A Formal Framework for Access Rights Analysis

Xin Li
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Japan
li-xin@jaist.ac.jp

Hua Vy Le Thanh
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Japan
thanhvy.hua@jaist.ac.jp

A stack-based access control mechanism is employed in an attempt to prevent untrusted codes from accessing protected resources in distributed application systems, such as Java-centric web applications and Microsoft .NET framework. Such an access control mechanism is enforced at runtime by stack inspection that inspects methods in the current call stack for granted permissions. Nowadays practiced approaches to generating policy files for an application are still manually done by developers based on domain-specific knowledges and testing, due to overwhelming technical challenges involved and engineering efforts in the automation.

This paper presents a formal framework of access rights analysis for Java applications, and algorithms of both policy generation and checking. The analysis of policy generation automatically generates access control policies for the given program that necessarily ensure the program to pass stack inspections. The analysis of policy checking takes as input a policy file and determines whether access control in the concerned domain always succeed or may fail. The answer can either help detect redundant inspection points or refine the given policies. All of the analysis algorithms are novelly designed by conditional weighted pushdown systems, in an attempt to achieve a high level of precision in the literature.

1 Introduction

In modern Web platforms, such as Java-centric web applications or Microsoft .NET framework, applications comprise components from different origins with diverse levels of trust. A stack-based access control mechanism is employed in an attempt to prevent untrusted codes from accessing protected resources. Access control policies are expressed in terms of permissions (e.g., a permission can be “writing the file C:/students_grades.txt”) that are granted to codes grouped by different domains (e.g., www.jaist.ac.jp/faculty). Developers can set checkpoints through the Java API CheckPermission(Permission) in their programs, and access control is enforced dynamically at runtime by the well-known security mechanism stack inspection. When stack inspection is triggered, the current call stack will be inspected in a top-down manner to see whether methods in the stack is granted the required permission until a privileged method is found. A caller can be marked as being privileged, and the stack inspection stops at such a caller. If all callers have the specified permission, access control is passed and stack inspection returns quietly, and the program execution will be interrupted immediately otherwise.

Access control is often the first step to protect safety-critical systems. However, for practiced approaches, nowadays policy files are still generated manually by developers based on domain-specific knowledge, and measured by a trial-and-error testing as to whether the policy file allows the application to run properly. Since testing cannot cover all program runtime behaviors, the application could malfunction due to accidental authorization failures given a misconfigured policy. On the other hand, if a security policy is too conservative, i.e., some codes are granted permissions than necessary, it violates the PLP

*Corresponding author.
(Principle of Least Privilege), and the codes become vulnerable points for malicious attacks. Moreover, from a practical perspective, such runtime inspection may cause a considerable runtime overhead. If access control at some checkpoints always succeed at runtime, the runtime overhead can be reduced by removing such redundant checkpoints.

**Example 1 (Semantics of Java Stack Inspection)** Consider the code snippet in Fig. 1 that we borrow from [7] and modify to explicitly show the control flows to checkpoints of stack inspections.

In Fig. 1, there are two library classes `Lib` and `Priv`, and two application classes `Faculty` and `Student`. At the beginning of each program execution, the Java VM assigns all classes along with their related methods to a set of permissions specified by a security policy. At runtime, the two clients will require to connect to their corresponding domains by creating a socket (Line 35 and 41, respectively). Such a request will trigger stack inspection at Line 13 by the API `checkPermission(Permission)` from the class `AccessController` with taking a single parameter of type `Permission` or its subclasses. `Student` is required to posses the permission `perm_s = "SocketPermission(jaist.ac.jp/student:8080, connect)"` and `Faculty` is required to hold the permissions `perm_f = "SocketPermission(jaist.ac.jp/faculty:8080, connect)"`.

Moreover, the socket construction process should be logged in C:/log.txt by the system for later observation. A file access permission `perm_a = "FilePermission(C:/log.txt, write)"` is required on the system to perform this task, and another stack inspection is triggered at Line 29. But note that `Student` and `Faculty` reside on the current call stack but should not posses `perm_a`. To avoid authorization failures while logging, `Lib` invokes the API `doPrivileged (Line 8)` from the class `AccessController` with passing an instance `op` of `Priv`, and by Java semantics, `op.run()` will be executed with full permissions granted to its caller, and the stack inspection stops at `createSocket` without requiring `perm_a` from clients of `Lib`.

As shown in Example 1, analysis on security policy is centered around reasoning permissions. A permission analysis demands points-to analysis for identifying objects of `Permission` type, and string analysis for resolving string parameters of relevant security APIs. Especially, since string operations are prevalent, e.g., string variables may be created through concatenation (Line 24, 34, 40), substring operation (Line 24), case conversion (Line 34, 40), etc., string analysis plays an important role. On one hand, it is challenging to design a precise and scalable algorithm for either of these analysis. On the other hand, it is not clear how to utilize these analysis results seamlessly in access rights analysis, and we are aware of no such investigations.

This paper presents a formal framework of access rights analysis for Java applications, which includes both policy generation and checking. The analysis of policy generation automatically generates access control policies for the given program that necessarily ensure the program to pass stack inspections. The analysis of policy checking takes as input a policy file and determines whether access control in the concerned domain always succeed or may fail. The answer can either help detect redundant inspection points or refine the given policies. All of our analysis algorithms are designed in the framework of conditional weighted pushdown systems (CWPDSs) [13] that is capable of reasoning properties over the stack.

Our analysis framework has the following features.

- First, we define an abstraction over the calling contexts that are uniformly adapted in context-sensitive string and points-to analysis, as a bridge for different analysis modules in the same analysis framework. The approach to identifying permission requirements for each stack inspection point are also based on such abstract calling contexts.

- Second, instead of conducting analysis on call graphs as usual, we model the analysis problem in terms of a kind of context-sensitive call graph, taking into account the dynamic features of
Java languages. This consideration is crucial to the precision of access rights analysis, because privileged actions are also dynamically invoked, as discussed in Example 2.

- Moreover, we systematically formalize the problem of policy generation and checking for Java programs, with taking into account the impact of string and points-to analysis, and the identification of permission requirements that was often ignored and assumed to be known beforehand in
existing works.

- Last but not least, instead of conducting analysis on context-sensitive call graphs alone, we also unify the program model over dependency graphs. The combined model enables us to precisely infer permission requirements at each checkpoint of stack inspection and generate policies.

The reason why call graph does not suffice for the analysis is because Java objects (here we are concerned with objects of the Permission type) can be created and referred to anywhere in the program, by either (i) accessing the heap, i.e., field access, or by (ii) passing and returning parameters to method calls that are finished before stack inspection. In either case, the data flow of permission objects is beyond the scope of the current call stack that is inspected by access control.

To the best of our knowledge, the only existing analysis that attempted to automatically identify authorization requirements and generate access control policies for Java applications is \[7\] from IBM, which is protected under the US patents. The analysis for the first time explored the impact of string analysis on the precision of analysis results. A modular analysis algorithm is proposed to achieve the context-insensitive slicing algorithm when identifying permission requirements.

The rest of the paper is organized as follows. Section 2 recalls conditional weighed pushdown systems. Section 3 defines problem abstractions, and pre-assumed points-to and string analysis. Section 4 formalizes the problem of policy generation and checking, and Section 5 gives realization algorithms of access rights analysis. Section 6 discusses related work, and we conclude in Section 7.

## 2 Preliminary

A pushdown system is a variant of pushdown automata without input alphabet, and thus is not used as language acceptors but a model of system computation behaviors.

**Definition 1** A pushdown system \( \mathscr{D} \) is \((P, \Gamma, \Delta, p_0, \omega_0)\), where \(P\) is a finite set of control locations, \(\Gamma\) is a finite stack alphabet, \(\Delta \subseteq P \times \Gamma \times P \times \Gamma^*\) is a finite set of transitions, \(p_0 \in P\) is the initial control location, and \(\omega_0 \in \Gamma^*\) is the initial stack contents. A transition \(\langle p, \gamma, q, \omega \rangle \in \Delta\) is written as \(\langle p, \gamma \rangle \rightarrow \langle q, \omega \rangle\). A configuration is a pair \(\langle q, \omega \rangle\) with \(q \in P\) and \(\omega \in \Gamma^*\). A set of configurations \(C\) is regular if \(\{ \omega \mid \langle p, \omega \rangle \in C \}\) is regular. A relation \(\Rightarrow\) on configurations is defined, such that \(\langle p, \gamma \omega \rangle \Rightarrow (q, \omega \omega')\) for each \(\omega' \in \Gamma^*\) if \(\langle p, \gamma \rangle \rightarrow (q, \omega)\), and the reflective and transitive closure of \(\Rightarrow\) is denoted by \(\Rightarrow^*\).

A pushdown system can be normalized by a pushdown system for which \(|\omega| \leq 2\) for each transition rule \(\langle p, \gamma \rangle \rightarrow (q, \omega)\) \[19\].

**Definition 2** A bounded idempotent semiring \(\mathscr{S}\) is \((D, \oplus, \otimes, 0, 1)\), where \(0, 1 \in D\), and

1. \((D, \oplus)\) is a commutative monoid with \(0\) as its unit element, and \(\oplus\) is idempotent, i.e., \(a \oplus a = a\) for all \(a \in D\);
2. \((D, \otimes)\) is a monoid with \(1\) as the unit element;
3. \(\otimes\) distributes over \(\oplus\), i.e., for all \(a, b, c \in D\), we have \(a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)\) and \((b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)\);
4. for all \(a \in D\), \(a \otimes 0 = 0 \otimes a = 0\);
5. A partial ordering \(\sqsubseteq\) is defined on \(D\) such that \(a \sqsubseteq b\) iff \(a \oplus b = a\) for all \(a, b \in D\), and there are no infinite descending chains in \(D\).
By Def. 2 we have that \( \bar{0} \) is the greatest element. From the standpoint of abstract interpretation, PDSs model the (recursive) control flows of the program, weight elements encode transfer functions, \( \otimes \) corresponds to the reverse of function composition, and \( \oplus \) joins data flows. A weighted pushdown system (WPDS) [18] is a generalized analysis framework for solving meet-over-all-path problems for which data domains comply with the bounded idempotent semiring.

**Definition 3** A weighted pushdown system \( \mathcal{W} \) is \((\mathcal{P}, \mathcal{I}, f)\), where \( \mathcal{P} = (P, \Gamma, \Delta, p_0, \omega_0) \) is a pushdown system, \( \mathcal{I} = (D, \oplus, \otimes, 0, \bar{1}) \) is a bounded idempotent semiring, and \( f : \Delta \to D \) is a weight assignment function.

Let \( \sigma = [r_0, \ldots, r_k] \) with \( r_i \in \Delta \) for \( 0 \leq i \leq k \) be a sequence of pushdown transition rules. A value associated with \( \sigma \) is defined by \( \text{val}(\sigma) = f(r_0) \otimes \cdots \otimes f(r_k) \). Given \( c, c' \in P \times \Gamma^* \), we denote by \( \text{path}(c, c') \) the set of transition sequences that transform configurations from \( c \) into \( c' \).

**Definition 4** Given a weighted pushdown system \( \mathcal{W} = (\mathcal{P}, \mathcal{I}, f) \) where \( \mathcal{P} = (P, \Gamma, \Delta, p_0, \omega_0) \), and regular sets of configurations \( S, T \subseteq P \times \Gamma^* \), the **meet-over-all-path** problem computes

\[
\text{MOV}\{S, T, \mathcal{W}\} = \oplus \{\text{val}(\sigma) \mid \sigma \in \text{path}(s, t), s \in S, t \in T\}
\]

We refer by MOV\(\{S, T, \mathcal{W}\} \) to the model checking problem on WPDSs when there are more than one weighted pushdown systems in the context.

WPDSs are extended to **Conditional WPDSs** in [13], by further associating each transition with regular languages that specify conditions over the stack under which a transition can be applied.

**Definition 5** A conditional pushdown system is \( \mathcal{P}_c = (P, \Gamma, \Delta_c, \mathcal{C}, p_0, \omega_0) \), where \( P \) is a finite set of control locations, \( \Gamma \) is a finite stack alphabet, \( \mathcal{C} \) is a finite set of regular languages over \( \Gamma \), \( \Delta_c \subseteq P \times \Gamma \times \mathcal{C} \times P \times \Gamma^* \) is a finite set of transitions, \( p_0 \in P \) is the initial control location, and \( \omega_0 \in \Gamma^* \) is the initial stack contents. A transition \((p, \gamma, L, q, \omega) \in \Delta_c\) is written as \((p, \gamma) \xrightarrow{L} (q, \omega)\). A computation relation \( \Rightarrow_c \) on configurations is defined such that \((p, \gamma_0') \Rightarrow_c (q, \omega_0')\) for all \( \omega' \in \Gamma^* \) if there exists a transition \( r : (p, \gamma) \xrightarrow{L} (q, \omega) \) and \( \omega' \in L \), written as \((p, \gamma_0') \Rightarrow_c (q, \omega_0')\). The reflective and transitive closure of \( \Rightarrow_c \) is denoted by \( \Rightarrow_c^* \). We define \( \text{cpre}^*(C) = \{c' \mid c' \Rightarrow_c c, c \in C\} \) and \( \text{cpost}^*(C) = \{c' \mid c \Rightarrow_c^* c', c \in C\} \) for any \( C \subseteq P \times \Gamma^* \).

**Definition 6** A conditional weighted pushdown system \( \mathcal{W}_c \) is \((\mathcal{P}_c, \mathcal{I}, f)\), where \( \mathcal{P}_c = (P, \Gamma, \Delta_c, p_0, \omega_0) \) is a conditional pushdown system, \( \mathcal{I} = (D, \oplus, \otimes, 0, \bar{1}) \) is a bounded idempotent semiring, and \( f : \Delta_c \to D \) is a weight assignment function.

We lift the model checking problem on WPDSs in Definition 4 to CWPSs and refer it by MOV\(P\) as well.

### 3 Abstraction and Prerequisites

#### 3.1 Abstraction of Java Programs

**Definition 7 (Program Points)** We denote by \( \mathcal{M} \) the set of methods in a program, and by \( \mathcal{L} \) the set of program line numbers each of which contains a statement. Let \( \text{Tag} = \{c, r\} \). A program point is characterized by its enclosing method \( m \in \mathcal{M} \), line number \( l \in \mathcal{L} \), and a tag \( \in \text{Tag} \), and the set of program points is denoted by \( \text{ProgPoint} \subseteq \mathcal{M} \times \mathcal{L} \times \text{Tag} \). We denote by
Definition 10 (Abstract Calling Contexts) a mapping \( \varphi \) sequences of call sites that may lead to \( m \) calling contexts \((\alpha)\). An abstraction function \( \phi \) is defined by a mapping \( \varphi \).

In sequel, we use variables \( \chi \) to range over \( \text{CallSite} \) and \( \zeta \) to range over \( \text{RetPoint} \). Let \( N \) denote the set of natural numbers. For any finite set \( S = \{s_0, \ldots, s_k\} \), and \( N = \{0, \ldots, k\} \), let

- \( \Pi S = \{s_{i_0}s_{i_1} \ldots s_{i_k} \mid \{i_0, \ldots, i_k\} \text{ is a permutation of } N\} \)
- \( \Sigma(\omega) = \{s_{i_0}s_{i_1} \ldots s_{i_j} \} \), for any word \( \omega = s_{i_0}s_{i_1} \ldots s_{i_j} \in S^* \) with \( 0 \leq j \leq k \) and \( 0 \leq i_j \leq k \).

Let \( n_{\text{check}}, n_{\text{priv}} \in \mathcal{M} \) denote the method checkPermission and doPrivileged from the class AccessController, respectively.

Definition 8 (Call Graph) a directed graph, where \( N \subseteq \mathcal{M} \) is the set of nodes, \( E \subseteq \mathcal{M} \times \text{CallSite} \times \mathcal{M} \) is the set of edges, \( s \in N \) is the initial node with no incoming edges, and \( n_{\text{check}} \in N \) is the final state with no outgoing edges. We write \( n \rightarrow n' \) if \( (n, \chi, n') \in E \), and let \( \rightarrow^* \) be the transitive and reflexive closure of \( \rightarrow \).

Definition 9 (Calling Contexts) We denote by \( \text{Context} \subseteq \text{RetPoint}^* \) the set of program calling contexts in terms of call site strings. Given a call graph \( G = (N, E, s, n_{\text{check}}) \), the calling contexts of a method \( m \) (equivalently, local variables of \( m \)) is defined by a mapping \( \phi : \mathcal{M} \rightarrow 2^{\text{Context}} \), such that

\[
\phi(m) = \{ \zeta_0, \ldots, \zeta_i \in \text{Context} \mid \exists k \in \mathbb{N} : m_0 = s, m_{k+1} = m, (m_i, \zeta_i, m_{i+1}) \in E, \chi_i = (m_i, l_i, c), \zeta_i = (m_i, l_i, r), \text{ for each } 0 \leq i \leq k \}
\]

By Def. \( \varphi \) the calling contexts of a method is the (possibly infinite) set of finite yet unbounded sequences of call sites that may lead to \( m \) from the program entry.

Definition 10 (Abstract Calling Contexts) We denote the set of abstract program calling contexts \( \text{AbsCtxt} \subseteq 2^{\text{RetPoint}} \) as sets of call sites along each call sequence. Moreover,

- An abstraction function \( \alpha : \text{Context} \rightarrow \text{AbsCtxt} \) is defined by \( \alpha(c) = \Sigma(c) \) for each \( c \in \text{Context} \), and abstraction function \( \bar{\alpha} : 2^{\text{Context}} \rightarrow 2^{\text{AbsCtxt}} \) is defined by, for each \( c \subseteq \text{Context} \),

\[
\bar{\alpha}(c) = \{ \Sigma(\text{ctxt}) \mid \text{ctxt} \in c, \text{and } cs' \notin \alpha(c) \text{ if } cs' \subset cs \text{ and } cs \in \alpha(c) \}\]

- A concretization function \( \gamma : \text{AbsCtxt} \rightarrow 2^{\text{Context}} \) is defined by \( \gamma(C) = \bigcup_{C' \subseteq C} \Pi C' \) for each \( C \in \text{AbsCtxt} \), and the powerset extension of \( \gamma \) is denoted by \( \hat{\gamma} : 2^{\text{AbsCtxt}} \rightarrow 2^{\text{Context}} \).

The abstract calling contexts of a method \( m \) is defined by a mapping \( \phi_{\text{method}} : \mathcal{M} \rightarrow 2^{\text{AbsCtxt}} \), such that

\[
\phi_{\text{method}}(m) = \bar{\alpha}(\phi(m))
\]

\( \text{AbsCtxt} \) is an over approximation of \( \text{Context} \). Let \( \preceq \) be a binary relation over \( \text{Context} \) such that \( \text{ctxt} \preceq \text{ctxt}' \) for any \( \text{ctxt}, \text{ctxt}' \in \text{Context} \) if \( \Sigma(\text{ctxt}) \subseteq \Sigma(\text{ctxt}') \). It is not hard to see that \( \preceq \) is a preorder, and \( \bar{\alpha}(c) \subseteq cs \text{ i} f c \preceq \hat{\gamma}(cs) \), i.e., Theorem \( \varphi \) which says that abstract calling contexts \( \phi_{\text{method}} \) soundly abstract calling contexts \( \phi \).

Theorem 1 \( (2^{\text{Context}}, \bar{\alpha}, \hat{\gamma}, 2^{\text{AbsCtxt}}) \) is a Galois connection.
3.2 Pre-assumed Analysis

Our framework for access rights analysis assumes points-to analysis and context-sensitive string analysis, and the precision of our analysis depends on that of points-to and string analysis. We therefore expect context-sensitivity from them that is shown to be crucial for the precision of points-to analysis and preferable for string-analysis analysis.

Definition 11 (Context-Sensitive Points-to Analysis) Given a reference variable \( v \) of the method \( m \in M \), a context-sensitive points-to analysis, denoted by \( \text{pta}(v) \), returns the finite set of abstract heap objects that \( v \) may refer to at runtime under certain calling contexts. Each object in \( \text{pta}(v) \) is represented as a triplet \((\text{type,loc},c)\), where \( \text{type} \) is the object type, \( \text{loc} \) is the object allocation site, and \( c \in \phi_m(m) \) is the calling contexts under which the object is constructed, and \( \bigcup_{(\text{type,loc},c) \in \text{pta}(v)} \{c\} = \phi_m(m) \).

Note that string is one kind of reference type. But points-to analysis usually does not handle character-wise operations on strings, whereas string-analysis does.

Definition 12 (Context-Sensitive String Analysis) Given a string variable \( v \) of the method \( m \in M \), a context-sensitive string analysis, denoted by \( \text{sa}(v) \), returns the finite set of string constants that \( v \) may contain at runtime under certain calling contexts. Each element in \( \text{sa}(v) \) is represented as a pair \((sv,c)\), where \( sv \) is the string constant value and \( c \in \phi_m(m) \) is the calling contexts under which \( sv \) is constructed, and \( \bigcup_{(sv,c) \in \text{sa}(v)} \{c\} = \phi_m(m) \).

The two dominating approaches to obtaining context-sensitivity in program analysis are known as context-cloning and context-stacking. The former resembles to inline expansion that copies the called procedures at each call site as much as possible, and hereby has an inherent limit in handling recursive procedure calls. The latter refers to model the program as a pushdown system and the analysis problem as model checking problems on it, e.g., analysis yielded by WPDSs are guaranteed to be context-sensitive. Since the stack of pushdown systems are unbounded, it can naturally model recursive procedure calls.

It is relatively straightforward to adapt a stacking-based analysis to our needs, since WPDSs have the advantage of handling data flow queries as regular languages of pushdown configurations. Consider the stacking-based points-to analysis Japot [12]. Let \( S \) be the source configurations. For each reference variable \( v \) of the method \( m \), we can compute \( \text{pta}(v) \) by unifying the result of \( \text{MOVP}(S,T_{\text{ctxt}}) \) (that returns the set of pairs of object type and allocation sites) for each \( T_{\text{ctxt}} = \{\langle v,m\omega \rangle \mid \Sigma(\omega) \subseteq \text{ctxt} \} \) where \( \text{ctxt} \in \phi_m(m) \). For string-analysis, the analysis in [5] based on context-cloning (specifically \( k \)-CFA) can be reformulated in the framework of WPDSs, and then adapted to a context-sensitive analysis similarly to points-to analysis.

To adapt cloning-based analysis to our needs demands a context-cloning method that is in line with approaches of \( k \)-CFA or approximating loops. In the following, we give an approach to context-cloning, driven by the abstraction on calling contexts in Def. [10].

Definition 13 (Context-Cloning Driven by AbsCtx) Given a call graph \( G = (N,E) \) that is commonly the starting point of program analysis. We construct another graph \( G_{\text{clone}} = (N_{\text{clone}},E_{\text{clone}}) \), where \( N_{\text{clone}} \subseteq 2^{\text{AbsCtx}} \times N \) is the set of nodes, \( E_{\text{clone}} \subseteq N_{\text{clone}} \times \text{CallSite} \times N_{\text{clone}} \) is the set of edges, and we have

- \((c,n) \in N_{\text{clone}} \) for \( c \in \phi_m(n) \), and \( n \in N \);
- \(((c,n),(c',n')) \in E_{\text{clone}} \) if \( c \subseteq c' \) and \((n,n') \in E \) for \((c,n),(c',n') \in N_{\text{clone}} \).
We can then obtain context-sensitive analysis by applying context-insensitive analysis to $G_{clone}$, e.g., the well-known points-to analysis framework Spark [11] can be lifted to a context-sensitive analysis easily by cloning its points-to graph (that is product with the call graph) in this manner, and the most influential string analysis for Java programs JSA (Java String Analyzer) [6] can be lifted to a context-sensitive counterpart as well by cloning its front-end flow graph, with no need to modify the back-end analysis algorithms.

3.3 Abstraction of Policy System

**Definition 14 (Policy System)** Let Domain denote a finite set of protection domains, and let Perms denote the universe of all permissions involved in the given program. We denote by

- $\text{dom}: \mathcal{M} \rightarrow \text{Domain}$ the mapping from methods to their protection domain.
- $\text{perm}: \text{Domain} \rightarrow 2^{\text{Perms}}$ the mapping that grants each protection domain a set of permissions.

We extend $\text{perm}$ element-wise and let $\text{policy} = \text{perm} \circ \text{dom}$.

All classes in a protection domain are granted the same set of permissions, so do all methods and all program points in the classes. Especially, all methods belonging to the system domain, e.g., the method doPrivileged from the class AccessController, are granted all permissions in Perms.

**Definition 15 (Check Points)** We define CheckPoint as the set of call sites that directly call the method checkPermission, by

$$\text{CheckPoint} = \{ \chi \mid \exists n \in N, \chi \in \text{CallSite} : (n, \chi, n_{\text{check}}) \in E \}$$

Let $\phi_{\text{perm}} : \text{Perms} \rightarrow 2^{\text{Context}}$ be a mapping from permissions to the calling contexts under which each permission is constructed. We generate Perms and $\phi_{\text{perm}}$ as follows. Initially, $\text{Perms} = \emptyset$, and $\phi_{\text{perm}} = \lambda x. \emptyset$. For each call site $\chi = (m, l, c) \in \text{CheckPoint}$ where $l$ is supposed to contain the expression of “checkPermission($pv$)”, we first call points-to analysis $pta(pv)$ to get the permission objects that $pv$ refers to. For each $(\text{Type}, loc, c) \in pta(pv)$, the heap allocation site referred to by $loc$ is supposed to contain statements in one of the following form,

\[
\begin{align*}
\text{npv} &= \text{new Type(target,action)} \quad (1) \\
\text{npv} &= \text{new Type(target)} \quad (2) \\
\text{npv} &= \text{new Type()} \quad (3)
\end{align*}
\]

Assume $loc$ belongs to the method $m'$. We add each of the following permission $\text{perm}$ to $\text{Perms}$,

\[
\text{perm} = \begin{cases} 
\text{Type,sv}_1,\text{sv}_2 \quad &\text{where } (\text{sv}_1, c_1) \in \text{sa(target)}, (\text{sv}_2, c_2) \in \text{sa(action)}, c_1 = c_2, \\
&\text{and } \phi_{\text{perm}}(\text{perm}) = \phi_{\text{perm}}(\text{perm}) \cup \{c_1\} \text{ for (1)} \\
\text{Type,sv} \quad &\text{where } (\text{sv}, c') \in \text{sa(target)} \\
&\text{and } \phi_{\text{perm}}(\text{perm}) = \phi_{\text{perm}}(\text{perm}) \cup \{c'\} \text{ for (2)} \\
\text{Type} \quad &\text{where } \phi_{\text{perm}}(\text{perm}) = \phi_{\text{perm}}(\text{perm}) \cup \phi_{\text{method}}(m') \text{ for (3)}
\end{cases}
\]
4 Problem Formalization

**Definition 16 (Context-Sensitive Call Graph)** A context-sensitive call graph \( G_{cs} = (G, \phi_{edge}) \) consists of a call graph \( G = (N, E, s, n_{check}) \) and a mapping \( \phi_{edge} : E \rightarrow 2^{AbsCtx} \), such that for each node \( n \in N \),

- \( \phi_{edge}(e) \subseteq \phi_{method}(n) \) for each edge \( e = (n, \chi, n') \in E \);
- \( \bigcup_{e=(n,\chi,n') \in E} \phi_{edge}(e) = \phi_{method}(n) \).

We define a mapping \( \phi_{route} : (\rightarrow^*) \rightarrow 2^{AbsCtx} \) by, for each \( n \rightarrow^i n' \), \( \phi_{route}(n \rightarrow^i n') = \)

\[
\begin{cases}
\phi_{edge}(n \rightarrow n') & \text{if } i = 1 \\
\{ c \cup c' \mid \exists n'' \in N : c \in \phi_{route}(n \rightarrow^{i-1} n''), c' \in \phi_{edge}(n'' \rightarrow n') \} & \text{if } i > 1
\end{cases}
\]

In Java, due to polymorphism and late binding, the target method of a dynamic dispatch (e.g., \( r\cdot fun(\cdots) \)) depends on the runtime type of receiver objects (i.e., objects that \( r \) refers to). Therefore, a call edge, as well as a path in the call graph, is conditioned by receivers’ points-to information which is further conditioned by calling-contexts as in Def. [11]. The calling contexts associated with a call edge and a call path is defined by \( \phi_{edge} \) and \( \phi_{route} \) in Def. [16] respectively. As aforementioned, \( \phi_{edge} \) can be generated by Java semantics of dynamic dispatch, given points-to analysis. The calling contexts of a call graph is obtained by the union of the calling contexts of each edge along this path. An algorithm for constructing \( G_{cs} \) is given in [13]. We refer to [13] for details, and illustrate in Example [11] the semantics for “privileged” codes specific to access control.

**Definition 17 (Valid Paths)** Given a context-sensitive call graph \( G_{cs} = (G, \phi_{edge}) \) where \( G = (N, E, s, n_{check}) \), we define

- the set of (call) paths from \( s \) to a node \( n \in N \) by
  \[
  \text{path}(n) = \{ e_0e_1\ldots e_k \mid \exists k \in \mathbb{N} : n_0 = s, n_{k+1} = n, e_i = (n_i, \chi_i, n_{i+1}) \in E \text{ for each } 0 \leq i \leq k \}
  \]

- the set of subsequences of \( \text{path}(n) \) that are truncated by the node \( n_{priv} \) as
  \[
  \text{t\_path}(n) = \{ e_0e_1e_2\ldots e_k \mid \exists k \in \mathbb{N} : n_0 = s, n_{k+1} = n, n_0 \rightarrow^* n_{priv}, e_0 = (n_{priv}, \chi_0, n_1),
  \]
  \[
  e_i = (n_i, \chi_i, n_{i+1}) \text{ for each } 1 \leq i \leq k \}
  \]

- the set of valid paths from \( s \) to a node \( n \in N \) by
  \[
  \text{v\_path}(n) = \{ \sigma \in \text{path}(n) \mid \exists c \in \phi_{route}(\sigma) : c \subseteq \text{sites}(\sigma) \}
  \]
  where given a node \( n \in N \), we define
  \[
  \text{sites}(\sigma) = \{ (n, l, r) \mid e = (n, \chi, n') \in \Sigma(\sigma) \text{ and } \chi = (n, l, c) \}
  \]
  for a path \( \sigma = e_0e_1\ldots e_k \in \text{path}(n) \) with \( k \in \mathbb{N} \).

Consider a call path \( \sigma \). Each \( c \in \phi_{route}(\sigma) \) is the set of call sites through which \( \sigma \) can be valid. Therefore, \( \sigma \) is a valid path if there exists \( c \in \phi_{route}(\sigma) \) such that \( c \) is included in the set of call sites visited by \( \sigma \), i.e., \( \text{sites}(\sigma) \), as defined in Def. [17]
Example 2 Consider the code snippet in Fig. 2 that consists in methods \( m_1, m_2, n_{priv}, n_{check}, \) and \( \text{OnePrivAction}\cdot \text{run}() \), \( \text{AnotherPrivAction}\cdot \text{run}() \), and \( m_1 \) and \( m_2 \) do not have paths reachable to each other. Methods are grouped by dotted circles. \( \text{OnePrivAction} \) and \( \text{AnotherPrivAction} \) are classes that implement the interface \( \text{PrivilegedAction} \). The set of call edges are \( \{e_1, \cdots, e_6\} \), e.g., \( e_1 = ((m_1, (l_1, 2, c)), n_{priv}) \) and \( e_5 = ((\text{OnePrivAction}\cdot \text{run}, (\text{OnePrivAction}\cdot \text{run}, l_5, c)), n_{check}) \). We do not explicitly show the call site inside \( n_{priv} \), because the method \( \text{act}\cdot \text{run}() \) is implicitly involved when the method \( \text{doPrivilege}(\text{act}) \) is called.

Since \( n_{check} \) and \( n_{priv} \) are static methods, \( \phi_{edge}(e_i) = \emptyset \) for \( i \in \{1, 2, 5, 6\} \). Assume the calling contexts of \( n_{priv} \) i.e. \( \phi_{method}(n_{priv}) = \{c_1, c_2\} \), such that \((m_1, l_1, r) \in c_1 \) but \( \notin c_2 \) (i.e., \( c \in c_1 \setminus c_2 \)) and \((m_2, l_4, r) \in c_2 \) but \( \notin c_1 \) (i.e., \( c_2 \setminus c_1 \)). We then have that \( \phi_{edge}(e_3) = \{c_1\} \) and \( \phi_{edge}(e_4) = \{c_2\} \) by the semantics of calling \( n_{priv} \). Let \( \sigma' \) be a path leading to \( m_1 \), and consider a path \( \sigma = \sigma'e_1e_4e_6 \). By above assumptions on the example, \((m_2, l_4, r) \notin \text{sites}(\sigma') \). We have \( \text{sites}(\sigma) = \text{sites}(\sigma') \cup \{l_1, l_2, r\} \) and \( \phi_{route}(\sigma) = \{c \cup c_2 \mid c \in \phi_{route}(\sigma')\} \). \( \sigma \) is not a valid path because there does not exist \( c \in \phi_{route}(\sigma) \) such that \( c \subseteq \text{sites}(\sigma) \).

Definition 18 (Dependency Graph) Given a program in SSA (Static Single Assignment) form. Let \( T_{perm} \) denote the class (or type) \( \text{Permission} \) or any of its subclasses. Let \( L_{alloc} \subseteq L \) be the set of program lines that allocate objects of \( T_{perm} \), and let \( \text{AllocPerm} \subseteq \mathcal{M} \times L_{alloc} \).

A dependency graph \( G_{dep} \) of the program is a directed graph \( (N_{dep}, E_{dep}, S_{dep}, F_{dep}) \), where \( N_{dep} \subseteq \mathcal{M} \times L \) is the set of nodes, \( E_{dep} \subseteq N_{dep} \times N_{dep} \) is the set of edges, \( S_{dep} = \text{AllocPerm} \) is the set of initial nodes with no incoming edges, and \( F_{dep} \subseteq \text{CheckPoint} \) is the set of final nodes without outgoing edges.

Moreover, \( E_{dep} \) is the smallest set that contains \((n, n')\) when \( n = (m, l) \) and \( n' = (m', l') \) if the variable (more specifically, local variables like \( x \), static fields like \( A.f \), and instance fields like \( c.f \) where \( c \) denotes the abstract heap object resolved by points-to analysis) of reference type \( T_{perm} \) defined in \( l \) is used in \( l' \).

Definition 19 (Dependency Paths) Given a dependency graph \( G_{dep} = (N_{dep}, E_{dep}, S_{dep}, F_{dep}) \), we define the set of dependency paths from \( S_{dep} \) to a node \( n \in N_{dep} \) by

\[
dpath(n) = \{e_0e_1 \cdots e_k \mid \exists k \in \mathbb{N} : n_0 \in S_{dep}, n_{k+1} = n, e_i = (n_i, n_{i+1}) \in E_{dep} \text{ for each } 0 \leq i \leq k\}
\]
In the rest of this section, we fix a context-sensitive call graph $G_{cs} = (G, \phi_{edge})$ where $G = (N, E, s, n_{check})$, and a dependency graph $G_{dep} = (N_{dep}, E_{dep}, S_{dep}, F_{dep})$.

**Definition 20 (Relate Dependency Paths to Permissions)** Given a node $n \in N_{dep}$ and a dependency path $\pi = e_0 e_1 \cdots e_k \in dpath(n)$ for some $k \in \mathbb{N}$, where $e_i = (n_i, n_{i+1})$ for each $0 \leq i \leq k$, and $n_j = (m_j, l_j)$ for each $0 \leq j \leq k + 1$. We define

$$\omega_\pi = \zeta_{i_0} \cdots \zeta_{i_j}$$

where $0 \leq i_0 \leq i_1 \cdots \leq i_j \leq k + 1$, $0 \leq j \leq k + 1$, and for each $i_m \in \{i_0, \cdots, i_j\}$,

$$\zeta_{i_m} = \begin{cases} (m_{i_m}, l_{i_m}, c) & \text{if } l_{i_m} \text{ is a method call, and } (n_{i_m}, n_{i_m+1}) \in E_{dep} \text{ such that } m_{i_m} \neq m_{i_m+1} \\ (m_{i_m}, l_{i_m}, r) & \text{if } l_{i_m} \text{ is a method call and } (n_{i_m-1}, n_{i_m}) \in E_{dep} \text{ such that } m_{i_m} \neq m_{i_m-1} \end{cases}$$

Specifically, $\omega_\pi = \varepsilon$ if such $i_m$ does not exist.

The initial node of $\pi$ is $n_0 = (m_0, l_0)$. Let $\sigma = e'_0 e'_1 \cdots e'_h \in vpath(m_0)$ be a valid path from $s$ to $m_0$ in $G$ for some $h \in \mathbb{N}$, where $e'_i = (n'_i, \chi_i, n'_{i+1})$ for each $0 \leq i \leq h$. We define

$$\omega_\sigma = \chi_0 \cdots \chi_h$$

Let $\lfloor (m,l,c) \rfloor$ denote $(m,l,c) \in \text{CallSite}$ and let $\lfloor (m,l,c) \rfloor$ denote $(m,l,c) \in \text{RetPoint}$. The set of all such parentheses induced by $\text{CallSite} \cup \text{RetPoint}$ is denoted by $\Sigma_{cf}$. We say $\pi$ matches with $\sigma$ if $\omega_\pi \omega_\sigma$ is a context-free language over $\Sigma_{cf}$. The set of all such $\sigma$ that $\pi$ matches with is denoted by $\text{match}(\pi)$.

Given a permission $\text{perm} \in \text{Perms}$, we say $\pi$ relates to $\text{perm}$, if (i) there exists $\sigma \in \text{match}(\pi)$, and (ii) there exists $c \in \phi_{\text{perm}}(\text{perm})$ such that $c \subseteq \text{sites}(\sigma)$.

In Def. 20 $\pi$ matches with $\sigma$ means that $\pi$ and $\sigma$ jointly constitute a valid inter-procedural data flow with respect to permissions allocated at the initial node of $\pi$. By valid, we mean as usual that method calls and returns match with each other. If any calling context (i.e., the set of call sites) for allocating a permission is included (i.e., visited) in $\sigma$, we regard that $\pi$ relates to that permission.

**Definition 21 (Relate Valid Paths to Dependency Paths)** Given a node $n \in N_{dep}$, and a dependency path $\pi = e_0 e_1 \cdots e_k \in dpath(n)$ for some $k \in \mathbb{N}$, where $e_i = (n_i, n_{i+1})$ for each $0 \leq i \leq k$, and $n_j = (m_j, l_j)$ for each $0 \leq j \leq k + 1$. We define nodes($\pi$) = $\{m_i \mid 0 \leq i \leq k + 1\}$.

Given a node $n' \in N$, and a valid path $\sigma = e'_0 e'_1 \cdots e'_h \in vpath(n')$ for some $h \in \mathbb{N}$ where $e_i = (m'_i, \chi_i, m'_{i+1})$ for each $0 \leq i \leq h$, we define nodes($\sigma$) = $\{m'_i \mid 0 \leq i \leq h + 1\}$.

We say $\sigma$ relates to $\pi$ if there exists a path $\sigma' \in \text{match}(\pi)$ such that nodes($\sigma$) $\subseteq$ nodes($\pi$) $\cup$ nodes($\sigma'$) $\cup$ $\{n_{\text{check}}\}$.

In Def. 21 we regard $\sigma$ relates to $\pi$ if the set of methods visited by $\sigma$ is included in the valid inter-procedural data flow constituted by $\pi$ and some $\sigma' \in \text{match}(\pi)$.

**Definition 22 (Policy Generation)** We define policy : $\mathcal{M} \rightarrow 2^{\text{Perms}}$ by, for each valid path $\sigma \in vpath(n_{\text{check}})$, and each dependency path $\pi \in dpath(n)$ for each $n \in E_{dep}$

(i) $\text{perm} \in \text{policy}(m)$ for each $m \in \text{nodes}(\sigma)$ if $n_{\text{priv}} \notin \text{nodes}(\sigma)$, $\sigma$ relates to $\pi$ and $\pi$ relates to $\text{perm}$;
(ii) $\text{perm} \in \text{policy}(m)$ for each $m \in \text{nodes}(\sigma')$ if $n_{\text{priv}} \in \text{nodes}(\sigma)$, $\sigma$ relates to $\pi$, $\pi$ relates to $\text{perm}$, and $\sigma'$ is a suffix of $\sigma$ for some $\sigma' \in tpath(n_{\text{check}})$.

Note that both $\pi$ and $\sigma$ in Def. 22 can be infinitely many.
Definition 23 (Policy Checking) Given a policy $\mathcal{M} \rightarrow 2^{\text{Perms}}$ and a policy $\mathcal{M} \rightarrow 2^{\text{Perms}}$ generated by Def[22] Stack inspection triggered in the program always succeed if $\text{policy}(m) \subseteq \text{policy}(m)$ for each $m \in \mathcal{M}$, and may fail otherwise.

We illustrate the idea of problem formalization by Example 3

Example 3 Consider the code fragments in Fig. 3, where in the dependency graph each node corresponds to a permission manipulation statement and connected by the dashed arrows. The underlying call graph is also shown of which each node is grouped by the dotted circles and connected by the solid arrows. The following permissions are involved in this example:

- $\text{perm}_1$: SocketPermission("domain: 80", "connect") induced at "npv = new SocketPermission("domain: 80", "connect")";
- $\text{perm}_2$: FilePermission("public", "read") induced at "npv = new FilePermission(arg_m, "read")";
- $\text{perm}_3$: FilePermission("personal", "read") induced at "npv = new FilePermission(arg_m, "read")".

where $\text{perm}_1$ is created and referred to by methods in the current call stack when stack inspection is triggered, whereas $\text{perm}_2, \text{perm}_3$ are created/passed by methods that are finished calls and do not reside on the current call stack of stack inspection, and stored/referred by field access such that their data flows are beyond the control flow to the checkpoints.

Consider a dependency path $\pi: (4)(5)(6)(7)(9)(10)(12)$ and a valid path $\sigma: (0)(1)(3)$. By Def. 20, $\omega_1 = [(m_2, d_1), (m_1, d_1)]$, and $\omega_2 = [(m_2, d_1), (m_1, d_1)]$. We have $\pi$ relates to $\sigma$ and further relates to $\text{perm}_2$ that is constructed following the path $\sigma$. We have a valid path $\sigma': (0)(11)(13)$. By Def. 22, nodes($\sigma'$) = {s, m2, m3, n_check}, and nodes($\pi$) = {s, m1, m2, m4, m10}. We have $\sigma'$ relates to $\pi$ and thus to $\text{perm}_2$. By Def. 22, s, m2, m3 will be granted $\text{perm}_2$, whereas m0, m1 should not. If we consider the dependency path $\pi': (14)$ and the valid path $\sigma'': (0)(15)$. We have $\omega_2 = \epsilon$ and that $\pi'$ relates to $\sigma''$ and hereby relates to $\text{perm}_1$, and s, m2 should be granted $\text{perm}_1$.

5 Realization Algorithms

5.1 Policy Generation

Definition 24 (Modeling Context-Sensitive Call Graph) Given a context-sensitive call graph $G_{cs} = (G, \phi_{\text{edge}})$ where $G = (N, E, s, n_{\text{check}})$, we define a conditional pushdown system $\mathcal{P}_c = (\{} \cdot, \Gamma, \mathcal{C}, s, \}$, where

- the set of control locations is a singleton $\{ \cdot \}$;
- the stack alphabet $\Gamma \subseteq \mathcal{M} \cup \text{RetPoint}$ is encoded from nodes of $G$, i.e., methods, and return points.
- we write $\alpha \xrightarrow{C_e} \omega$ for $(\alpha, \Gamma, \cdot, \omega) \in \Delta_e$. $\Delta_e$ is constructed as follows. for each edge $e = (n, \chi, n') \in E$ where $\chi = (n, l, c)$, let $\xi = (n, l, r)$, we have

$$n \xrightarrow{C_e} n' \xi$$

where $C_e = \bigvee_{c=(y_0, \ldots, y_c) \in \phi_{\text{edge}}(e)} \bigvee_{\{i_0, \ldots, i_{|c|}\} \in \Xi(|c|)} \Gamma^* \chi_{i_0} \Gamma^* \chi_{i_1} \cdots \chi_{i_{|c|-1}} \Gamma^* \chi_{i_c} \Gamma^*$
where $\Sigma(k)$ denote the set of all permutations of $\{0, 1, \cdots, k\}$ for $k \in \mathbb{N}$, and $\cup$ denote the set union of regular expressions.

In Def. 24, $C_e$ means that some calling context of the call edge in question is contained in the current call stack.

**Definition 25 (Modeling Dependency Graph)** Give a dependency graph $G_{dep} = (N_{dep}, E_{dep}, S_{dep}, F_{dep})$, we define a conditional pushdown system $P'$, where $P'$ is constructed as follows, for each edge $e = (n, n') \in E_{dep}$ where $n = (m, l)$ and $n' = (m', l')$, we have

$$m \xrightarrow{C_e} e \text{ and } (m', l', r) \xrightarrow{C_e} m'$$

if $l$ is a method return statement (or $l'$ is a method call statement), and $m \neq m'$, where $C_e = \Gamma^*$, i.e., no conditions.

A dependency graph has edges that correspond to statements of method calls. But they have corresponding call edges in the call graph and are considered when modeling context-sensitive call graphs.

**Definition 26 (Program Modeling)** We define a conditional pushdown system $P_{prog} = (\{\cdot\}, \Gamma, \mathcal{C}_{prog}, \Delta_{prog}, \cdot, s)$ where $\Gamma \subseteq \mathcal{M} \times \text{RetPoint}$, $\mathcal{C}_{prog} = \mathcal{C} \cup \mathcal{C}'$, and $\Delta_{prog} = \Delta_c \cup \Delta_s$, by combining the conditional pushdown system $P_c$ and $P_e$ generated for $G_{cs}$ and $G_{dep}$, respectively.

**Definition 27 (Weight Domain)** We define a bounded idempotent semiring $\mathcal{I}_{gen} = (D_{gen}, \oplus_{gen}, \otimes_{gen}, 0, I)$, where

- $D_{gen} \subseteq 2^{\mathcal{M}} \times 2^{\mathcal{M}} \times 2^{\text{RetPoint}} \cup \{0\}$, and $I = \{(0, 0, 0, 0)\}$;
- for any $d, d' \in D_{gen}$, $d \oplus_{gen} d' = d \cup d'$, and

$$d \otimes_{gen} d' = \{(M_1 \cup M_1', M_2, M_3 \cup M_3', M_4 \cup M_4') | (M_1, M_2, M_3, M_4) \in d, (M_1', M_2', M_3', M_4') \in d'\}$$

It is not hard to prove that both $\oplus_m$ and $\otimes_m$ are associative, and $\oplus_m$ is commutative and distributive over $\otimes_m$, which holds for a bounded idempotent semiring.

**Definition 28 (Modeling Policy Generation)** We define a conditional weighted pushdown system $\mathcal{W}_{gen} = (P_{prog}, \mathcal{I}_{gen}, f_{gen})$. For each transition rule $\delta \in \Delta_{gen}$, $f_{gen}(\delta)$ is defined as follows,

- if $\delta$ is a push rule $m \xrightarrow{C_e} m'(m, l, r)$,

$$f_{gen}(\delta) = \begin{cases} \{(m, \Gamma, 0, \{(m, l, r)\})\}, & \text{if } m = n_{priv}, \\ \{(m, 0, 0, \{(m, l, r)\})\}, & \text{otherwise} \end{cases}$$

- if $\delta$ is a pop rule $m \xrightarrow{C_e} \varepsilon$, $f_{gen}(\delta) = \{(0, 0, \{m\}, \emptyset)\}$.

- otherwise $f_{gen}(\delta) = I$

**Definition 29 (Algorithm for Policy Generation)** Given a conditional weighted pushdown system $\mathcal{W}_{gen} = (P_{prog}, \mathcal{I}_{gen}, f_{gen})$ constructed by Def. 28 We compute

$$\text{result} = \text{MOV}(\{(\cdot, s)\}, T, \mathcal{W}_{prog})$$

where $T = \{(\cdot, n_{check} \omega) | \omega \in \Gamma^*\}$.

For any $d = (M_1, M_2, M_3, M_4) \in \text{result}$, and $perm \in \text{Perms}$, we say $perm$ is required by $d$ if there exists $c \in \phi_{perm}(\text{perm})$ such that $c \subseteq M_4$. For each $m \in M_1 \setminus M_3$, we have $perm \in \text{policy}(m)$ if $perm$ is required by $d$.  

For each $d$ computed in Def. 29, $M_d$ is the calling history, in terms of return points of the valid inter-procedural data flow constituted by some call path and dependency path, when stack inspection is triggered, $M_1$ contains methods that reside on the call paths truncated by $n_{\text{priv}}$. $M_2$ is supposed to be $\emptyset$ by our modeling, because $n_{\text{priv}}$ can never be the initial node of the call graph, and $M_3$ contains finished called methods that do not reside on the current call stack.

**Example 4** We use Example 1 to show how the algorithm for policy generation in Def. 29 basically functions. We denote methods connectToFaculty, ConnectToStudent, createSocket, checkConnect, checkAccess by $m_f$, $m_s$, $m_{\text{socket}}$, $m_{\text{connect}}$, $m_{\text{access}}$, respectively; and assume that $m_f$ and $m_s$ are called from the entry method $s$ from $l_1$ and $l_2$, respectively. $\Delta_c$ in Def. 27 is constructed as follows, and $\Delta_e'$ for dependency graphs in Def 25 is $\emptyset$. 

\[
\begin{align*}
\delta_1 : s \xrightarrow{C} m_f \zeta_1, \text{ where } \zeta_1 &= (s, l_1, r) \\
\delta_2 : s \xrightarrow{C} m_s \zeta_2, \text{ where } \zeta_2 &= (s, l_2, r) \\
\delta_3 : m_f \xrightarrow{C} m_{\text{socket}} \zeta_3, \text{ where } \zeta_3 &= (m_f, l_3, r) \\
\delta_4 : m_s \xrightarrow{C} m_{\text{socket}} \zeta_4, \text{ where } \zeta_4 &= (m_s, l_4, r) \\
\delta_5 : m_{\text{socket}} \xrightarrow{C} m_{\text{connect}} \zeta_5, \text{ where } \zeta_5 &= (m_{\text{socket}}, l_5, r) \\
\delta_6 : m_{\text{connect}} \xrightarrow{C} n_{\text{check}} \zeta_6, \text{ where } \zeta_6 &= (m_{\text{connect}}, l_6, r) \\
\delta_7 : m_{\text{socket}} \xrightarrow{C} n_{\text{priv}} \zeta_7, \text{ where } \zeta_7 &= (m_{\text{socket}}, l_7, r) \\
\delta_8 : n_{\text{priv}} \xrightarrow{C} \text{Priv.run} \zeta_8, \text{ where } \zeta_8 &= (n_{\text{priv}}, l_8, r) \\
\delta_9 : \text{Priv.run} \xrightarrow{C} m_{\text{access}} \zeta_9, \text{ where } \zeta_9 &= (\text{Priv.run}, l_9, r) \\
\delta_{10} : m_{\text{access}} \xrightarrow{C} n_{\text{check}} \zeta_{10}, \text{ where } \zeta_{10} &= (m_{\text{access}}, l_{10}, r)
\end{align*}
\]

where $l_i$ denotes somewhere unknown, and (i) $f_{\text{gen}}(\delta_8) = \{(\{n_{\text{priv}}\}, \Gamma, 0, \{\zeta_8\})\}$ and $f_{\text{gen}}(\delta_i) = \{(\{m\}, 0, 0, \{\zeta_i\})\}$ for $\delta_i : m \xrightarrow{C} m' \zeta_i$ with $i \neq 8$; (ii) $C = \Gamma^*$ and $\Delta_c$ is in the form of Def. 24 with $\Phi_{\text{edge}}(e) = \{\{\zeta_i, \zeta_3, \zeta_4\}, \{\zeta_2, \zeta_5, \zeta_6\}\}$.

We compute $\text{result} = \{d_1, d_2, d_3, d_4\}$ in Def. 29 where 

\[
\begin{align*}
d_1 &= \{(s, m_s, m_{\text{socket}}, m_{\text{connect}}), 0, 0, \{\zeta_i | i \in \{2, 4, 5, 6\}\}\} \\
d_2 &= \{(\text{Priv.run}, m_{\text{access}}), 0, 0, \{\zeta_i | i \in \{1, 3, 7, 8, 9, 10\}\}\} \\
d_3 &= \{(\text{Priv.run}, m_{\text{access}}), 0, 0, \{\zeta_i | i \in \{2, 4, 7, 8, 9, 10\}\}\} \\
d_4 &= \{(s, m_f, m_{\text{socket}}, m_{\text{connect}}), 0, 0, \{\zeta_i | i \in \{1, 3, 5, 6\}\}\}
\end{align*}
\]

Furthermore, we have 

\[
\begin{align*}
\Phi_{\text{perm}}(\text{perm}_a) &= \{(\zeta_i | i \in \{1, 3, 7, 8, 9\}\}, \{\zeta_i | i \in \{2, 4, 7, 8, 9\}\}\} \\
\Phi_{\text{perm}}(\text{perm}_f) &= \{(\zeta_i | i \in \{1, 3, 5\}\} \\
\Phi_{\text{perm}}(\text{perm}_s) &= \{(\zeta_i | i \in \{2, 4, 5\}\}
\end{align*}
\]

By Def. 29 we can conclude that $\text{perm}_a$ is required by $d_2, d_3$, $\text{perm}_f$ is required by $d_4$, and $\text{perm}_s$ is required by $d_1, \text{Priv.run}, m_{\text{access}} \text{ possess } \text{perm}_a, m_f \text{ possess } \text{perm}_f, m_s \text{ possess } \text{perm}_s$, and $s, m_{\text{socket}}, m_{\text{connect}} \text{ possess } \text{perm}_a, \text{perm}_f$.

We use parts of Example 3 to briefly demonstrate the case when dependency graphs play a role. Consider the call path $(0)(1)(3)$ and dependency path $(4)(5)(6)(7)(9)(10)(12)(13)$. $\Delta_e$ is constructed
as follows.

\[
\begin{align*}
\delta_1 : s & \xrightarrow{c} m_2 \xi_1, \text{ where } \xi_1 = (s, l, r) \\
\delta_2 : m_2 & \xrightarrow{c} m_1 \xi_2, \text{ where } \xi_2 = (m_2, l_1, r) \\
\delta_3 : m_1 & \xrightarrow{c} m_0 \xi_3, \text{ where } \xi_3 = (m_1, l_3, r) \\
\delta_4 : m_2 & \xrightarrow{c} m_3 \xi_4, \text{ where } \xi_4 = (m_2, l_4, r) \\
\delta_5 : m_3 & \xrightarrow{c} n_{\text{check}} \xi_5, \text{ where } \xi_5 = (m_3, l_6, r)
\end{align*}
\]

and \(\Delta'_i\) is constructed as follows,

\[
\begin{align*}
\delta_6 : m_0 & \xrightarrow{c} \epsilon \\
\delta_7 : \xi_3 & \xrightarrow{c} m_1 \\
\delta_8 : m_1 & \xrightarrow{c} \epsilon \\
\delta_9 : \xi_2 & \xrightarrow{c} m_2
\end{align*}
\]

where \(f_{\text{gen}}(\delta_6) = \{(\emptyset, \emptyset, \{m_0\}, \emptyset)\}\), \(f_{\text{gen}}(\delta_8) = \{(\emptyset, \emptyset, \{m_1\}, \emptyset)\}\), and \(f_{\text{gen}}(\delta_i) = \{\{(m), \emptyset, \emptyset, \{\xi_i\}\}\}\) for \(\delta_i : m \xrightarrow{c} m' \xi_i\) with \(i \in \{1, 2, 3, 4, 5\}\), and \(C = \Gamma^*\).

We compute \(\text{result} = \{(M_1, \emptyset, \{m_0, m_1\}, M_4)\}\), where \(M_1 = \{s, m_0, m_1, m_2, m_3, n_{\text{check}}\}\) and \(M_4 = \{\xi_i | 1 \leq i \leq 5\}\). Since \(\phi_{\text{perm}}(\text{perm}_2) = \{\xi_1, \xi_2, \xi_3\}\). We know \(\text{perm}_2\) is granted to methods in \(M_1 \setminus \{m_0, m_1\}\).

### 5.2 Policy Checking

Another popular need in access rights analysis is checking whether the program function properly, e.g., codes from trusted domains always pass access control, given a policy file that is commonly generated by the application developers. By Def. 23, one approach to policy checking is first generating a policy required by passing stack inspections by Def. 22, and then check whether the given policy includes the required policy. Instead of generating the required policy in advance, an alternative is on-demand checking whether all methods in the current call stack are granted required permissions at checkpoints. The two approaches to policy checking is quite in line with the two ways of implementing the stack inspection mechanism by virtual machines in an either eager or lazy manner.

We present the on-demand checking algorithm in this section. Given a trusted domain, our approach consists of three steps of

- determining analysis points within codes of the given domain that trigger stack inspection; and
- identifying permission requirements involved in policy checking on each analysis point; and
- checking policy which determines whether stack inspections triggered by a concerned domain always succeed or may fail.

#### 5.2.1 Determining Analysis Points

**Definition 30 (Boundary)** Given a call graph \(G = (N, E, s, n_{\text{check}})\). Let \(l : N \rightarrow \text{Domain}\) be a mapping from methods to their belonging protection domains. A boundary of a domain \(dm \in \text{Domain}\), denoted by \(\mathcal{B}(dm)\), is defined by

\[
\mathcal{B}(dm) = \{n \in N \mid (n, \chi, n') \in E, l(n) = dm, l(n) \neq l(n')\}
\]

The boundary of a domain \(dm\) refers to methods with outgoing edges to methods from different domains, e.g., Java libraries.
Definition 31 (Analysis Points) Assume the conditional pushdown system encoded by Def. 22. We define analysis points of a given domain $dm \in \text{Domain}$ by

$$\text{AnalysisPoints}(dm) = \{\zeta = (n,l,r) \in \text{RetPoint} \mid n \in \mathcal{B}(dm), l \in \mathcal{L}, \exists n' \in \Gamma, \omega, \omega' \in \Gamma^*: \langle \cdot, n' \zeta \omega' \rangle \in c\text{post}^*(\{\langle \cdot, s \rangle \}) \cap c\text{pre}^*(\{\langle \cdot, n \text{check} \omega \rangle \})\}$$

5.2.2 Identifying Permission Requirements

Definition 32 (Modeling Permission Requirements) We define a conditional weighted pushdown system $\mathcal{W}_{ctx} = (\mathcal{P}_{prog}, \mathcal{I}_{ctx}, \mathcal{F}_{ctx})$ where $\mathcal{P}_{prog}$ is the conditional pushdown system defined before, and

- the idempotent semiring $\mathcal{I}_{ctx} = (D_{ctx}, \oplus_{ctx}, \otimes_{ctx}, 0, T)$, where $D = 2^{\mathcal{C}} \cup \{\emptyset\}$, $T = \emptyset$, $\oplus_{ctx}$ is set union, and $\otimes_{ctx}$ is element-wise set union;

- for each $\delta : \alpha \xrightarrow{C} \omega \in \Delta_{gen}$, $f_{ctx}(\delta) = \{n'\}$ if $\omega = n \zeta$, $\zeta = (n',l,r)$, and $f_{ctx}(r) = 1$ otherwise.

Definition 33 (Identifying Permission Requirements) Given a domain $dm \in \text{Domain}$ and an analysis point $\zeta = (n,l,r) \in \text{AnalysisPoints}(dm)$, we compute

$$\phi_{method}(n) = \text{MOVP}(S, T, \mathcal{W}_{ctx})$$

where $S = \{\langle \cdot, s \rangle\}$, $T = \{\langle \cdot, n \omega \rangle \mid \omega \in \Gamma^*\}$. We define permission requirements on $n$ by

$$\text{PermReqs}(\zeta) = \{\text{perm} \in \text{Perms} \mid \exists c \in \phi_{method}(n), c' \in \phi_{perm}(\text{perm}) : c' \subseteq c\}$$

5.2.3 Permission Checking

We adopt the semiring $\mathcal{I}_{check} = (D_{check}, \oplus_{check}, \otimes_{check}, 0, T)$ in [14] given a a PER (Partial Equivalence Relation)-based abstraction with 2-point domain $\{\text{ANY}, \text{ID}\}$, where $D_{check} = \{\lambda x.\text{ANY}, \lambda x.\text{ID}, 0, T\}$ with the ordering $\lambda x.\text{ANY} \sqsubseteq T \sqsubseteq \lambda x.\text{ID} \sqsubseteq 0$.

Definition 34 (Modeling Policy Checking) Given a context-sensitive call graph $G_{cx} = (G, \phi_{edge})$ where $G = (N, E, s, n_{check})$, we define a conditional weighted pushdown system $\mathcal{W}_{check} = (\mathcal{P}_{check}, \mathcal{I}_{check}, f_{check})$, where $\mathcal{P}_{check} = (\{\cdot\}, \Gamma_{check}, \mathcal{C}_{check}, \mathcal{A}_{check}, \{\cdot, s\})$ with $\Delta_{check} = \Delta \cup \Delta_{cp}$ and $\Gamma_{check} = \Gamma \cup \{e_{cp}, x_{cp}\}$. $\Delta_c$ is defined in Def. 22 and $\Delta_{cp}$ is constructed as follows, for each perm in Perms, we have

$$\begin{cases} \delta : e_{cp} \xrightarrow{L \& C} x_{cp} \in \Delta_{cp} & f_{check}(\delta) = \lambda x.\text{ID} \\ \delta : e_{cp} \xrightarrow{(L)\& C} x_{cp} \in \Delta_{cp} & f_{check}(\delta) = \lambda x.\text{ANY} \end{cases}$$

where $L = (\alpha^*) + (\alpha^*)[\beta \alpha (\Gamma_{check}^*)]$ and $L$ is the complement of $L$, with

- $\alpha = \{\{n,l,r\} \in \Gamma_{check} \mid \text{perm} \in \text{policy}(n), n \in N\}$,
- $\beta = \alpha \cap \{\{n,l,r\} \in \Gamma_{check} \mid m = n_{priv}\}$,
- $C = \Gamma_{check}^*(\zeta_0 + \zeta_1 + \cdots + \zeta_k)\Gamma_{check}^*$, where $\{\zeta_0, \zeta_1, \cdots, \zeta_k\} = \{\zeta \mid \text{perm} \in \text{PermReqs}(\zeta)\}$ for $k \in \mathbb{N}$.

Definition 35 (Algorithm for Policy Checking) Given a domain $dm \in \text{Domain}$, and let $\{\zeta_0, \zeta_1, \cdots, \zeta_k\} = \text{AnalysisPoints}(dm)$. We compute

$$\text{result} = \text{MOVP}(S, T, \mathcal{W}_{check})$$

where $S = \{\langle \cdot, s \rangle\}$ and $T = \{\langle \cdot, x_{cp} \omega \rangle \mid \omega = \Gamma_{check}^*(\zeta_0 + \zeta_1 + \cdots + \zeta_k)\Gamma_{check}^*\}$. We say access control for $dm$ may fail if result $= \lambda x.\text{ANY}$ and always succeed if result $= \lambda x.\text{ID}$.
6 Related Work

From the theoretical aspect, Banerjee et al. in [1] gave a denotational semantics and thereby proved the equivalence of eager and lazy evaluation for stack inspection. They further proposed a static analysis of safety property, and also identified program transformations that help remove redundant runtime access control checks. The problem to decide whether a program satisfies a given policy properties via stack inspection, was proved intractable in general by Nitta et al. in [15]. They showed that there exists a solvable subclass of programs which precisely model programs containing checkPermission of Java 2 platform. Moreover, the study concluded the computational complexity of the problem for the subclass is linear time in the size of the given program.

Chang et al. [4] provided a backward static analysis to approximate redundant permission checks with must-fail stack inspection and success permission checks with must-pass stack inspection. This approach was later employed in a visualization tool of permission checks in Java [9]. But the tool didn’t provide any means to relieve users from the burden of deciding access rights. In addition to a policy file, users were also required to explicitly specify which methods and permissions to check. Two control flow forward analysis, Denied Permission Analysis and Granted Permission Analysis, were defined by Bartoletti et al. [2] [3] to approximate the set of permissions denied or granted to a given Java bytecode at runtime. Outcome of the analysis were then used to eliminate redundant permission checks and relocate others to more proper places in the given program.

Koved et al. in [10] proposed a context-sensitive, flow-sensitive, and context-sensitive (1-CFA) data flow analysis to automatically estimate the set of access rights required at each program point. In spite of notable experimental results, the study suffered from a practical matter, as it does not properly handle strings in the analysis. Being a module of privilege assertion in a popular tool – IBM Security Workbench Development for Java (SWORD4J) [8], the interprocedural analysis for privileged code placement [17] tackled three neat problems: identifying portions of codes that necessary to make privileged, detecting tainted variables in privileged codes, and exposing useless privileged blocks of codes, by utilizing the technique in [10].

In aforementioned works, they all assume permissions required at every checkPermission(perm) point. In other words, they either ignored or employed limited computation of String parameters. Correspondingly, the access rights analysis become too conservative, e.g., many false alarms may be produced in policy checking.

To the best of our knowledge, the modular permission analysis proposed in [7] is the most relevant to our work. On one hand, it was also concerned with automatically generating security polices for any given program, with particular attention on the principle of least privilege. On the other hand, they were the first to attempt to reflect the effects of string analysis in access rights analysis in terms of slicing. The authors also developed a tool Automated Authorization Analysis (A3) to assess the precision of permission requirements for stack inspection. However, their algorithms are based on a context-insensitive call graph and the analysis results can be polluted by invalid call call paths. Moreover, their slicing algorithms are also context-insensitive.

Although stack inspection is widely adopted as a simple and practical model in stack-based access control, it has a number of inherent flaws, e.g., an unauthorized code which is no longer in the call stack may be allowed to affect the execution of security-sensitive code. A worth highlighting alternate model is IBAC (Information-based Access Control) proposed by Pistoia et al. in [16] for programs with access control based on execution history.
7 Conclusions

We have presented a formal framework of access rights analysis for Java programs, and analysis algorithms of automatically generating security policies for an Java program and analysis of policy checking on whether stack inspection from the concerned domain always succeed or may fail, given a policy file. Our analysis integrates with both points-to analysis and string-analysis in a unified analysis framework. All analysis algorithms are novelty designed in the framework of conditional weighted pushdown systems, which is modeled after combining a context-sensitive call graph and dependency graph of the target program and precisely identifies permission requirements at checkpoints of stack inspection. We expect a good precision of our analysis, which means low false alarms in policy checking and high compliance with the principle of least privilege. An available tool that can automatically generating security policies for Java applications doesn’t exist so far. It would be interesting to evaluate algorithms proposed in this paper, and to put the techniques into practice by settling the scalability issue.

8 Bibliography

References


A Formal Framework for Access Rights Analysis

Figure 3: An Example for Dependency Graph with Call Graph

expr\_1: abbreviates "new FilePermission(arg\_m, "read")"
expr\_2: abbreviates "new SocketPermission("domain: 80", "connect")"
m\_i(0 \leq i \leq 4), s, n\_check:
methods grouped by the dotted circles
→: edges in the call graph
→→: edges in the dependency graph
check\_Permission(...): final nodes of the dependency graph
npv = new...: initial nodes of the dependency graph
l\_1, ..., l\_7: line numbers for method invocation statements