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 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, 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 inspection. 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 novelly designed in the framework of conditional weighted pushdown systems, and are expected to achieve a high level of precision in the literature.

1 Introduction

Access control is often the first step to protect safety-critical systems. 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* that are granted to codes grouped by different domains, and developers can set checkpoints in their programs, and access control is enforced dynamically at runtime by *stack inspection*. When a 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 a caller that is marked as "privileged". 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.

From a practical perspective, such runtime inspection may cause a high overhead cost. If access control at some checkpoints always succeed at runtime, the runtime overhead can be reduced by removing such redundant checkpoints. Moreover, to our knowledge of practiced approaches, nowadays policy files are still generated manually by developers based on domain-specific knowledges, 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 behaviors, the application could malfunction due to the authorization failures given the misconfigured policies. On the other hand, if the policy file is too conservative, i.e., some codes

*Corresponding author.

Submitted to:

are granted permissions than necessary, it violates the so-called PLP (Principle of Least Privilege), and such codes become vulnerable points and can be manipulated by malicious attacks.

Example 1.1 (Semantics of Java Stack Inspection) *Consider the code snippet in Fig. 1.1 that we borrow from [7] and modify to explicitly show the control flows to checkpoints of stack inspection.*

```
1  public class Lib {
2    private static final String dir = "C:";
3    private static final String logFile = "/log.txt";
4    private static final String domain = "JAIST.AC.JP";
5    public static void createSocket(final String host) throws Exception {
6      checkConnect(host, 8080);
7      Priv op = new Priv(dir, logFile);
8      AccessController.doPrivileged(op);
9      System.out.println("Enough permissions granted.");
10   }
11   public static void checkConnect(String host, int port) throws Exception {
12   SocketPermission p = new SocketPermission(host+":"+port,"connect");
13   AccessController.checkPermission(p);
14   }}
15 class Priv implements PrivilegedExceptionAction {
16   private final String dir;
17   private final String name;
18   Priv(final String dir, final String name) {
19     this.dir = dir;
20     this.name = name;
21   }
22   public Object run() throws Exception {
23     String fn = dir + File.separator + name.substring(1);
24     checkAccess(fn);
25   }
26   public static void checkAccess(String fn) throws Exception {
27   FilePermission p = new FilePermission(fn,"write");
28   AccessController.checkPermission(p);
29   }}
30 public class Faculty {
31   public void connectToFaculty() throws Exception {
32     String host = Lib.domain.toLowerCase() + "/faculty";
33     Socket s = Lib.createSocket(host);
34   }}
35 public class Student {
36   public void connectToStudent() throws Exception {
37     String host = Lib.domain.toLowerCase() + "/student";
38     Socket s = Lib.createSocket(host);
39   }}
```

Figure 1: An Example for Java Stack Inspection

In Fig. 1.1, 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 require to connect to their corresponding domains by creating a socket (Line 33 and 38, 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 $perm_f$ = "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 28. But note that Student and Faculty reside on the current call stack yet 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 1.1, analysis on security policies 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 23, 32, 37), substring operation (Line 23), case conversion (Line 32, 37), etc., string analysis plays an important role. It is known challenging to design a precise and scalable algorithm for either string or points-to 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 [13] that are capable of reasoning properties over the stack.

Our analysis framework has many novel features. First of all, 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. Moreover, instead of conducting analysis on call graphs as usual, we model the analysis problem in terms of a type of context-sensitive call graph, taking into account the dynamic features of Java languages. Another reason why call graph does not suffice for the analysis is that 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. In view of the aforementioned reasons, 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. We expect our analysis algorithms enjoy a high precision in the literature.

The rest of our paper is organized as follows. Section 2 recalls conditional weighted pushdown model checking. Section 3 defines abstractions on the program and the system of security policy, as well as pre-assumed points-to analysis and string analysis. Section 4 formalizes our idea of policy generation and policy checking regarding stack inspection problems, and Section 5 gives realization algorithms in the framework of conditional weighed pushdown systems. Related work is discussed in Section 6, and we conclude in Section 7.

2 Preliminary

Definition 2.1 *A **pushdown system** $\mathscr{P}$ is $(P,\Gamma,\Delta,p_0,\omega_0)$, where P is a finite set of control locations, Γ 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 $(p,\gamma,q,\omega)\in\Delta$ is written as $\langle p,\gamma\rangle\hookrightarrow\langle q,\omega\rangle$. 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 $\langle p,\gamma\omega'\rangle\Rightarrow\langle q,\omega\omega'\rangle$ for each $\omega'\in\Gamma^*$ if $\langle p,\gamma\rangle\hookrightarrow\langle q,\omega\rangle$, and the reflective and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.*

A pushdown system can be normalized (or simulated) by a pushdown system for which $|\omega|\leq 2$ for each transition rule $\langle p,\gamma\rangle\hookrightarrow\langle q,\omega\rangle$ [19]. In sequel, we always assume such normalized forms.

Definition 2.2 *A **bounded idempotent semiring** $\mathscr{S}$ is $(D,\oplus,\otimes,\overline{0},\overline{1})$, where $\overline{0},\overline{1}\in D$, and*

1. *$(D,\oplus)$ is a commutative monoid with $\overline{0}$ 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 $\overline{1}$ 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 $a\in D, a\otimes\overline{0}=\overline{0}\otimes a=\overline{0}$;*
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. 2.2, we have that $\overline{0}$ is the greatest element. From the standpoint of abstract interpretation, PDSs model the (recursive) control flows of the program, weight elements encodes transfer functions, $\otimes$ corresponds to 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 2.3 *A **weighted pushdown system** $\mathscr{W}$ is $(\mathscr{P},\mathscr{S},f)$, where $\mathscr{P}=(P,\Gamma,\Delta,p_0,\omega_0)$ is a pushdown system, $\mathscr{S}=(D,\oplus,\otimes,\overline{0},\overline{1})$ is a bounded idempotent semiring, and $f:\Delta\to D$ is a weight assignment function.*

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

Definition 2.4 *Given a weighted pushdown system $\mathscr{W}=(\mathscr{P},\mathscr{S},f)$ where $\mathscr{P}=(P,\Gamma,\Delta,p_0,\omega_0)$, and regular sets of configurations $S,T\subseteq P\times\Gamma^*$, the **meet-over-all-path** problem computes*

$$MOVP(S,T)=\oplus\{val(\sigma)\mid\sigma\in path(s,t),s\in S,t\in T\}$$

We refer to weighted pushdown model checking by $\mathsf{MOVP}(S,T,\mathscr{W})$ sometime when there are more than one weighted pushdown systems. WDPSs 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 2.5 *A **conditional pushdown system** is $\mathscr{P}_c = (P, \Gamma, \Delta_c, \mathscr{C}, p_0, \omega_0)$, where P is a finite set of control locations, Γ is a finite stack alphabet, $\mathscr{C}$ is a finite set of regular languages over Γ, $\Delta_c \subseteq P \times \Gamma \times \mathscr{C} \times P \times \Gamma^*$ 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 $\langle p, \gamma \rangle \overset{L}{\hookrightarrow} \langle q, \omega \rangle$. A computation relation $\Rightarrow_c$ on configurations is defined such that $\langle p, \gamma\omega' \rangle \Rightarrow_c \langle q, \omega\omega' \rangle$ for all $\omega' \in \Gamma^*$ if there exists a transition $r : \langle p, \gamma \rangle \overset{L}{\hookrightarrow} \langle q, \omega \rangle$ and $\omega' \in L$, written as $\langle p, \gamma\omega' \rangle \Rightarrow_c \langle q, \omega\omega' \rangle$. The reflective and transitive closure of $\Rightarrow_c$ is denoted by $\Rightarrow_c^*$. We define $cpre^*(C) = \{c' \mid c' \Rightarrow_c^* c, c \in C\}$ and $cpost^*(C) = \{c' \mid c \Rightarrow_c^* c', c \in C\}$ for any $C \subseteq P \times \Gamma^*$.*

Definition 2.6 *A **conditional weighted pushdown system** (CWPDS) $\mathscr{W}_c$ is $(\mathscr{P}_c, \mathscr{S}, f)$, where $\mathscr{P}_c = (P, \Gamma, \mathscr{C}, \Delta_c, p_0, \omega_0)$ is a conditional pushdown system, $\mathscr{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ 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 2.4 to CWPDSs and refer it by MOVP.

3 Abstraction and Prerequisites

3.1 Abstraction of Java Programs

Definition 3.1 (Program Points) *We denote by $\mathscr{M}$ the set of all methods in a program, and by $\mathscr{L}$ the set of program line numbers each of which contains a statement. Let $Tag = \{c, r\}$. A program point is characterized by its enclosing method $m \in \mathscr{M}$, line number $l \in \mathscr{L}$, and a tag $\in Tag$, and the set of program points is denoted by $ProgPoint \subseteq \mathscr{M} \times \mathscr{L} \times Tag$. Moreover, we denote by*

- *$CallSite \subseteq \mathscr{M} \times \mathscr{L} \times \{c\}$: the set of call sites, such that l contains a method invocation for any $(m, l, c) \in CallSite$; and*
- *$RetPoint \subseteq \mathscr{M} \times \mathscr{L} \times \{r\}$: the set of return points of method invocations, such that $(m, l, r) \in RetPoint$ is the unique return point of a call site $(m, l, c) \in CallSite$; and*

We use variables χ to range over *ProgPoint* and ζ to range over *RetPoint*. Let $\mathbb{N}$ denote the set of natural numbers. For any finite set $S = \{s_0, \ldots, s_k\}$, we define $\Pi\, S = \{s_{i_0} s_{i_1} \ldots s_{i_k} \mid \{i_0, \ldots, i_k\}$ is a permutation of $S\}$. For any word $\omega = s_{i_0} s_{i_1} \ldots s_{i_j} \in S^*$ with $0 \le j \le k$ and $0 \le i_j \le k$, we define $\Sigma(\omega) = \{s_{i_0}, s_{i_1}, \ldots, s_{i_j}\}$.

Definition 3.2 (Call Graph) *A call graph $G = (N, E, s, n_{check})$ is a directed graph, where $N \subseteq \mathscr{M}$ is the set of nodes, $E \subseteq \mathscr{M} \times CallSite \times \mathscr{M}$ is the set of edges, $s \in N$ is the initial node with no incoming edges, and $n_{check} \in N$ is the final state with no outgoing edges which in particular denotes the method checkPermission from the class AccessController. We also denote by $n_{priv} \in N$ the method doPrivileged from the class AccessController. We write $n \to n'$ if $(n, \chi, n') \in E$, and let $\to^*$ be the transitive and reflective closure of $\to$.*

G is built by call graph construction algorithms which is known to be cyclically dependent of points-to analysis. If call graph construction detects that m calls m' at line l, we have $(m, (m, l, c), m') \in E$.

Definition 3.3 (Calling Contexts) *We denote by $Context \subseteq RetPoint^*$ the set of program calling contexts in terms of call site strings. Given a call graph $G = (N, E, s, n_{check})$, the calling contexts of a method m is defined by $\phi : \mathscr{M} \to 2^{Context}$, such that*

$$\phi(m) = \{\zeta_k \ldots \zeta_1 \zeta_0 \in Context \mid \exists k \in \mathbb{N} : m_0 = s,\ m_{k+1} = m, \\ (m_i, \chi_i, m_{i+1}) \in E, \chi_i = (m_i, l_i, c), \zeta_i = (m_i, l_i, r),\ \text{for each } 0 \le i \le k\}$$

The calling contexts of a method (equivalently, local variables within this method scope) is the (finite) set of finite yet unbounded sequences of return points that may lead to m from the program entry.

Definition 3.4 (Abstract Calling Contexts) *We denote by $AbsCtxt \subseteq 2^{RetPoint}$ the set of abstract program calling contexts in terms of sets of call sites along each call sequence, as an over approximation of calling contexts $Context$. An abstraction function $\alpha : Context \to AbsCtxt$ is defined by $\alpha(c) = \Sigma(c)$ for each $c \in Context$. A concretization function $\gamma : AbsCtxt \to 2^{Context}$ is defined by $\gamma(C) = \Pi\ C$ for each $C \in AbsCtxt$. The powerset extension of α and γ are denoted by $\tilde{\alpha} : 2^{Context} \to 2^{AbsCtxt}$ and $\tilde{\gamma} : 2^{AbsCtxt} \to 2^{Context}$, respectively.*

The abstract calling contexts of a method m is defined by $\phi_{method} : \mathscr{M} \to 2^{AbsCtxt}$, such that

$$\phi_{method}(m) = \{\Sigma(ctxt) \mid ctxt \in \phi(m), and\ c' \notin \phi_{method}(m)\ if\ c' \subseteq c\ and\ c \in \phi_{method}(m)\}$$

It is not hard to conclude that $(2^{Context}, \tilde{\alpha}, \tilde{\gamma}, 2^{AbsCtxt})$ is a Galois connection in abstract interpretation.

3.2 Pre-assumed Analysis

Our framework for access rights analysis assumes context-sensitive points-to analysis, context-sensitive string analysis. The precision of our analysis depends on the precision of points-to and string analysis.

Definition 3.5 (Context-Sensitive Points-to Analysis) *Given a reference variable v of the method $m \in \mathscr{M}$, a context-sensitive points-to analysis, denoted by $pta(v)$, returns the finite set of abstract heap objects that v may refer to at runtime under certain calling contexts. Each object $o \in pta(v)$ is represented as a triplet $o = (type, loc, c)$, where type is the object type, loc is the object allocation site, and $c \in \phi_{method}(m)$ is the calling contexts under which the object is constructed, and $\bigcup_{(type,loc,c) \in pta(v)} \{c\} = \phi_{method}(m)$.*

Definition 3.6 (Context-Sensitive String Analysis) *Given a string variable v of the method $m \in \mathscr{M}$, a context-sensitive string analysis, denoted by $sa(v)$, returns the finite set of string constants that v may contain at runtime under certain calling contexts. Each element in $sa(v)$ is represented as a pair (sv, c), where sv is the string value and $c \in \phi_{method}(m)$ is the calling contexts under which sv is constructed, and $\bigcup_{(sv,c) \in sa(v)} \{c\} = \phi_{method}(m)$.*

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 if possible and as such has an inherent limit if there exist recursive procedural calls. The latter refers to model the program as a pushdown system and the analysis problem as model checking problems, e.g., in the framework of WPDSs. 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]. For each reference variable v of the method m, we can compute $pta(v)$ by $\bigcup_{T_{ctxt}} \mathsf{MOVP}(S, T_{ctxt})$ where S is the source configurations, and $T_{ctxt} = \{\langle v, m\omega \rangle \mid \Sigma(\omega) \subseteq ctxt\}$ for each $ctxt \in \phi_{method}(m)$. For string-analysis, the analysis in [5] based on context-cloning (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. Given a call graph $G = (N, E)$ that is commonly the starting point of program analysis. We construct another graph $G_{clone} = (N_{clone}, E_{clone})$, where $N_{clone} \subseteq$

$2^{AbsCtxt} \times N$ is the set of nodes, and $E_{clone} \subseteq N_{clone} \times CallSite \times N_{clone}$ is the set of edges. For each $n \in N$, $(ctxt, n) \in N_{clone}$ for each $ctxt \in \phi_{method}(n)$, and $((ctxt, n), (ctxt', n')) \in E_{clone}$ if $ctxt \subseteq ctxt'$ and $(n, n') \in E$ for any $(ctxt, n), (ctxt', n') \in N_{clone}$. 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 (i.e., product with the call graph) in this manner, and the most influential string analysis 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 3.7 (Policy System) *Let Domain denote a finite set of protection domains, and Perms denote the universe of all permissions involved in the given program. We denote by*

- $\mathtt{dom}: \mathscr{M} \longrightarrow$ *Domain the mapping from methods to their protection domains.*
- $\mathtt{perm}: Domain \longrightarrow 2^{Perms}$ *the mapping that grants a set of permissions to each protection domain.*

Let $\mathtt{perm}$ *be extended element-wise and let* $policy = \mathtt{perm} \circ \mathtt{dom}$.

Recall that, all classes in a protection domain are granted the same set of permissions. Consequently, all methods and all program points in it will possess the same permissions granted. Especially, all methods belonging to the system domain, e.g., method doPrivileged from the class AccessController, are granted all permissions in *Perms*.

Definition 3.8 (Check Points) *We define CheckPoint as the set of call sites that directly call the method checkPermission, by* $CheckPoint = \{\chi \mid \exists n \in N, \chi \in CallSite : (n, \chi, n_{check}) \in E\}$.

Let $\phi_{perm} : Perms \to 2^{Context}$ be a mapping from each permission to the calling contexts under which the permission is constructed. We generate *Perms* and ϕ_{perm} as follows. Initially, $Perms = \emptyset$, and $\phi_{perm} = \lambda x.\emptyset$. For each call site $\chi = (m, l, c) \in CheckPoint$ where l is supposed to contain the expression of "`checkPermission(pv)`", we first call points-to analysis $pta(\mathtt{pv})$. For each $(\mathtt{Type}, loc, c) \in pta(\mathtt{pv})$, the heap allocation site referred to by loc is supposed to contain expressions in one of the following form according to the Java API specification,

$$\begin{cases} \texttt{npv = new Type(target,action)} & (1) \\ \texttt{npv = new Type(target)} & (2) \\ \texttt{npv = new Type()} & (3) \end{cases}$$

Assume loc belongs to the method m'. We add each of the following permission $perm$ to *Perms*,

$$perm = \begin{cases} (\mathtt{Type}, sv_1, sv_2) & \text{where } (sv_1, c_1) \in sa(\mathtt{target}), (sv_2, c_2) \in sa(\mathtt{action}), c_1 = c_2, \\ & \text{and } \phi_{perm}(perm) = \phi_{perm}(perm) \cup \{c_1\} \text{ for (1)} \\ (\mathtt{Type}, sv) & \text{where } (sv, c') \in sa(\mathtt{target}) \\ & \text{and } \phi_{perm}(perm) = \phi_{perm}(perm) \cup \{c'\} \text{ for (2)} \\ \mathtt{Type} & \text{where } \phi_{perm}(perm) = \phi_{perm}(perm) \cup \phi_{method}(m') \text{ for (3)} \end{cases}$$

4 Formalization

Definition 4.1 (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 $\phi_{edge} : E \to 2^{AbsCtxt}$, such that for each node $n \in N$,*

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

We define a mapping $\phi_{route} : (\to^*) \to 2^{AbsCtxt}$ by, for each $n \to^i n'$,

$$\phi_{route}(n \to^i n') = \begin{cases} \phi_{edge}(n \to n') & \text{if } i = 1 \\ \{ctxt \cup ctxt' \mid ctxt \in \phi_{route}(n \to^{i-1} n''), ctxt' \in \phi_{edge}(n'' \to n')\} & \text{if } i > 1 \end{cases}$$

Definition 4.2 (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 paths from s to a node $n \in N$ by*

$$path(n) = \{e_0 e_1 \dots e_k \mid \exists k \in \mathbb{N} : n_0 = s,\ n_{k+1} = n,\ e_i = (n_i, \chi_i, n_{i+1}) \in E \text{ for each } 0 \le i \le k\}$$

- *the set of subsequences of $path(n)$ that are truncated by the node n_{priv} as*

$$tpath(n) = \{e_0 e_1 e_2 \dots e_k \mid \exists k \in \mathbb{N} : n_0 = s,\ n_{k+1} = n,\ n_0 \to^* n_{priv}, \\ e_0 = (n_{priv}, \chi_0, n_1), e_i = (n_i, \chi_i, n_{i+1}) \text{ for each } 1 \le i \le k\}$$

- *the set of valid paths from s to a node $n \in N$ by*

$$vpath(n) = \{\sigma \in path(n) \mid \exists ctxt \in \phi_{route}(\sigma) : ctxt \subseteq sites(\sigma)\}$$

where given a node $n \in N$, we define $sites(\sigma) = \{\zeta = (n, l, r) \mid e = (n, \chi, n') \in \Sigma(\sigma)$ and $\chi = (n, l, c)\}$ for a path $\sigma = e_0 e_1 ... e_k \in path(n)$ with $k \in \mathbb{N}$.

A context-sensitive call graph is constructed during call graph construction, given a context-sensitive points-to analysis. One such algorithm is given in [13], where for each call edge $e \in E$, $ctxt \in \phi_{edge}(e)$ specifies a calling context under which e is valid. In Java, due to polymorphism and late biding, that target method of a dynamic dispatched call (e.g., $r.fun(\cdots)$) depends on the runtime type of receiver objects (i.e., r). Therefore, a call edge is conditioned by receivers' points-to information (which is further conditioned by calling-contexts), so does a path in the call graph. We refer to [13] for details, but illustrate in Example 1.1 the semantics for "privileged" codes specific to access control.

Example 4.3 *Consider the code snippet in Fig. 2 that consist in methods m_1, m_2, OnePrivAction.run(), AnotherPrivAction.run(), and n_{priv}, n_{check}. Methods are grouped by dotted circles. OnePrivAction and AnotherPrivAction are classes that implement the interface PrivilegedAction. There are call edges $\{e_1, \cdots, e_6\}$, e.g., $e_1 = (m_1, (m_1, l_2, c)$ and $e_5 = (OnePrivAction.run, (OnePrivAction.run, l_5, c), n_{check})$. We do not explicitly show the call site inside n_{priv}. Since doPrivilege and checkPermission are static, $\phi_{edge}(e_i) = \emptyset$ for $i \in \{1, 2, 5, 6\}$. Assume $(OnePrivAction, l_1, ctxt) \in pta(x)$ and $(AnotherPrivAction, l_3, ctxt') \in pta(y)$. We have $ctxt \in \phi_{edge}(e_3)$ and $ctxt' \in \phi_{edge}(e_4)$.*

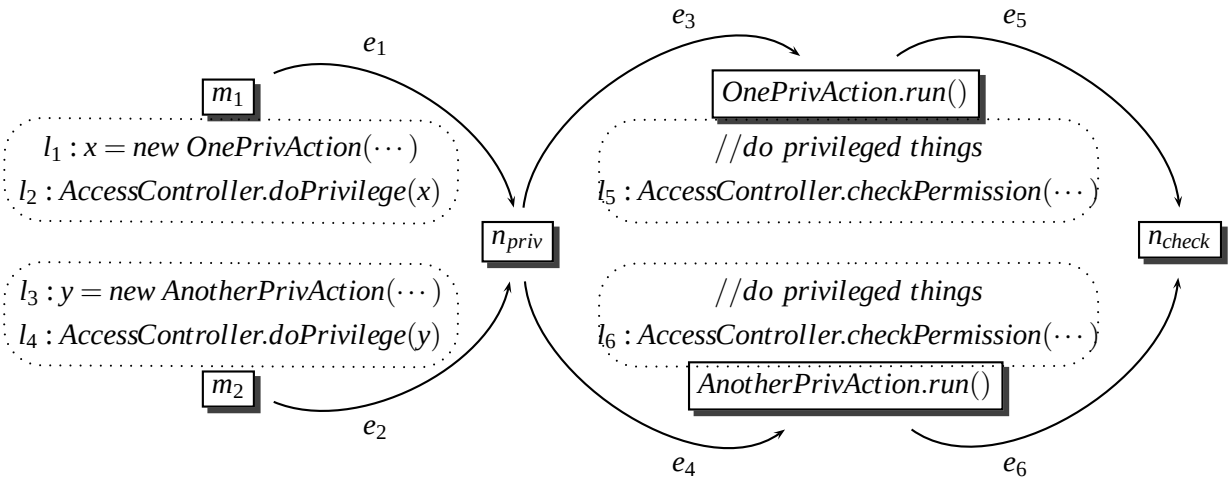


Figure 2: An Example for Context-Sensitive Call Graph

Definition 4.4 (Dependency Graph) *Given a program in SSA (Static Single Assignment) form. Let T_{perm} denote the class (or type) `Permission` or any of its subclassses. Let $\mathscr{L}_{alloc} \subseteq \mathscr{L}$ be the set of program lines that allocate objects of T_{perm}, and let $AllocPerm \subseteq \mathscr{M} \times \mathscr{L}_{alloc}$.*

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

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

Definition 4.5 (Dependency Paths) *Give 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*

$$dpath(n) = \{e_0 e_1 \dots e_k \mid \exists k \in \mathbb{N} : n_0 \in S_{dep},\ n_{k+1} = n,\ e_i = (n_i, n_{i+1}) \in E_{dep} \text{ for each } 0 \le i \le k\}$$

Definition 4.6 (Relate Dependency Paths to Permissions) *Given a context-sensitive call graph $G_{cs} = (G, \phi_{edge})$ where $G = (N, E, s, n_{check})$, and a dependency graph $G_{dep} = (N_{dep}, E_{dep}, S_{dep}, F_{dep})$.*

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

$$\omega_r = \zeta_{i_0} \cdots \zeta_{i_j}$$

where $0 \le i_0 \le i_1 \cdots \le i_j \le k+1$, $0 \le j \le k+1$, and for each $i_m \in \{i_0, \cdots, i_j\}$, $\zeta_{i_m} = (m_{i_m}, l_{i_m}, r)$, $(n_{i_m - 1}, (m_{i_m}, l_{i_m})) \in E_{dep}$, and $l_{i_m - 1}$ is a method return statement. Specifically, $\omega_r = \varepsilon$ if such i_m does not exist .

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

$$\omega_l = \chi_0 \cdots \chi_h$$

*Let $[_{(m,l)}$ denote $(m,l,c) \in CallSite$ and let $]_{(m,l)}$ denote $(m,l,r) \in RetPoint$. The set of all such parentheses induced by $CallSite \cup RetPoint$ is denoted by Σ_{cfl}. We say π **matches with** σ if $\omega_l \omega_r$, called a **valid flow**, is a context-free language over Σ_{cfl}. The set of all such σ for π is denoted by $match(\pi)$.*

*Given a permission $perm \in Perms$, we say π **relates to** perm, if there exists (i) $\sigma \in match(\pi)$, i.e., π matches with σ, and (ii) $ctxt \in \phi_{perm}(perm)$ such that $ctxt = sites(\sigma)$.*

Definition 4.7 (Relate Valid Paths to Dependency Paths) *Given a context-sensitive call graph $G_{cs} = (G, \phi_{edge})$ where $G = (N, E, s, n_{check})$, and a dependency graph $G_{dep} = (N_{dep}, E_{dep}, S_{dep}, F_{dep})$.*

Given a node $n \in N_{dep}$, and a dependency path $\pi = e_0 e_1 \cdots e_k \in dpath(n)$ for $k \in \mathbb{N}$, 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 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 σ **relates to** π if there exists a path $\sigma' \in match(\pi)$ such that $nodes(\sigma) \subseteq nodes(\pi) \cup nodes(\sigma') \cup \{n_{check}\}$.*

Definition 4.8 (Policy Generation) *We define $policy : \mathscr{M} \to 2^{Perms}$ by, for each valid path $\sigma \in vpath(n_{check})$, and each dependency path $\pi \in dpath(n)$ for each $n \in F_{dep}$*

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

Note that both π and σ in Def. 4.8 can be infinitely many.

Definition 4.9 (Policy Checking) *Given a $policy : \mathscr{M} \to 2^{Perms}$ and a $policy' : \mathscr{M} \to 2^{Perms}$ generated by Def.4.8. All stack inspections triggered in the program always succeed if $policy'(m) \subseteq policy(m)$ for each $m \in \mathscr{M}$, and may fail otherwise.*

Example 4.10 *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 shown for which each node is grouped by the dotted circles and connected by the solid arrows. The following permissions are involved in this example.*

$$\begin{cases} perm_1 : SocketPermission(\text{“}domain:80\text{”}, \text{“}connect\text{”}) & at \text{“}npv = expr''_2 \\ perm_2 : FilePermission(\text{“}public\text{”}, \text{“}read\text{”}) & at \text{“}npv = expr''_1 \\ perm_3 : FilePermission(\text{“}personal\text{”}, \text{“}read\text{”}) & at \text{“}npv = expr''_1 \end{cases}$$

where $perm_1$ is created and referred to by methods in the current call stack when the inspection is triggered, whereas $perm_2, perm_3$ are created/passed by finished method calls that 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 the checkpoints. By Def. 4.6 and 4.7, we have the dependency path $\pi : (4)(5)(6)(7)(9)(10)(11)$ relates to the valid path $\sigma : (0)(1)(3)$ and thus to $perm_2$, and the valid path $\sigma' : (0)(11)(12)$ relates to π and thus to $perm_2$. Other permission requirements can be similarly inferred.

5 Realization Algorithms

5.1 Policy Generation

Definition 5.1 (Modelling Context-Sensitive Call Graph) *Given a context-sensitive call graph $G_{cs} = (G, \phi_{edge})$ where $G = (N, E, s, n_{check})$, we define a conditional pushdown system $\mathscr{P}_c = (\{\cdot\}, \Gamma, \mathscr{C}, \Delta_c, \{\cdot\}, s)$, where the set of control locations is a singleton $\{\cdot\}$, the stack alphabet $\Gamma \subseteq \mathscr{M} \cup RetPoint$ is encoded*

from nodes of G and return points. We write $\alpha \overset{C}{\hookrightarrow} \omega$ *for each* $(\cdot,\alpha,C,\cdot,\omega) \in \Delta_c$. Δ_c *is constructed as follows, for each edge* $e = (n,\chi,n') \in E$ *where* $\chi = (n,l,c)$, *let* $\zeta = (n,l,r)$, *we have*

$$n \overset{C_e}{\hookrightarrow} n'\zeta$$

$$\text{where} \qquad C_e = \bigvee_{ctxt=\{\gamma_0,\gamma_1,\cdots,\gamma_{|ctxt|}\}\in\phi_{edge}(e)} \; \bigvee_{\{i_0,i_1,\cdots,i_{|ctxt|}\}\in\Xi(ctxt)} \Gamma^*\gamma_{i_0}\Gamma^*\gamma_{i_1}\Gamma^*\cdots\gamma_{i_{|ctxt|}}\Gamma^*$$

where $\Xi(S)$ *denote the set of all permutations of* $\{0,1,\cdots,|S|\}$ *for a finite set S, and* $\bigvee$ *denote the set union of regular expressions.*

Definition 5.2 (Modeling Dependency Graph) *Give a dependency graph* $G_{dep} = (N_{dep},E_{dep},S_{dep},F_{dep})$, *we define a conditional pushdown system* $\mathscr{P}'_c = (\{\cdot\},\Gamma,\mathscr{C}',\Delta'_c)$, *where* Δ'_c *is constructed as follows, for each edge* $e = (n,n') \in E_{dep}$ *where* $n = (m,l)$ *and* $n' = (m',l')$, *we have*

$$m \overset{C_e}{\hookrightarrow} \varepsilon \text{ and } (m',l',r) \overset{C_e}{\hookrightarrow} m'$$

if l is a method return statement, where $C_e = \Gamma^*$, *i.e., no conditions.*

Definition 5.3 (Program Modeling) *We define a conditional pushdown system* $\mathscr{P}_{prog} = (\{\cdot\},\Gamma,\mathscr{C}_{prog},\Delta_{prog},\{\cdot\},s)$ *where* $\Gamma \subseteq \mathscr{M} \times RetPoint$, $\mathscr{C}_{prog} = \mathscr{C} \cup \mathscr{C}'$, *and* $\Delta_{prog} = \Delta_c \cup \Delta'_c$, *by combining the conditional pushdown system* $\mathscr{P}_c$ *and* $\mathscr{P}'_c$ *generated for* G_{cs} *and* G_{dep}, *respectively.*

Definition 5.4 (Weight Domain) *We define a bounded idempontent semiring* $\mathscr{S}_{gen} = (D_{gen},\oplus_{gen},\otimes_{gen},\overline{0},\overline{1})$, *where* $D_{gen} \subseteq 2^{2^{\mathscr{M}}\times 2^{\mathscr{M}}\times 2^{\mathscr{M}}\times 2^{\mathscr{M}}} \cup \{\overline{0}\}$, *and* $\overline{1} = \{(\emptyset,\emptyset,\emptyset,\emptyset)\}$. *For any* $d,d' \in D_m$, $d \oplus_{gen} d' = d \cup d'$, *and*

$$d \otimes_{gen} d' = \{(M_1 \cup M'_1 \setminus M'_2, M_2, M_3 \cup M'_3, M_4 \cup M'_4) \mid (M_1,M_2,M_3,M_4) \in d, (M'_1,M'_2,M'_3,M'_4) \in d'\}$$

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

Definition 5.5 (Modeling Policy Generation) *We define a conditional weighted pushdown system* $\mathscr{W}_{gen} = (\mathscr{P}_{prog},\mathscr{S}_{gen},f_{gen})$. *For each transition rule* $\delta \in \Delta_{gen}$, $f_{gen}(\delta)$ *is defined as follows,*

- *if* δ *is a push rule* $m \overset{C_e}{\hookrightarrow} m'(m,l,c)$,

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

- *if* δ *is a pop rule* $m \overset{C_e}{\hookrightarrow} \varepsilon$ *and* $((m,l),(m',l')) \in E_{dep}$, $f_{gen}(\delta) = \{(\emptyset,\emptyset,\{m'\},\emptyset)\}$.
- *otherwise* $f_{gen}(\delta) = \overline{1}$

Definition 5.6 (Algorithm for Policy Generation) *Given a conditional weighted pushdown system* $\mathscr{W}_{gen} = (\mathscr{P}_{prog},\mathscr{S}_{gen},f_{gen})$ *constructed by Def. 5.5. For any* $d = (M_1,M_2,M_3,M_4) \in \mathtt{MOVP}(\{\langle\cdot,s\rangle\},T,\mathscr{W}_{prog})$ *where* $T = \{\langle\cdot,n_{check}\omega\rangle \mid \omega \in \Gamma^*\}$, *and* $perm \in Perms$, *we say perm is required by d if there exists* $ctxt \in \phi_{perm}(perm)$ *such that* $ctxt \subseteq M_4$. *For each* $m \in M_1 \setminus M_3$, *we have* $perm \in policy(m)$ *if perm is required by d.*

For each d in Def. 5.6, M_4 is the calling history before stack inspection is triggered, M_1 is the calling history truncated by n_{priv}, M_2 is supposed to be $\emptyset$ by our modeling, and M_3 is methods that do not reside on the current call stack.

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. 4.9, one approach to policy checking is first generating a policy required by passing stack inspections by Def. 4.8, 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 5.7 (Boundary) *Given a call graph $G = (N, E, s, n_{check})$. Let $l : N \to Domain$ be a mapping from methods to their belonging protection domains. A **boundary** of a domain $dm \in Domain$, denoted by $\mathscr{B}(dm)$, is defined by*

$$\mathscr{B}(dm) = \{n \in N \mid (n, \chi, n') \in E, l(n) = dm, l(n) \neq l(n)\}$$

The boundary of a domain dm refers to the set of methods with outgoing edges to methods from different domains, e.g., Java libraries typically.

Definition 5.8 (Analysis Points) *Assume the conditional pushdown system encoded by Def. 5.1. We define **analysis points** of a given domain $dm \in Domain$ by*

$$\begin{aligned} AnalysisPoints(dm) = \{\zeta = (n,l,r) \in RetPoint \mid n \in \mathscr{B}(dm), l \in \mathscr{L}, \\ \exists n' \in \Gamma, \omega, \omega' \in \Gamma^* : \langle \cdot, n'\zeta\omega' \rangle \in cpost^*(\{\langle \cdot, s \rangle\}) \cap cpre^*(\{\langle \cdot, n_{check}\omega \rangle\})\} \end{aligned}$$

5.2.2 Identifying Permission Requirements

Definition 5.9 (Modeling Permission Requirements) *We define a conditional weighted pushdown system $W_{ctx} = (\mathscr{P}_{prog}, \mathscr{S}_{ctx}, f_{ctx})$ where $\mathscr{P}_{prog}$ is the conditional pushdown system defined before, and*

- *the idempotent semiring $\mathscr{S}_{ctx} = (D_{ctx}, \oplus_{ctx}, \otimes_{ctx}, \overline{0}, \overline{1})$, where $D = 2^{2^{\mathscr{M}}} \cup \{\overline{0}\}$, $\overline{1} = \emptyset$, $\oplus_{ctx}$ is set union, and $\otimes_{ctx}$ is element-wise set union;*
- *for each $\delta : \alpha \overset{C}{\hookrightarrow} \omega \in \Delta_{gen}$, $f_{ctx}(\delta) = \{\{n'\}\}$ if $\omega = n\zeta$, $\zeta = (n', l, r)$, and $f_{ctx}(r) = \overline{1}$ otherwise.*

Definition 5.10 (Identifying Permission Requirements) *Given a domain $dm \in Domain$ and an analysis point $\zeta = (n,l,r) \in AnalysisPoints(dm)$, we compute $\phi_{method}(n) = \texttt{MOVP}(S, T, W_{ctx})$ where $S = \{\langle \cdot, s \rangle\}$, $T = \{\langle \cdot, n\omega \mid \omega \in \Gamma^* \}$. We define permission requirements on n by*

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

5.2.3 Permission Checking

We adopt the semiring $\mathscr{S}_{check} = (D_{check}, \oplus_{check}, \otimes_{check}, \overline{0}, \overline{1})$ in [14] given a a PER (Partial Equivalence Relation)-based abstraction with 2-point domain $\{ANY, ID\}$, where $D_{check} = \{\lambda x.ANY, \lambda x.ID, \overline{0}, \overline{1}\}$ with the ordering $\lambda x.ANY \sqsubseteq \overline{1} \sqsubseteq \lambda x.ID \sqsubseteq \overline{0}$.

Definition 5.11 (Modeling Policy Checking) *Given a context-sensitive call graph $G_{cs} = (G, \phi_{edge})$ where $G = (N, E, s, n_{check})$, we define a conditional weighted pushdown system $\mathscr{W}_{check} = (\mathscr{P}_{check}, \mathscr{S}_{check}, f_{check})$, where $\mathscr{P}_{check} = (\{\cdot\}, \Gamma_{check}, \mathscr{C}_{check}, \Delta_{check}, \{\cdot\}, s)$ with $\Delta_{check} = \Delta_c \cup \Delta_{cp}$ and $\Gamma_{check} = \Gamma \cup \{e_{cp}, x_{cp}\}$. Δ_c is defined in Def. 5.1, and Δ_{cp} is constructed as follows, for each perm $\in$ Perms, we have*

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

where $L = (\alpha^*) + (\alpha^*)\beta\alpha(\Gamma^*_{check})$ and $!L$ is the complement of *L, with*

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

Definition 5.12 (Algorithm for Policy Checking) *Given a domain $dm \in Domain$, and let $\{\zeta_0, \zeta_1, \cdots, \zeta_k\} = AnalysisPoints(dm)$. We compute $result = \mathtt{MOVP}(S, T, \mathscr{W}_{check})$, 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 $result = \lambda x.ANY$ and always succeed if $result = \lambda x.ID$.*

6 Related Work

From the theoretical aspect, Banerjee et al. in [1] gived a denotational semantics and hereby 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 code.

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 all aforementioned works, they all assume the 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 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 in this paper a formal framework of access rights analysis for Java programs, including analysis of automatically generating security policies for any given 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 abstract framework. All analysis algorithms are novelly 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 high precision of our analysis, which means low false alarms in policy checking and high compliance with the principle of least privilege.

8 Bibliography

References

[1] Anindya Banerjee & David A Naumann (2001): *A simple semantics and static analysis for Java security*. Technical Report, Stevens Institute of Technology. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.6307&rep=rep1&type=pdf`.

[2] Massimo Bartoletti & Pierpaolo Degano (2001): *Static analysis for stack inspection*. *Electronic Notes in Theoretical Computer Science* 54, pp. 706–80, doi:10.1016/S1571-0661(04)00236-1. Available at http://www.sciencedirect.com/science/article/pii/S1571066104002361.

[3] Massimo Bartoletti & Pierpaolo Degano (2004): *Stack inspection and secure program transformations*. *International Journal of Information*. Available at http://www.springerlink.com/index/P1LL0C8M558B3C7A.pdf.

[4] BM Chang (2006): *Static check analysis for Java stack inspection*. *ACM SIGPLAN Notices* 41(3), p. 40, doi:10.1145/1140543.1140550. Available at http://dl.acm.org/citation.cfm?id=1140543.1140550.

[5] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim & Kyung-Goo Doh (2006): *A practical string analyzer by the widening approach*. In: *Proceedings of the 4th Asian conference on Programming Languages and Systems*, APLAS'06, Springer-Verlag, Berlin, Heidelberg, pp. 374–388, doi:10.1007/11924661_23. Available at http://dx.doi.org/10.1007/11924661_23.

[6] Aske Simon Christensen, Anders Møller & Michael I. Schwartzbach (2003): *Precise analysis of string expressions*. In: *Proceedings of the 10th international conference on Static analysis*, SAS'03, Springer-Verlag, Berlin, Heidelberg, pp. 1–18. Available at http://dl.acm.org/citation.cfm?id=1760267.1760269.

[7] Emmanuel Geay, Marco Pistoia, Barbara G. Ryder & Julian Dolby (2009): *Modular string-sensitive permission analysis with demand-driven precision*. *2009 IEEE 31st International Conference on Software Engineering*, pp. 177–187, doi:10.1109/ICSE.2009.5070519. Available at http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5070519.

[8] Ted Habeck, Larry Koved, Marco Pistoia & Yorktown Heights (2008): *SWORD4J : Security WORkbench Development environment 4 Java*. Technical Report, IBM.

[9] Yoonkyung Kim (2007): *Visualization of permission checks in java using static analysis*. *Information Security Applications*, pp. 133–146. Available at http://www.springerlink.com/index/3r6167p185551545.pdf.

[10] Larry Koved, Marco Pistoia & Aaron Kershenbaum (2002): *Access rights analysis for Java*. In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 37, ACM, pp. 359—-372, doi:10.1145/583854.582452. Available at http://portal.acm.org/citation.cfm?doid=583854.582452http://dl.acm.org/citation.cfm?id=582452.

[11] Ondřej Lhoták & Laurie Hendren (2003): *Scaling Java points-to analysis using SPARK*. In: *Proceedings of the 12th international conference on Compiler construction*, CC'03, Springer-Verlag, Berlin, Heidelberg, pp. 153–169. Available at http://dl.acm.org/citation.cfm?id=1765931.1765948.

[12] Xin Li & Mizuhito Ogawa (2009): *Stacking-based context-sensitive points-to analysis for Java*. *Hardware and Software: Verification and Testing*, pp. 133–149. Available at http://www.springerlink.com/index/W6242785246N3500.pdf.

[13] Xin Li & Mizuhito Ogawa (2010): *Conditional weighted pushdown systems and applications*. *Proceedings of the ACM SIGPLAN 2010 workshop on Partial evaluation and program manipulation - PEPM '10*, p. 141, doi:10.1145/1706356.1706382. Available at http://portal.acm.org/citation.cfm?doid=1706356.1706382.

[14] Xin Li, Daryl Shannon, Indradeep Ghosh, Mizuhito Ogawa, Sreeranga P. Rajan & Sarfraz Khurshid (2008): *Context-Sensitive Relevancy Analysis for Efficient Symbolic Execution*. In: *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, Springer-Verlag, Berlin, Heidelberg, pp. 36–52, doi:10.1007/978-3-540-89330-1_4. Available at http://dx.doi.org/10.1007/978-3-540-89330-1_4.

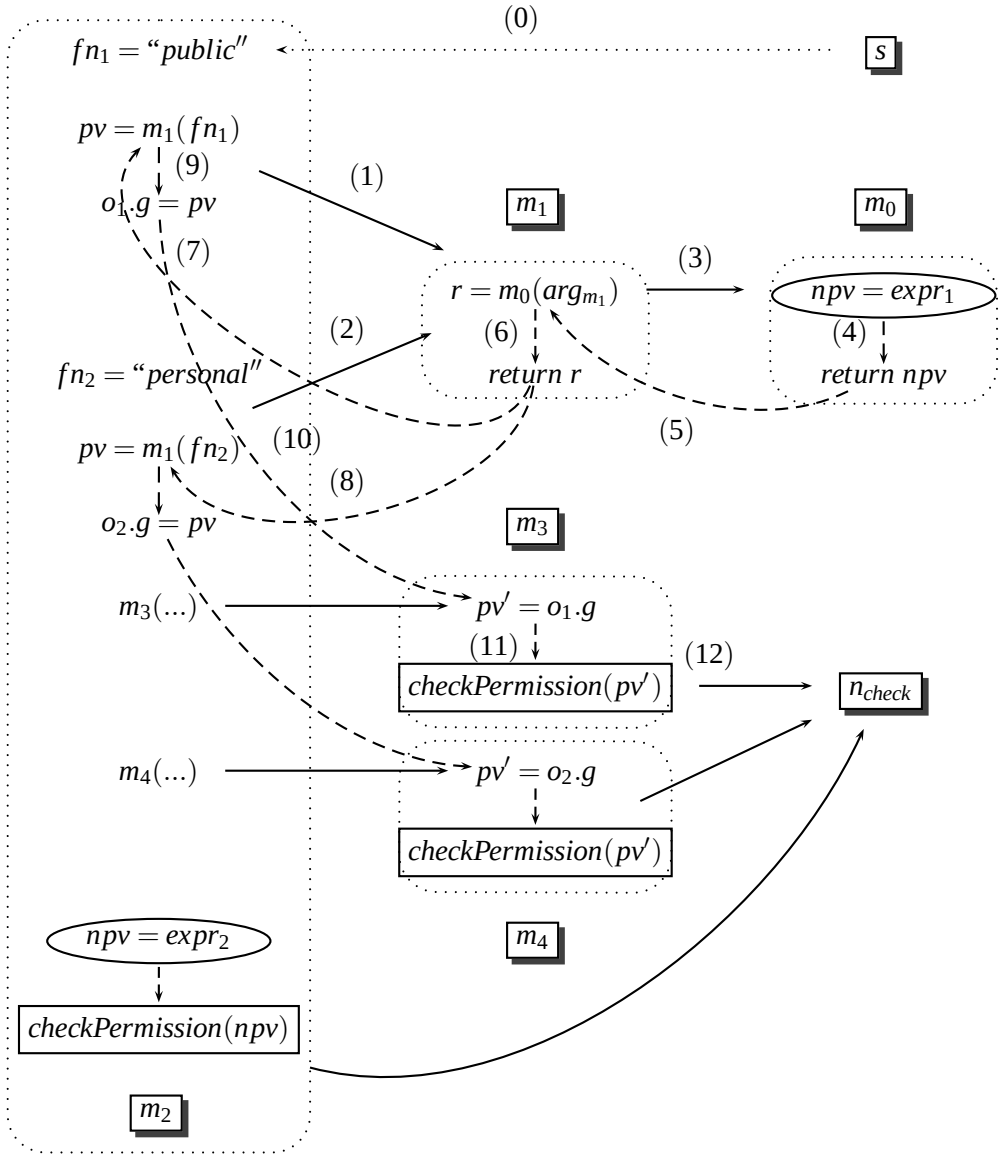
[15] Naoya Nitta & Yoshiaki Takata (2001): *An efficient security verification method for programs with stack inspection*. *Computer and Communications Security*, pp. 68–77. Available at http://dl.acm.org/citation.cfm?id=501994.

[16] Marco Pistoia, Anindya Banerjee & David Naumann (2007): *Beyond stack inspection: A unified access-control and information-flow security model*. *Security and Privacy, 2007*. Available at `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4223221`.

[17] Marco Pistoia, R Flynn & Larry Koved (2005): *Interprocedural analysis for privileged code placement and tainted variable detection*. *ECOOP 2005-Object-Oriented*, pp. 362–386. Available at `http://www.springerlink.com/index/8hhv386p25ee0417.pdf`.

[18] Thomas Reps, Stefan Schwoon, Somesh Jha & David Melski (2005): *Weighted pushdown systems and their application to interprocedural dataflow analysis*. *Science of Computer Programming* 58(1-2), pp. 206–263, doi:`10.1016/j.scico.2005.02.009`. Available at `http://linkinghub.elsevier.com/retrieve/pii/S0167642305000493`.

[19] Stefan Schwoon (2002): *Model-Checking Pushdown Systems*. Ph.D. thesis, Technische Universitat Munchen. Available at `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/schwoon-phd02.pdf`.



$expr_1$:	abbreviates “$new\ FilePermission(arg_{m_1}, \text{“}read\text{”})$”
$expr_2$:	abbreviates “$new\ SocketPermission(\text{“}domain:80\text{”}, \text{“}connect\text{”})$”
$m_i (0 \leq i \leq 4), s, n_{check}$:	methods grouped by the dotted circles
$\longrightarrow$:	edges in the call graph
$-\rightarrow$:	edges in the dependency graph
$checkPermission(\cdots)$:	final nodes of the dependency graph
$npv = new \cdots$:	initial nodes of the dependency graph

Figure 3: An Example for Dependency Graph with Call Graph

.