\,\Gamma \vdash (\tau)\ e : \tau}$$

$$\text{(FID)}\ \frac{\Gamma(x) = \tau_0 \quad \boldsymbol{\tau_0 \in \mathcal{G}^+} \quad \text{fields}(\tau_0)(f) = \tau \quad \boldsymbol{\tau \in \mathcal{G}^+}}{\mathcal{G}\,\Gamma \vdash x.f : \tau}$$

$\boxed{\text{Statements}}$

$$\text{(AGN)}\ \frac{\Gamma(x) = \tau \quad \mathcal{G}\,\Gamma \vdash e : \tau}{\mathcal{G}\,\Gamma \vdash x = e} \qquad \text{(SEQ)}\ \frac{\mathcal{G}\,\Gamma \vdash s_1 \quad \mathcal{G}\,\Gamma \vdash s_2}{\mathcal{G}\,\Gamma \vdash s_1;\ s_2}$$

$$\text{(FIDAGN)}\ \frac{\Gamma(x) = \tau_0 \quad \boldsymbol{\tau_0 \in \mathcal{G}^+} \quad \text{fields}(\tau_0)(f) = \tau \quad \mathcal{G}\,\Gamma \vdash y : \tau}{\mathcal{G}\,\Gamma \vdash x.f = y}$$

$$\text{(IF)}\ \frac{\Gamma(x) = \textbf{int} \quad \mathcal{G}\,\Gamma \vdash s_1 \quad \mathcal{G}\,\Gamma \vdash s_2}{\mathcal{G}\,\Gamma \vdash \textbf{if}\ x\ \textbf{then}\ s_1\ \textbf{else}\ s_2}$$

$$\text{(CALL)}\ \frac{\Gamma(x) = \tau_0 \quad \boldsymbol{\tau_0 \in \mathcal{G}^+} \quad \text{methodsigs}(\tau_0)(m) = (\overline{\tau_p}) \quad \overline{\mathcal{G}\,\Gamma \vdash y : \tau_p}}{\mathcal{G}\,\Gamma \vdash x.m(\overline{y})}$$

$\Gamma$ and the permission set $\mathcal{G}$, then the whole statement is also well-formed under $\Gamma$ and $\mathcal{G}$. To type a sequential composition, each statement needs to be typed in the typing environment $\Gamma$ under the permission set $\mathcal{G}$, which is shown in rule (SEQ). The last rule (CALL) looks up the method signature and checks whether the types of the method arguments (i.e., the types of $\overline{y}$) are the same as the ones of the method parameters (e.g., $\overline{\tau_p}$).

## 3.3 Example Revisited

We revisit the example of Java file output access used in Sect. 2 to demonstrate the applicability of the proposed model. The capability *OutCap* and its implementation class are given in Listing 2. A combined capability *InOutCap* for both input and output access, and an application class are given in Listing 3.

The class *Application* is granted the type *InOutCap*("dir/*") and the type *OutCapImp*("dir/A.txt") as permissions by the user, thus we have that:

$$\mathcal{G} = \Sigma(\mathit{Application}) = \{\mathit{InOutCap}(\text{“dir/*”}), \mathit{OutCapImp}(\text{“dir/A.txt”})\}$$

**Listing 3.** Application class using file stream capabilities

```
capability InOutCap(n) extends OutCap(n){
    write(int b);
    read()
}
//grant: InOutCap("dir/*") and OutCapImp("dir/A.txt")
class Application{
    main() {
        OutCap("dir/A.txt") out = new OutCapImp("dir/A.txt");
        InOutCap("dir/A.txt") inOut = (InOutCap("dir/A.txt")) out;
        OutCap("*") out2 = (OutCap("*")) inOut //invalid
    }
}
```

We check the following three statements based on our typing rules and illustrate why the third statement in the *main* method is invalid. The first statement creates an instance of capability *OutCap*("dir/A.txt"), which passes the type checking using rules (NEW), (SUB) and (AGN) from Table 5. The inference steps are illustrated below. According to Definition 1, we have both *OutCap*("dir/A.txt") $\in \mathcal{G}^+$ and *OutCapImp*("dir/A.txt") $\in \mathcal{G}^+$.

$$\dfrac{\dfrac{\dfrac{\mathit{OutCapImp}(\text{“dir/A.txt”}) \in \mathcal{G}^+}{\mathcal{G}\ \Gamma \vdash \mathit{new}\ \mathit{OutCapImp}(\text{“dir/A.txt”}) : \mathit{OutCapImp}(\text{“dir/A.txt”})} \quad \mathit{OutCapImp}(\text{“dir/A.txt”}) <: \mathit{OutCap}(\text{“dir/A.txt”}) \quad \mathit{OutCap}(\text{“dir/A.txt”}) \in \mathcal{G}^+}{\mathcal{G}\ \Gamma \vdash \mathit{new}\ \mathit{OutCapImp}(\text{“dir/A.txt”}) : \mathit{OutCap}(\text{“dir/A.txt”}) \quad \Gamma(\mathit{out}) = \mathit{OutCap}(\text{“dir/A.txt”})}}{\mathcal{G}\ \Gamma \vdash \mathit{out} = \mathit{new}\ \mathit{OutCapImp}(\text{“dir/A.txt”})}$$

The second statement casts the type of the instance *out* to capability *InOutCap* with the parameter "dir/A.txt". The following inference steps are given based on the rules (VAR), (CAST) and (AGN) in Table 5. Also, based on Definition 1, we can deduce that *InOutCap*("dir/A.txt") $\in \mathcal{G}^+$.

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathit{out}) = \mathit{OutCap}(\text{“dir/A.txt”}) \quad \mathit{OutCap}(\text{“dir/A.txt”}) \in \mathcal{G}^+}{\mathcal{G}\ \Gamma \vdash \mathit{out} : \mathit{OutCap}(\text{“dir/A.txt”}) \quad \mathit{InOutCap}(\text{“dir/A.txt”}) \in \mathcal{G}^+}}{\mathcal{G}\ \Gamma \vdash (\mathit{InOutCap}(\text{“dir/A.txt”}))\mathit{out} : \mathit{InOutCap}(\text{“dir/A.txt”}) \quad \Gamma(\mathit{inOut}) = \mathit{InOutCap}(\text{“dir/A.txt”})}}{\mathcal{G}\ \Gamma \vdash \mathit{inOut} = (\mathit{InOutCap}(\text{“dir/A.txt”}))\mathit{out}}$$

However, the third statement cannot pass the type checking as we **cannot** deduce *OutCap*("*") $\in \mathcal{G}^+$, based on Definition 1 and the types granted to *Application*.

Through controlling the type visibility, we avoid capabilities escaping to unauthorised code and restrict the targets that capabilities can access, based on a user-defined policy file. Revisiting and applying our approach to the motivating example in Sect. 1, we can find that the *UnauthorisedCode* is granted neither the type *OutCap*("dir/B.txt") nor the type *OutCap*("dir/*"), thus the declaration itself of class *UnauthorisedCode* cannot pass the well-formedness check at compile time.

## 4 Dynamic Semantics

In this section, we present the dynamic semantics and security-related subject reduction theorem of the type system. We show that the type-correctness of the runtime state and the security invariant are preserved over the evaluation of expressions and statements.

### 4.1 Operational Semantics

The dynamic semantics is devised using the big-step style operational semantics. We start by adding some additional notations to represent runtime values and states as follows.

$$\begin{aligned} e &::= ... \mid v \\ v &::= l^{c(\widetilde{\kappa})} \mid \mathit{null} \mid \mathit{num} \end{aligned}$$

$v$ is a runtime value, denoting the result of evaluating an expression. It can be an integer $\mathit{num}$, a location $l$ labeled with its dynamic type $c(\widetilde{\kappa})$, or $\mathit{null}$.

We use $S$ to stand for the stack, mapping from local variables to values (e.g., $S(x) = v$ denotes that the variable $x$ contains the value $v$), and $H$ represents the heap, mapping from locations and fields to values (e.g., $H(l^{c(\widetilde{\kappa})})(f) = v$ describes that the field $f$ of class $c(\widetilde{\kappa})$ which is allocated at the location $l$ on the heap contains the value $v$). The notation $A$ denotes a list recording method invocation actions taken by the program. Each action is recorded as a quadruple $(c, c_i(\widetilde{\kappa_i}), \tau_r, m)$, in which $c$ stands for the class name of the current calling class, $c_i(\widetilde{\kappa_i})$ is the class that contains the implementation of the method we are calling, $\tau_r$ is the runtime type of the object on which we are calling the method and $m$ is the method name. The evaluation rule for expressions is of the form $c \vdash \langle e \mid S\ H\ A\rangle \rightarrow \langle v \mid S'\ H'\ A'\rangle$, which represents that in a given class $c$, an expression $e$ can make a transition into a value $v$, and the evaluation of their side effects is shown on the stack, heap and action list. The evaluation rule for statements is in the form of $c \vdash \langle s \mid S\ H\ A\rangle \rightarrow \langle S'\ H'\ A'\rangle$, which denotes that statements are evaluated for their side effects only.

We proceed with a detailed explanation of the semantic rules for expressions and statements in Table 6. The notation $S[x \mapsto v]$ represents the update of the stack $S$ that maps the variable $x$ to the value $v$, which is similar with the update of the heap $H$ with the form of $H[l^{c(\widetilde{\kappa})} \mapsto [f \mapsto v]]$. We use notations $\mathit{dom}(S)$

**Table 6.** Dynamic semantics for expressions and statements

$\boxed{\text{Expressions}}$

$$\text{(T-VAL)}\ \frac{S(x) = v}{c \vdash \langle x \mid S\ H\ A\rangle \to \langle v \mid S\ H\ A\rangle} \qquad \text{(T-LOAD)}\ \frac{S(x) = l^{c'(\widetilde{\kappa})} \quad H(l^{c'(\widetilde{\kappa})})(f) = v}{c \vdash \langle x.f \mid S\ H\ A\rangle \to \langle v \mid S\ H\ A\rangle}$$

$$\text{(T-CAST)}\ \frac{c \vdash \langle e \mid S\ H\ A\rangle \to \langle l^{c'(\widetilde{\kappa})} \mid S\ H'\ A\rangle \quad c'(\widetilde{\kappa}) <: \tau}{c \vdash \langle (\tau)e \mid S\ H\ A\rangle \to \langle l^{c'(\widetilde{\kappa})} \mid S\ H'\ A\rangle}$$

$$\text{(T-NEW)}\ \frac{l^{c'(\widetilde{\kappa})} \notin dom(H) \quad \text{dom}(\text{fields}(c'(\widetilde{\kappa}))) = \overline{f} \quad H' = H, \{l^{c'(\widetilde{\kappa})} \mapsto [\overline{f \mapsto \text{null}}]\}}{c \vdash \langle \textbf{new}\ c'(\widetilde{\kappa}) \mid S\ H\ A\rangle \to \langle l^{c'(\widetilde{\kappa})} \mid S\ H'\ A\rangle}$$

$\boxed{\text{Statements}}$

$$\text{(T-AGN)}\ \frac{c \vdash \langle e \mid S\ H\ A\rangle \to \langle v \mid S\ H'\ A\rangle}{c \vdash \langle x = e \mid S\ H\ A\rangle \to \langle S[x \mapsto v]\ H'\ A\rangle}$$

$$\text{(T-FLD)}\ \frac{c \vdash \langle y \mid S\ H\ A\rangle \to \langle v \mid S\ H\ A\rangle \quad S(x) = l^{c'(\widetilde{\kappa})} \quad l^{c'(\widetilde{\kappa})} \in \text{dom}(H)}{c \vdash \langle x.f = y \mid S\ H\ A\rangle \to \langle S\ H[l^{c'(\widetilde{\kappa})} \mapsto H(l^{c'(\widetilde{\kappa})})[f \mapsto v]]\ A\rangle}$$

$$\text{(T-CALL)}\ \frac{\begin{array}{c} S(x) = l^{c'(\widetilde{\kappa})} \quad S(\overline{y}) = \overline{v} \quad \text{methods}(c'(\widetilde{\kappa}))(m) = (c_i(\widetilde{\kappa_i}), (\overline{\tau\ z})\{s\}) \\ c_i \vdash \langle s \mid S_0\ H\ A_0\rangle \to \langle S'\ H'\ A'\rangle \quad S_0 = \{\overline{z \mapsto v}\} \quad A_0 = A, \{c, c_i(\widetilde{\kappa_i}), c'(\widetilde{\kappa}), m\}\end{array}}{c \vdash \langle x.m(\overline{y}) \mid S\ H\ A\rangle \to \langle S\ H'\ A'\rangle}$$

$$\text{(T-THEN)}\ \frac{S(x) = v \quad v > 0 \quad c \vdash \langle s_1 \mid S\ H\ A\rangle \to \langle S_1\ H_1\ A_1\rangle}{c \vdash \langle \textbf{if}\ x\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid S\ H\ A\rangle \to \langle S_1\ H_1\ A_1\rangle}$$

$$\text{(T-ELSE)}\ \frac{S(x) = v \quad v \leq 0 \quad c \vdash \langle s_2 \mid S\ H\ A\rangle \to \langle S_2\ H_2\ A_2\rangle}{c \vdash \langle \textbf{if}\ x\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid S\ H\ A\rangle \to \langle S_2\ H_2\ A_2\rangle}$$

$$\text{(T-SEQ)}\ \frac{c \vdash \langle s_1 \mid S\ H\ A\rangle \to \langle S_1\ H_1\ A_1\rangle \quad c \vdash \langle s_2 \mid S_1\ H_1\ A_1\rangle \to \langle S_2\ H_2\ A_2\rangle}{c \vdash \langle s_1;\ s_2 \mid S\ H\ A\rangle \to \langle S_2\ H_2\ A_2\rangle}$$

and $dom(H)$ to stand for the domain of the stack $S$ and the heap $H$, respectively. The notation $H' = H, \{l^{c(\widetilde{\kappa})} \mapsto ...\}$ is used to represent an extension of heap $H$ where $l^{c(\widetilde{\kappa})} \notin \text{dom}(H)$, and $A' = A, \{...\}$ is used for the extension of list $A$.

The first two rules (T-VAL) and (T-LOAD) evaluate variables and fields from the stack and the heap respectively. The rule (T-CAST) describes the downcasting between objects or capability variables if the runtime type is a subtype of the type to be converted to. The last rule (T-NEW) is used for an object creation, which extends the heap with the new object. All fields are initially set to null.

The remaining rules are used for statement evaluations. The rules (T-AGN) and (T-FLD) are used to update the stack and the heap respectively. Method invocation in (T-CALL) dynamically looks up the target method to be called based on the dynamic type of the object. A record of the method invocation action will be added into the action list $A$. The rules (T-THEN) and (T-ELSE) describe the transitions performed by the conditional choice. In particular, the rule (T-THEN) accounts for the case where the condition is true (indicated by the value of the variable $x$ is greater than zero); whereas the rule (T-ELSE) accounts for the case where it is false. The last rule (T-SEQ) is used for evaluating the sequential composition of two statements in order, which means that statement $s_2$ is evaluated based on the output configuration of statement $s_1$.

## 4.2 Subject Reduction

In this section, we prove that well-formed programs are safe over subject reduction [13], which means that the type-correctness of the program state and the security invariant are preserved under evaluations.

$$\text{(CORR)}\quad \frac{\begin{array}{c}\left(\begin{array}{c}\forall x \in \mathrm{dom}(\Gamma), \tau \cdot \Gamma(x) = \tau \wedge \boldsymbol{\tau \in (\Sigma(c))^{+}} \Longrightarrow \\ x \in \mathrm{dom}(S) \wedge \exists v \cdot S(x) = v \wedge H \vdash v : \tau\end{array}\right) \\ \left(\begin{array}{c}\forall l^{c'(\widetilde{\kappa})} \in \mathrm{dom}(H), f, \tau \cdot \mathrm{fields}(c'(\widetilde{\kappa}))(f) = \tau \Longrightarrow \\ f \in \mathrm{dom}(H(l^{c'(\widetilde{\kappa})})) \wedge \exists v \cdot H(l^{c'(\widetilde{\kappa})})(f) = v \wedge H \vdash v : \tau\end{array}\right) \\ \Sigma(c)\ \Gamma \Vdash A\end{array}}{\Sigma(c)\ \Gamma \vdash S\ H\ A}$$

An additional rule (CORR) is given to illustrate the correspondence between the typing environment $\Gamma$ of type system and the configuration, including stack $S$ and heap $H$, under the user-defined policy file $\Sigma$ and the current executing class $c$. It requires that for every variable $x$ in $\Gamma$, a value $v$ exists for variable $x$ on the stack $S$ such that $v$ is type-correct to $\Gamma(x)$. Similarly, for every object on the heap, both the fields present, and their values, must match the object's type information. Lastly, the security invariant on the action list $A$ must be maintained.

**Definition 2 (Security Invariant).** *For the action list $A$, user-defined policy file $\Sigma$, calling class $c$, class $c_i(\widetilde{\kappa_i})$ containing the method body that is called, runtime type $c'(\widetilde{\kappa})$ of the object on which the method is called and method name $m$, the security invariant (represented as $\Sigma(c)\ \Gamma \Vdash A$) says that:*

$$\begin{array}{c}\forall (c, c_i(\widetilde{\kappa_i}), c'(\widetilde{\kappa}), m) \in A, \exists \tau \cdot \tau \in (\Sigma(c))^{+} \wedge c'(\widetilde{\kappa}) <: \tau\ \wedge \\ c'(\widetilde{\kappa}) <: c_i(\widetilde{\kappa_i}) \wedge \mathrm{methodsigs}(\tau)(m) = \mathrm{methodsigs}(c_i(\widetilde{\kappa_i}))(m)\end{array}$$

The security invariant says that for all method invocation actions in $A$, there exists a type $\tau$ granted to the current calling class $c$, of which the runtime type $c'(\widetilde{\kappa})$ is a subtype. Also, the runtime type is a subtype of the type of class $c_i(\widetilde{\kappa_i})$ which contains the implementation of the method we invoked and the method signature looked up based on $\tau$ and $c_i(\widetilde{\kappa_i})$ should be the same. It provides a guarantee that each well-formed method invocation action in the action list $A$

can only use the types (i.e., capabilities and classes) granted to its invoking class based on the user-defined policy file, restricting capability types only to authorised code.

In order to accommodate runtime values, we add three more rules to extend our static inference rules for checking the runtime values are type correct in the context of heap $H$.

$$\text{(NULL)}\quad H \vdash \mathit{null} : \tau \qquad\qquad \text{(LOC)}\quad \frac{l^{c(\widetilde{\kappa})} \in \text{dom}(H) \quad c(\widetilde{\kappa}) <: \tau}{H \vdash l^{c(\widetilde{\kappa})} : \tau}$$

$$\text{(NUM)}\quad H \vdash \mathit{num} : \textbf{int}$$

The preservation theorem for subject reduction is given in Theorem 1, which presents the preservation of well-formedness and security invariant on statements. Preservation for expressions is trivial as expressions only look up values from well-formed stack or heap, thus we omit it. Theorem 1 can be proved by structural induction on the semantic derivation.

**Theorem 1 (Preservation).** *For any typing environment $\Gamma$, stack $S$, heap $H$, action list $A$, statement $s$, current executing class $c$, user-defined policy file $\Sigma$ and the well-formed program $P$:*

$$\left.\begin{array}{r} \Sigma \vdash P \\ \Sigma(c)\ \Gamma \vdash s \\ \Sigma(c)\ \Gamma \vdash S\ H\ A \\ c \vdash \langle s|S\ H\ A\rangle \rightarrow \langle S'\ H'\ A'\rangle \end{array}\right\} \Longrightarrow \Sigma(c)\ \Gamma \vdash S'\ H'\ A'$$

## 5 Conclusion and Future Work

Existing authorisation mechanisms used in programming languages like Java are not effective in controlling interactions between different parts of code within the same process. In this paper, we tackled the problem of adapting capabilities to programming languages for providing authorisation to code. We regarded capabilities as types and presented a statically-checked type system to enforce the proper use of capabilities by controlling the type visibility at compile time, providing a security guarantee that restricts capabilities (i.e., the access to resources) only to authorised code. We also introduced parameterised capability types to provide a finer-grained access control and to identify system resources more precisely. We applied our model on file-access examples.

Future directions for our research include building a prototype implementation of the type system, and validating its usability by applying it to real-world case studies. Other possible directions include extending the language with even richer parameterisation to increase its expressiveness, and adding more language features (e.g., method overloading, return values and exceptions) to improve the quality of our formalism.

**Acknowledgements.** The research presented here is supported by Australian Research Council Linkage Grant LP140100700 in collaboration with Oracle Labs Australia.

## References

1. Dennis, J.B., van Horn, E.C.: Programming semantics for multiprogrammed computations. Commun. ACM **9**(3), 143–155 (1966)
2. Gong, L., Ellison, G., Dageforde, M.: Inside Java 2 Platform Security: Architecture, API Design, and Implementation, Second edn. Addison Wesley, Boston (2003)
3. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D.: The Java language specification: Java SE 10 edition, 20 February 2018. https://docs.oracle.com/javase/specs/jls/se10/html/index.html. Accessed 27 Sept 2018
4. Hardy, N.: The confused deputy: (or why capabilities might have been invented). SIGOPS Oper. Syst. Rev. **22**(4), 36–38 (1988)
5. Hardy, N.: KeyKOS architecture. Oper. Syst. Rev. **19**(4), 8–25 (1985)
6. Hayes, I.J., Wu, X., Meinicke, L.A.: Capabilities for Java: secure access to resources. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 67–84. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_4
7. Kozen, D.: Language-based security. In: Kutyłowski, M., Pacholski, L., Wierzbicki, T. (eds.) MFCS 1999. LNCS, vol. 1672, pp. 284–298. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48340-3_26
8. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: Going beyond the sandbox: an overview of the new security architecture in the Java development Kit 1.2. In: Proceedings of 1st USENIX Symposium on Internet Technologies and Systems, USITS 1997. USENIX (1997)
9. Mettler, A., Wagner, D.: The Joe-E language specification, version 1.0. Technical report EECS-2008-91, University of California, Berkeley, August 2008
10. Mettler, A., Wagner, D., Close, T.: Joe-E: a security-oriented subset of Java. In: Proceedings of the Symposium on Network and Distributed System Security, NDSS 2010. The Internet Society (2010)
11. Miller, M.S.: Robust composition: towards a unified approach to access control and concurrency control. Ph.D. thesis, Johns Hopkins University (2006)
12. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: safe active content in sanitized JavaScript, 7 June 2008. https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/google-caja/caja-spec-2008-06-07.pdf. Accessed 27 Sept 2018
13. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
14. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
15. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proc. IEEE **63**(9), 1278–1308 (1975)
16. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: a fast capability system. In: Proceedings of 17th ACM Symposium on Operating System Principles, SOSP 1999, pp. 170–185. ACM (1999)
17. Google Caja Team: Google-Caja: a source-to-source translator for securing JavaScript-based web. http://code.google.com/p/google-caja/. Accessed 27 Sept 2018