Modular String-Sensitive Permission Analysis with Demand-Driven Precision

Emmanuel Geay
IBM T. J. Watson Research Center
Hawthorne, NY, USA
egeay@us.ibm.com

Marco Pistoia
IBM T. J. Watson Research Center
Hawthorne, NY, USA
pistoia@us.ibm.com

Takaaki Tateishi
IBM Tokyo Research Laboratory
Tokyo, Japan
tate@jp.ibm.com

Barbara G. Ryder
Virginia Tech
Blacksburg, VA, USA
ryder@cs.vt.edu

Julian Dolby
IBM T. J. Watson Research Center
Hawthorne, NY, USA
dolby@us.ibm.com

Abstract

In modern software systems, programs are obtained by dynamically assembling components. This has made it necessary to subject component providers to access-control restrictions. What permissions should be granted to each component? Too few permissions may cause run-time authorization failures, too many constitute a security hole. We have designed and implemented a composite algorithm for precise static permission analysis for Java and the CLR. Unlike previous work, the analysis is modular and fully integrated with a novel slicing-based string analysis that is used to statically compute the string values defining a permission and disambiguate permission propagation paths. The results of our research prototype on production-level Java code support the effectiveness, practicality, and precision of our techniques, and show outstanding improvement over previous work.

1 Introduction

Operating-system access control allows restricting access to resources based on the identity of the authenticated user executing a program. In modern run-time environments, such as Java and the Common Language Runtime (CLR), programs under execution are obtained by dynamically assembling components. At run time, when different components are dynamically combined to form a program, a component provider could behave as an *active attacker* [16] and violate the integrity of the system by *injecting* into the program a component that performs operations that the system administrator did not intend to authorize. As a result, in such systems, there is the need for a code provider to undergo authentication and authorization much the same way as the user who executes the final program. A component can authenticate itself based on its origin in the network and the digital signature applied by the component provider before distribution. The system administrator assigns "permissions" to authenticated components. A *permission* is the right to access a restricted resource. At run time, when a component attempts, directly or indirectly, to access a restricted resource, the underlying run-time environment demands that the component prove possession of the relevant permission.

A system administrator installing a program must configure its access-control policy by granting or denying permissions to the program components. In the software lifecycle, this challenge is faced also by component developers and providers who publish permission recommendations for their components before distributing them.[1] An overly permissive policy constitutes a violation of the Principle of Least Privilege (PLP) [22], which establishes that a user or program should never be granted more permissions than those it requires to function correctly. If the policy is too restrictive, the program will not function properly due to run-time authorization failures. Source code is not always available, so manual code inspection, besides being complicated, time consuming, and error prone, may not even be an option.

Dynamic analysis is an alternative. With this technique, a component is tested, initially with no permissions. Any attempt by the component to directly or indirectly access a restricted resource will result in a run-time authorization failure. Each failure is logged, the access-control policy is updated by granting the component the missing permissions (assuming that it is safe to do so), and the program is restarted. This process must be iterated until no more

[1] We are currently collaborating with the Eclipse community on the effort of enabling Java security on the Rich Client Platform (RCP) [3].

authorization failures are found. However, there is no guarantee that the access-control policy obtained at the end of this process will be sufficient to execute the program without failures. Absence of a complete suite of test cases can leave some execution paths undiscovered until deployment, thereby exposing the deployed program to unjustified authorization failures. Furthermore, testing a potentially malicious program can harm the underlying system.

Static analysis has the ability to model all the possible paths of execution of a program without executing it. Thus, a static permission analysis can detect all the permission requirements. However, if the analysis is too conservative, it will compute *false positives*—permissions that are not actually needed and that, if granted, will result in PLP violations. Hence, minimizing the number of false positives in permission analysis is crucial. This work combines string analysis [15] with a demand-driven permission-tracking algorithm to reduce the number of false positives.

In summary, this paper makes three major contributions:

1. A Composite Static Analysis Algorithm: Key insights used in the design of this technique include combining a modular library analysis with a library-client permission analysis. The latter is made more precise by the use of string analysis and slicing to find security-sensitive program regions and to disambiguate permission-propagation paths in them for acceptable precision at practical cost. String analysis is also used to resolve string values in permission requirements.

2. A Research Prototype Implementation: Automated Authorization Analysis (A3).

3. Experimental Findings: Using A3 on publicly available benchmarks shows outstanding precision, scalability, and efficiency improvements over the best existing analysis [26].

The remainder of this paper is organized as follows. Section 2 presents a simple Java program demonstrating the need for more precision in permission analysis. Section 3 explains the semantics of correct use of Java permissions, and introduces the concept of stack inspection. Section 4 describes the composite static-analysis algorithm. Section 5 gives specifics of the algorithm pertaining to the A3 implementation. Section 6 contains the empirical findings and their interpretation. Finally, Section 7 presents related work and Section 8, our conclusions.

2 Motivating Example

In Java and the CLR, access control is based on *stack inspection* [6]: when a security-sensitive operation is performed, all the methods currently on the stack are checked to see if their classes have been granted the relevant permission. The Application Programming Interface (API) performing the stack walk is `checkPermission` in Java and `Demand` in the CLR, taking a `Permission` and `IPermission` parameter, respectively. Stack inspection is intended to prevent *confused-deputy attacks* [9], which arise when a component C_1 that was not granted access to a resource r obtains access to r indirectly, by calling into a component C_2 that was granted access to r.

```
public class Lib {
  private static final String dir = "C:";
  private static final String logFileName = "/log.txt";
  private static final int port = 80;
  public static Socket createSocket(final String host)
      throws Exception {
    Socket socket = new Socket(host, port);
    Priv op = new Priv(dir, logFileName);
    FileOutputStream fos = (FileOutputStream)
        AccessController.doPrivileged(op);
    OutputStream bos = new BufferedOutputStream(fos);
    PrintStream ps = new PrintStream(bos, true);
    ps.print("Socket: " + host + ":" + port);
    return socket;
  } }
class Priv implements PrivilegedExceptionAction {
  private final String dir;
  private final String name;
  Priv(final String dir, final String name) {
    this.dir = dir;
    this.name = name;
  }
  public Object run() throws IOException {
    String fn = dir + File.separator + name.substring(1);
    return new FileOutputStream(fn);
  } }
public class Enterprise {
  private final String enterprise = "IBM";
  private final String domain = ".com";
  public void connectToEnt() throws Exception {
    String host = enterprise.toLowerCase() + domain;
    Socket s = Lib.createSocket(host);
  } }
public class School {
  private final String school = "VT";
  private final String domain = ".edu";
  public void connectToSchool() throws Exception {
    String host = school.toLowerCase() + domain;
    Socket s = Lib.createSocket(host);
  } }
```

Figure 1. Sample Code

Figure 1 shows the Java code of two library classes, `Lib` and `Priv`, and two client classes, `Enterprise` and `School`. `Lib` exposes a public API, `createSocket`, which constructs `Socket` objects on behalf of its clients. At run time, the two clients will require `SocketPermissions` to resolve the names and connect to ports `80` of hosts `ibm.com` and `vt.edu`, respectively.

Upon constructing a `Socket`, `Lib` logs the operation to a file. To prevent its clients (now on the stack) from requiring the relevant `FilePermission`—which a maliciously crafted client could misuse to erase the contents

Permissions	Classes			
	`Enterprise`	`School`	`Lib`	`Priv`
`java.net.SocketPermission "ibm.com", "resolve"`	✓		✓	
`java.net.SocketPermission "ibm.com:80", "connect"`	✓		✓	
`java.net.SocketPermission "vt.edu", "resolve"`		✓	✓	
`java.net.SocketPermission "vt.edu:80", "connect"`		✓	✓	
`java.io.FilePermission "C:/log.txt", "write"`			✓	✓

Table 1. Security Policy for Sample Program

of the file or log false information in it—`Lib` creates an instance of `Priv` and passes it to `doPrivileged`, the Java *privilege-asserting* API [6], which modifies the stack-inspection mechanism as follows: at run time, `doPrivileged` invokes the `run` method of that `Priv` object, and when the stack inspection is performed to verify that each caller on the stack has been granted the necessary `FilePermission`, the stack walk recognizes the presence of `doPrivileged` and stops at `createSocket`, without demanding the `FilePermission` of the clients of `Lib`. In the CLR, the privilege-asserting API is `Assert`.

Table 1 shows a minimal security policy that prevents authorization failures. Although this program is simple, configuring its policy requires computing non-trivial string operations, accounting for privilege-asserting code, and distinguishing the `SocketPermission` needed by `School` from the one needed by `Enterprise`. Complications arise with multithreaded programs that involve thousands of classes, partitioned in numerous security domains, and forming long sequences of method invocations. The solution presented in this paper addresses these concerns.

3 Concrete Semantics of Stack Inspection

For performance and scalability, Java and the CLR have adopted a *lazy semantics* for stack inspection; security information is not passed across method calls, but retrieved on demand at authorization checkpoints [6]. It has been proved that this lazy semantics is equivalent to an *eager semantics* in which security information is passed across method calls, ready to be used when needed at authorization checkpoints [28]. This section presents an eager semantics for stack inspection that, unlike previous work [19], models all the variants—including multithreaded code—and accounts for the differences between Java and the CLR. Section 4 describes our abstract semantics for stack inspection.

In Java and the CLR, a permission can guard multiple resources. Therefore, permissions are complex structures and carry an implication ordering. For example, the resource set guarded by `java.io.SocketPermission "*:80", "connect, resolve"` is a superset of the one guarded by `java.io.SocketPermission "ibm.com", "connect"`. In this paper, we use a simple concrete representation: we consider the universe $\mathcal{P}$ of *atomic permissions*, each guarding an individual resource.

Given a program p with sets of classes $\mathcal{C}$ and methods $\mathcal{M}$, an *access-control policy* for p is a function $\pi : \mathcal{M} \to 2^{\mathcal{P}}$. A policy π grants every method m a set of permissions, $\pi(m)$; if $\pi(m) = \varnothing$, then m is totally untrusted. Typically, in Java and the CLR, permissions are not granted directly to methods, but, with more coarse-grained granularity, to classes; a class induces its permissions on its methods.

Figure 2 defines an instrumented concrete eager semantics of a program assuming stack-inspection-based access control. We consider a standard concrete semantics for the program in the underlying language, where the program state consists of a program counter, stack, heap, local variables, and global variables. We instrument the program state additionally with a stack w of dynamically held permissions, with the convention that the stack grows from right to left. The stack alphabet is $2^{\mathcal{P}}$; each $\sigma \in 2^{\mathcal{P}}$ represents the set of permissions that an execution may hold at a particular point. We augment the program state with the set T of currently instantiated `Thread` objects, and a function $\alpha : T \to 2^{\mathcal{P}}$ that maps each `Thread` in T to its security context. If S is the program configuration under the standard concrete semantics, then $\langle S, w, T, \alpha\rangle$ is the program configuration under the instrumented concrete semantics.

When the main method m' of the program is invoked, the set of `Thread` instances instantiated so far is the singleton $T_0 = \{t_0\}$, where t_0 is the `Thread` instance created by the program launcher and representing the thread of execution of m'. α_0 is the function mapping `Thread` instances to their security contexts, with $\alpha_0(t_0) = \mathcal{P}$, representing the fact that the main thread, not having a parent thread, is only going to be constrained by the methods that will appear on its stack when an authorization check is performed.

The instrumentation for an execution x is defined as follows. Given a configuration $\langle S, w, T, \alpha\rangle$, we denote a transition of the instrumented concrete semantics into a configuration $\langle S', w', T', \alpha'\rangle$ by $\langle S, w, T, \alpha\rangle \Rrightarrow \langle S', w', T', \alpha'\rangle$, assuming that S' is the updated configuration according to the standard concrete semantics applied to S. Since the only operations that affect the instrumentation are method calls and returns, we only describe the effects of these operations. Such effects change the instrumentation based on which methods are involved in the invocation.

In general, when a method m invokes another method

- **Initialization—** Call to main method $m' \in \mathcal{M}$: $\langle S, \epsilon, \{t_0\}, \alpha_0\rangle \Rrightarrow \langle S', \pi(m')\epsilon, \{t_0\}, \alpha_0\rangle$
- **Method Call—** $m \in \mathcal{M}$ calls $m' \in \mathcal{M}$:

$$\langle S, \sigma w, T, \alpha\rangle \Rrightarrow \begin{cases} \langle S', (\pi(m) \cap Q)\sigma w, T, \alpha\rangle, & m' \text{ is the privilege-asserting API with parameter } Q \subseteq \mathcal{P} \\ \langle S', (\sigma \cap \pi(m'))\sigma w, T \cup \{t\}, [\alpha | t \mapsto \sigma \cap \pi(m')]\rangle, & m' = \texttt{<init>}\text{, creating } \texttt{Thread} \text{ instance } t \\ \langle S', (\pi(m') \cap \alpha(t))\sigma w, T, \alpha\rangle, & m = \texttt{start} \wedge m' = \texttt{run} \wedge m, m' \text{ have } \texttt{Thread} \text{ receiver } t \\ \langle S', \sigma\sigma w, T, \alpha\rangle, & m' \text{ is } \texttt{checkPermission/Demand} \text{ with parameter } p \in \sigma \\ \texttt{ERROR}, & m' \text{ is } \texttt{checkPermission/Demand} \text{ with parameter } p \notin \sigma \\ \langle S', (\sigma \cap \pi(m'))\sigma w, T, \alpha\rangle, & \text{otherwise} \end{cases}$$

- **Method Return—** $\langle S, \sigma w, T, \alpha\rangle \Rrightarrow \langle S', w, T, \alpha\rangle$

Figure 2. Stack-Inspection Eager Concrete Semantics

m', the set of permissions σ held at the top of the instrumentation stack w is intersected with $\pi(m')$, and this new set of permissions is pushed onto the top of the stack. The top of w is the authorization token examined by `checkPermission` in Java and `Demand` in the CLR.

If stack inspection traversed only the stack of the current thread, an active attacker could spawn a child thread—which would have fewer stack frames, and potentially more permissions, than its parent—and let it access a restricted resource. To prevent such attacks, Java and the CLR force the stack walk to traverse not only the stack of the child thread, but that of its parent as well. We model this behavior as follows. If m' is the constructor of a new `Thread`, then the new `Thread` instance, t, is added to T, and function α is augmented to map t to the current security context. When `start` is called on t, that causes a call to `run`. At that point, the authorization token is intersected with $\alpha(t)$.

If m' is the privilege-asserting API, with a set of permissions Q, then what gets pushed on the top of the stack is $\pi(m) \cap Q$. Therefore, all the permission sets that were intersected to compute σ up to that point are stripped away, except for $\pi(m)$.

In Java, when a `Permission` object q is demanded, the stack inspection is stopped at the stack frame preceding `doPrivileged` *indiscriminately*, as long as $q \in \pi(m)$. Conversely, the CLR's `Assert` can be parameterized with a set Q of `IPermission` objects, and a stack inspection for q is stopped only if $q \in \pi(m) \cap Q$. Our unified treatment of Java and the CLR assumes $Q = \mathcal{P}$ in the Java case.

The static analysis presented in this paper computes a policy for a program p that is *sufficient* for the program to run without entering the `ERROR` state. Furthermore, as we demonstrate empirically in Section 6, the policy we compute has a very small number of false positives—where a false positive, in this case, is a superfluous permission, and therefore, a PLP violation.

4 Static-Analysis Algorithm

A simple static permission analysis can be implemented by representing the execution of the program as a callgraph, propagating sets of permissions backwards in the callgraph starting from authorization checkpoints, and performing set unions at merge points [13, 18]. As we will observe in Section 7, this approach is too conservative, unless aggressive, but unscalable, context-sensitivity is adopted. This section describes a modular static analysis algorithm that combines a context-sensitive library analysis and a context-insensitive library-client analysis based on it. The library-client analysis is composed of (1) a string analysis that, for every program variable of type `String`, generates a Context-Free Language (CFL) representing possible values assigned to that variable, and (2) a program slicer [10] that tracks interprocedural dataflows.

4.1 Modular Library Analysis

In component-based systems, access control is centralized. For example, in Java, all the security-sensitive functions trigger, directly or indirectly, a call to the `checkPermission` method on the instance of `SecurityManager` currently active on the system, passing it a `Permission` object. This function calls `AccessController.checkPermission`, which performs the stack inspection. A context-insensitive callgraph represents all calls to the same method as one node; only one node models all the calls to the `checkPermission` methods of all the `SecurityManager` instances, and only one node models all the `AccessController.checkPermission` calls, irrespective of which `Permission` parameter those methods are passed.

The components of the program of Section 2 collectively require five permissions, as shown in Table 1. Modeling stack inspection as a simple backward dataflow problem in a context-insensitive callgraph would not effectively disambiguate the different propagation paths of those permission requirements. Furthermore, it would not be possible to distinguish which permission is shielded by the `doPrivileged` call in `createSocket`. A security policy based on the results of this analysis would conservatively fill every cell of Table 1 with a checkmark. Because of lack of scalability, a global context-sensitive analysis [13] offers no solution for large programs.

The analysis presented in this paper improves on precision and scalability via the construction of library summaries, which eliminate the need for reanalyzing a library at every library call. Once a summary has been built, the permission analysis can start modeling stack inspections directly at the library entrypoints instead of starting from the access-control enforcer. This reduces any imprecision due to overlapping callgraph library paths.

To construct library entrypoint summaries, the library l itself is analyzed context-sensitively as an incomplete program, considering all its public and protected methods as possible entrypoints. Let $G = (N, E)$ be a callgraph representing the set of all possible executions of l with an arbitrary client, and N_1 and N_2 the subsets of N corresponding to the `checkPermission`/`Demand` and `doPrivileged`/`Assert` APIs, respectively. At run time, each permission in $\mathcal{P}$ is represented as an object of type `Permission` in Java and `IPermission` in the CLR. Statically, a permission can be represented as the allocation site of the corresponding `Permission` or `IPermission` object. Two permissions whose objects share the same allocation site can be considered equivalent. This equivalence relation partitions $\mathcal{P}$ into a finite set $\mathcal{P}'$ of permission allocation sites.

Stack inspection can be statically modeled as a *two-phase backward dataflow problem.* Phase 1 is a standard backward dataflow propagation, initialized as follows. We define $\text{Gen}(n) := \varnothing, \forall n \in N \setminus N_1$. If $n \in N_1$, let $Q \subseteq \mathcal{P}'$ be the set of `Permission`/`IPermission` allocation sites that, in the model, can flow to the parameter of `checkPermission`/`Demand`; we impose $\text{Gen}(n) := Q$. Furthermore, we define $\text{Kill}(n) := \varnothing, \forall n \in N \setminus N_2$. If $n \in N_2$, let $Q \subseteq \mathcal{P}'$ be the set of `Permission`/`IPermission` allocation sites that, in the model, can flow to the parameter of `doPrivileged`/`Assert`; we impose $\text{Kill}(n) := Q$. Notice that `doPrivileged` is not parameterized based on permissions [18]. Thus, in Java, $\text{Kill}(n) = \mathcal{P}', \forall n \in N_2$. The dataflow equations are as follows:

$$\text{Out}(n) := (\text{In}(n) \setminus \text{Kill}(n)) \cup \text{Gen}(n) \qquad (1)$$

$$\text{In}(n) := \bigcup_{m \in \Gamma^+(n)} \text{Out}(m) \qquad (2)$$

for all $n \in N$, where $\Gamma^+ : N \to 2^N$ is the *successor function* in G, defined by $\Gamma^+(n) := \{n' \in N \mid (n, n') \in E\}$.[2] By Tarski's Theorem [7], the recursive computation of the solutions of Equations (1) and (2) converges to a fixed point in $\mathcal{O}(|E||\mathcal{P}'|)$ time , given that the height $\mathcal{H}(2^{\mathcal{P}'})$ of the lattice $2^{\mathcal{P}'}$ is $|\mathcal{P}'|$, and to reach a fixed point, each edge of G can be traversed at most $\mathcal{H}(2^{\mathcal{P}'})$ times.

Phase 1 does not propagate permissions beyond the privilege-asserting API. However, according to the stack-inspection semantics, the code calling the privilege-asserting API needs to be granted the shielded permissions, as noted in Sections 2 and 3. Phase 2 models this through a one-step, non-recursive backward propagation, as described by Equation (3), to be solved upon completion of Phase 1.

$$\text{In}(n) := \text{In}(n) \cup \bigcup_{n_2 \in \Gamma^+(n) \cap N_2} \text{In}(n_2) \qquad (3)$$

for all $n \in N$. Equation (3) has a worst-case time complexity of $\mathcal{O}(|E|)$ since each edge is traversed at most once.

To model multithreaded programs, we need to connect the stack of a thread with that of its parent as described in Section 3. This can be done by *thread-augmenting G* as follows. Given a node $r \in N$ representing the invocation of the `run` method on a set of `Thread` receivers, we identify all the `Thread` constructor nodes $t_1, \ldots, t_k \in N$ where those receivers were constructed, and add edges $(t_1, r), \ldots, (t_k, r)$ to E. Equations (1), (2), and (3) take care of properly annotating the nodes in a parent thread with the permissions required by its children.

To detect what permissions are needed by each method, we define a function $\pi' : \mathcal{M} \to 2^{\mathcal{P}'}$ as follows: for every $m \in \mathcal{M}$, if m is represented by nodes $n_1, n_2, \ldots, n_k \in N$, then $\pi'(m) := \bigcup_{i=1}^{k} \text{In}(n_i)$.[3] The static analysis presented in this section *induces* an access-control policy $\tilde{\pi} : \mathcal{M} \to 2^{\mathcal{P}}$ on l defined by $\tilde{\pi}(m) := \bigcup_{Q \in \pi'(m)} Q, \forall m \in \mathcal{M}$.

Our library analysis is sound, as can be easily proved by a straightforward induction on the structure of G.

4.2 String Analysis

It is very common for a fully-qualified file name to be specified as something like `dir + File.separator + name.substring(1)`. A permission analyzer that does not perform string analysis will have to conservatively approximate the permission to read such a file as the permission to read *all the files* of the file system—an overapproximation that may cause PLP violations. Our library-client analysis, which will be presented in Section 4.3, is the first one to use string analysis to resolve string values used in permission requirements.

[2] $\text{In}(n)$ and $\text{Out}(n)$ are defined at the exit and entry of n, respectively.

[3] If m is not represented by any node in the callgraph (m is unreachable), then $k = 0$ and $\pi'(m) = \varnothing$. If m is reachable, then $k \geq 1$ since G is context-sensitive.

String analysis is a family of static program analyses that approximate the possible string values of the program variables arising at run time. Our string analysis algorithm is based on the one proposed by Minamide [15]. It produces a Context-Free Grammar (CFG) that represents the possible string values assigned to program variables. The CFG is deduced by solving the subset constraints among the sets of strings assigned to program variables. For every predefined string operation that appears in a program, we automatically generate a sound approximation of the transformation that maps the CFG representing the possible input strings to the CFG of the possible output strings. *Sound* here means that the resulting CFG contains all the actual strings arising at run time. For example, `toLowerCase()` in Figure 1 can be approximated by a homomorphism, which is a structure-preserving mapping between CFGs. Similarly, `substring(2)` in Figure 4 can be represented by a finite automaton with output, or *transducer*, which induces a stateful transformation, as shown in Figure 3. In Figure 3, the transitions labeled with A/ϵ indicate that the transducer will produce ϵ for the first two input characters read. The CFG approximating the output of the Java program obtained by considering assignments as production rules is then transformed into another CFG via the transducer, as in Figure 4 (the image of a CFG under a transducer is a CFG).

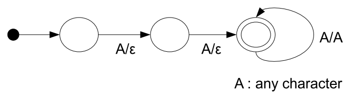

Figure 3. The Transducer for `substring(2)`

A cheaper, yet sound, string analysis based on constant propagation would only give us the constant values in the program propagated through the string operations, and would overapproximate every non-constant value to $\star$, which is too conservative.

Our string analysis has a novel labeling feature that records how each string is constructed. Every label is determined based on the program locations corresponding to string creations and manipulations. Labeling starts by associating labels with every character and each approximated string operation while translating a program into production rules. Each labeled approximated string operation propagates its own label as well as the labels of the input characters to the characters of the resulting CFG. Label propagation is performed by calculating label-set unions.

Figure 4 shows a sample program and the production rules translated from it. Superscript numbers are program locations indicating where the string characters are created and manipulated. Label 1 on characters x,y, and z indicates that x,y, and z are created on program location 1. Label 2 on character a indicates that a is created on program location 2. The approximated function <u>`concat`</u> has label 3 to

```
String a = "xyz";
for (int i = 0; i < 3; i++) a = a + "a";
String r = a.substring(2);
```

$$\begin{aligned} S_a &\rightarrow \mathtt{x}^{\{1\}}\ \mathtt{y}^{\{1\}}\ \mathtt{z}^{\{1\}} \\ S_a &\rightarrow \underline{\mathtt{concat}}^3(S_a, \mathtt{a}^{\{2\}}) \\ S_r &\rightarrow \underline{\mathtt{substring}}^4(S_a, 2) \end{aligned}$$

Figure 4. Program and Production Rules

indicate that concatenation of strings is performed on program location 3. Label 4 represents a program location on which `substring` is applied. The labeled approximated function $\underline{\mathtt{concat}}^3$ concatenates two CFGs and propagates label 3 to every character. Likewise, $\underline{\mathtt{substring}}^4$ propagates label 4. By applying the labeled approximated functions, we obtain the following CFG: $S'_a \rightarrow \mathtt{z}^{\{1,4\}}, S'_a \rightarrow S'_a\ \mathtt{a}^{\{2,3,4\}}, S_r \rightarrow S'_a$. This CFG represents the set of strings $\{\mathtt{z}^{\{1,4\}}, \mathtt{za}^{\{1,2,3,4\}}, \mathtt{zaa}^{\{1,2,3,4\}}, \ldots\}$. We conclude that z is constructed only through locations 1 and 4, while the other strings are constructed through all the locations. More formally:

Definition 4.1 (String Analysis) *A* string analysis S *for a call graph* $G = (N, E)$ *is a tuple* (S_v, S_o, S_s, S_c) *where* S_v, S_o *are two sets and* S_s, S_c *are two functions such that:*

- $S_v := \{$*primitive string components in* $G\}$
- $S_o := \{$*primitive string operations in* $G\}$
- $S_s(\mathtt{w}) := \{$*CFG estimate of strings held by* $w\}$
- $S_c(x) := \{y \in S_o \cup S_v\ |$*x is labeled with* $y\}$

For each disjunct x, $S_c(x)$ returns the constituent components that *may* have been used in the computation of the value of x. These constituents have two forms: S_v is the set of manifest string constants and string input parameters in G, and S_o is the set of result values of all primitive string operations in G. Thus, $S_c(x)$ denotes all primitive string components and all applied string operations that together gave rise to x. Therefore, the universe of labels is $S_v \cup S_o$, and S_c is the labeling function of our string analysis.

4.3 Library-Client Analysis

Let $G = (N, E)$ be a thread-augmented callgraph representing the execution of a program p including library l. A summary for l constructed as described in Section 4.1 identifies a subset $N_3 \subseteq N$ of nodes that overapproximates the set of *security-sensitive entrypoints* of l (methods that, when invoked, trigger a stack inspection), and since the library analysis is sound, for each entrypoint m, $\tilde{\pi}(m)$ is a superset of the set of permissions required to invoke m.

If a program p includes l, a backward dataflow permission analysis for p can be initialized at the nodes in N_3 as opposed to the ones in N_1. As observed in Section 4.1, this will avoid the false positives due to callgraph paths overlapping on one or more l nodes. However, summaries of l cannot avoid the false positives generated by paths overlapping on nodes outside of l—in the callgraph representing the executions of clients of l. In the example of Section 2, summarizing the permission requirements of the constructors of `Socket` and `FileOutputStream` would be sufficient to identify that the only permission shielded by the call to `doPrivileged` is `FilePermission "C:/log.txt", "write"`, but that will not disambiguate the four `SocketPermission` requirements for `connectToEnt` and `connectToSchool`. The analysis would conservatively tag every node in both paths as requiring all four permissions. This section presents a scalable algorithm combining callgraph analysis, pointer analysis, program slicing, and string analysis.

We partition N_3 into subsets $N_{3,1}, N_{3,2}, N_{3,3}$. Nodes in $N_{3,1}$ represent methods that require "constant permissions". A *constant permission* is either a permission with no parameters, such as `AllPermission`, or a permission with string-constant parameters where the constants are defined by the access-control enforcer and do not depend on the client. A callgraph node representing a call to `ClassLoader.<init>` is in $N_{3,1}$ because the permission requirement it generates is `RuntimePermission "createClassLoader"`, and `createClassLoader` is a string constant defined by the Java runtime. A node in $N_{3,2}$ represents a *string-parameterized entrypoint*, whose required permissions depend on one or more `String` parameters passed by the client. A callgraph node representing a call to `Socket.<init>` is in $N_{3,2}$ because its `String` parameter (for example, `ibm.com`) flows directly to a parameter of the required permission (as in `SocketPermission "ibm.com", "resolve"`). Finally, a node in $N_{3,3}$ represents a *non-string-parameterized entrypoint*, whose required permissions depend on one or more non-`String` parameters passed by the client. For example, any node representing a call to the `FileOutputStream` constructor with a `File` parameter is in $N_{3,3}$. That `File` parameter wraps the filename `String` object that becomes the target parameter in the permission requirement, as in `FilePermission "C:/log.txt", "write"`. Our static permission analysis for client code treats these three cases differently.

Permission Requirements from Nodes in $N_{3,1}$ can be computed via a simple backwards dataflow problem as in Equations (1), (2), and (3).

Permission Requirements from Nodes in $N_{3,2}$ are more complicated to model since they require distinguishing the propagation of permission requirements that differ by their parameters. Figure 5 shows the callgraph of a program call-

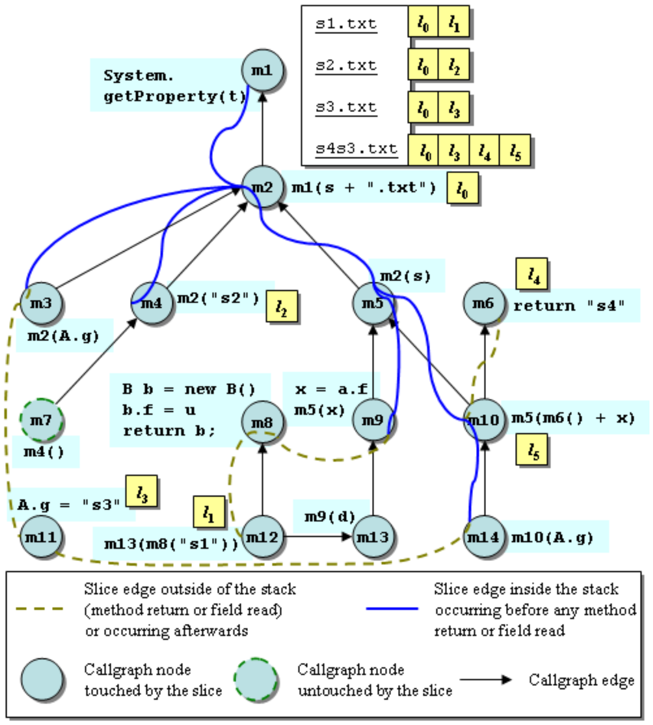

Figure 5. Library-Client-Analysis Scenario

ing `System.getProperty`, a string-parameterized entrypoint.[4] Informally, the analysis proceeds as follows:

1. The string analysis computes the CFL of the possible values for the `String` parameters to the security-sensitive call. The possible `String` values computed for the parameter to `getProperty` in Figure 5 are `s1.txt`, `s2.txt`, `s3.txt`, and `s4s3.txt`.

2. The string analysis maps each `String` parameter identified in Point 1. to its labels, as discussed in Section 4.2. For example, in Figure 5, the four `String` values listed in Point 1. are mapped to sets of labels $\{l_0, l_1\}$, $\{l_0, l_2\}$, $\{l_0, l_3\}$, and $\{l_0, l_3, l_4, l_5\}$, respectively.

3. Each of the values in the CFL is used to instantiate a permission requirement, such as `PropertyPermission "s1.txt", "read"` in Figure 5.[5]

4. An analysis that agglomerates all these permissions indiscriminately is unsatisfactory since it may lead to PLP violations. To identify the propagation stacks of each permission requirement, our analysis computes a backward slice rooted at the parameter-passing statement in each predecessor (`m1`) of the security-sensitive entrypoint (`getProperty`), and then follows the backward slice as

[4]Next to each node in Figure 5 is a box showing the relevant code in the corresponding method.

[5]If the security-sensitive method takes multiple `String` parameters, a permission will be instantiated for each element of the Cartesian product of the sets of the different parameter values.

this overlaps possible stacks; we get what we call *stack slices*, represented in Figure 5 with solid, curved edges.

5. Where the slice stops overlapping with any stack (by coming to an end, or going through a method return or field-read operation), the string analysis is queried to collect the set of labels that may participate in the definition of the resulting string. Points of query in Figure 5 are `m3`, `m9`, `m14` (field reads), `m4` (slice end), and `m10` (method return).

6. This operation annotates some of the nodes of the stack slice with sets of labels; for example, `m2` $\rightarrow \{l_0\}$, `m4` $\rightarrow \{l_2\}$, `m3` $\rightarrow \{l_3\}$, `m2` $\rightarrow \{l_0\}$, `m9` $\rightarrow \{l_1\}$, `m10` $\rightarrow \{l_4, l_5\}$.

7. The sets of labels are then propagated in a backward dataflow problem through the stack slice, performing set unions at the merge points. The sets of labels obtained when the fixed point is reached will be used to distinguish the possible permission requirements at those entrypoints. For example, in Figure 5, client entrypoints `m3`, `m7`, `m12`, `m14` will be tagged as requiring the following sets of permissions, respectively:

- {`PropertyPermission "s3.txt", "read"`}
- {`PropertyPermission "s2.txt", "read"`}
- {`PropertyPermission "s1.txt", "read"`}
- {`PropertyPermission "s4s3.txt", "read", PropertyPermission "s3.txt", "read"`[6]}

8. Those requirements are then propagated forward in the stack slice and then backwards in the callgraph, performing set unions at the entrypoints.

9. Both the backward propagations of Points 7 and 8 above must stop at privilege-asserting API nodes. A one-step backward propagation as the one induced by Equation (3) is then required as discussed in Section 4.1.

We can describe this algorithm more formally:

Definition 4.2 (Stack Slice) *A stack slice $\Sigma(v, n)$ is a backward slice with respect to a local variable or parameter v and node n that follows only definitions within a given program stack, stopping at any other kind of definition, such as a read from the heap or a function return value. For our purposes, all values in the stack fall into four categories, and the stack slice is defined as follows in terms of them:*

- constant *c:* $\Sigma(c, n) := \{c\}$
- parameter *p:* $\Sigma(p, n) := \{p\} \cup \bigcup_{\eta(x) \in \Delta^-(n,p)} \Sigma(x, n)$
- primitive string operation $r = f(v_1, \ldots, v_n)$*:* $\Sigma(r, n) := \{r\} \cup \bigcup_{v_i} \Sigma(v_i, n)$
- other *v:* $\Sigma(v, n) := \{v\}$

where function η maps any local variable or parameter to its defining callgraph node, $\Delta^-(n, p) := \{\eta(x) \,|n \rightarrow^* \eta(x) \in E \wedge \eta(x) \rightarrow \eta(p) \in E\}$, and $\rightarrow$ denotes edges in G. We also define $\tau(v)$ to return which of the four categories a given v belongs to, for any v from a stack slice.

[6]This is a false positive for `m14`.

Our notion of permissions pertains to sensitive nodes. A *sensitive node s* has two properties: a *sensitive value v_s*, which holds permissions, and the set of permissions required, p_s. For simplicity, we assume that this set of permissions is determined by the set of strings reaching v_s.

We would like to define a safe approximation of the required permissions. To do so, we observe that only strings actually read onto or computed in a given stack can be passed up that stack to a sensitive operation. Thus, if we compute the components of all strings read onto the stack and all operations on the stack itself, we can filter any string at the sensitive operation that is composed in part of any other operation or component. Based on that observation, we take a stack slice $\Sigma(v_s, n)$ that covers all dataflow through stacks from n to the sensitive operation in s, and we compute (1) all the components of all the strings read into the slice, and (2) the operations on them. We then use those components to prune the full set of strings at the sensitive operation as determined by the overall string analysis. Hence, the set of string components needed for the stacks rooted at n for sensitive value v_s is defined by $\text{CS}(v_s, n) := \{c | \exists x \in \Sigma(v_s, n) \wedge c \in S_c(x) \wedge \tau(x) \in \{\text{constant}, \text{other}\}\}$. The components in $\text{CS}(v_s, n)$ are the *only ones* that can appear in strings passed to the sensitive operation. So we need permissions in the stack rooted at n only for strings that contain those components; we define this as $P(v_s, n) := \{s \,| s \in S_s(v_s) \wedge (S_c(s) \cap \text{CS}(v_s, n)) \neq \varnothing\}$. Given any $m \in \mathcal{M}$ represented by $n \in N$, the permissions needed by m at runtime are overapproximated by set $\tilde{\pi}(m) := \bigcup_{v_s | n \rightarrow^* \eta(v_s)} P(v_s, n)$. The soundness of this algorithm can be proved with a straightforward induction on the depth of the stack, based on the correctness of the stack slice.

Permission Requirements from Nodes in $N_{3,3}$. Non-string-parameterized permissions depend on non-`String` parameters passed by the client. Since permissions are characterized by string values, those non-`String` parameters act as *string containers* from which the string values are extracted at run time to form the permissions. For example, the call to the `FileOutputStream` constructor in the following code snippet is a node in $N_{3,3}$, and the `File` object passed to it acts as a string container.

```
File f = new File("C:", "log.txt");
FileOutputStream fos = new FileOutputStream(f);
```

To reconstruct the string in the permission requirement (which here is `FilePermission "C:/log.txt", "write"`), pointer analysis is used to locate the allocation of the `File` object, detect its `String` parameters `C:` and `log.txt`, and compute the String `C:/log.txt` that appears in the permission, according to predefined rules. If no rule is available for a specific string container, the conservative top value for the corresponding permission type, typically a wildcard ($\star$), is used. If one of the parameters is itself a string container, then the process is iterated until all the strings are fully evaluated. Once the strings are available, the analysis proceeds as in the $N_{3,2}$ case.

5 Implementation

A3 is based on top of IBM Research's Watson Libraries for Analysis (WALA) [27]. A novel contribution of this paper is the functional modularity of the analysis. A3 uses automatically generated summaries for all the standard Java libraries. Without such summaries, the analysis was shown not to scale to large applications. To disambiguate permission-propagation paths inside a library, A3 builds a 1-CFA context-sensitive library callgraph [8], but this may be too expensive even for a library. A preanalysis of the standard Java libraries has shown that their permission requirements are totally functional because the permissions required by a program when invoking an entrypoint method depend at most on the receiver and the parameters passed to it; there are no side effects due to interactions with the heap or other entrypoint calls. Therefore, partitioning the library entrypoints into smaller subsets, and repeatedly running the 1-CFA summary-construction analysis on the callgraphs generated based on those partitions is *sound*; no permission requirement will be lost. Empirically, we found that an effective way to partition the entrypoints of the standard libraries is by groups of 10 packages.

For client analysis, A3 builds a 0-1-CFA context-insensitive callgraph [8]. As discussed in Section 4.3, A3 makes heavy use of program slicing. The slicing algorithm of A3 is built on top of WALA and has the following characteristics: (1) it tracks data dependencies, (2) it *safely* ignores control dependencies (which are irrelevant for permissions), and (3) it is context-insensitive [25].

6 Experimental Results

This section summarizes the experimental results obtained by executing A3 on five applications from SourceForge [24], listed in Table 2. The results are compared with those obtained with SWORD4J [26] (the SWORD4J algorithm is discussed in Section 7). All the public methods of those five applications were considered as entrypoints for the analyses. The results were obtained on a Lenovo T61P ThinkPad with an Intel T7700 Core Duo 2.40 GHz processor, 3 GB of Random Access Memory (RAM), and Microsoft Windows XP SP2 operating system. Both analyzers were run on a Sun Microsystems Java Standard Edition (SE) V1.4.2_05 Runtime Environment.

For each application, Table 2 compares the number of nodes in the generated callgraphs and the times taken to perform the two analyses, and classifies the permissions detected. Among these permissions, we identify the false positives, and distinguish those due to Wrong Method Identification (WMI) from those due to Wrong Permission Identification (WPI). A WMI comes from the analysis conservatively positioning the method on a stack requiring permissions when that method is not security-sensitive at run time. A WPI comes from the analysis conservatively identifying an unnecessary permission class on a particular stack. Once a false positive is computed for a method, the analysis automatically propagates it to all the predecessors of that method. However, we report each false positive only once at its first occurrence in the application code.

For A3, we pick out those permissions that are *instantiated* starting from the library entrypoints, based on the specific parameters passed by the client code. Computing the values of these permissions requires string analysis. Thus, among them, we do not count those that depend on constant strings, which both A3 and SWORD4J can detect. Since SWORD4J does not model string operations, each permission instantiated by A3 is overapproximated by SWORD4J with $\star$. For example, `JPTAPI` constructs a `FileReader` object with a `File` parameter instantiated with `String` value `C:/test.txt`. The corresponding `FileReader` constructor node is in $N_{3,3}$, and the `File` parameter is a string container. Following the algorithm described in Section 4.3, A3 identifies the precise `FilePermission "C:/test.txt", "read"` requirement, whereas SWORD4J reports a conservative `FilePermission "*", "read"`—an approximation that can easily lead to PLP violations.

By comparing the number of non-instantiated permissions detected by the two analyzers, A3 has a false-positive rate of only 14%, as opposed to 61% of SWORD4J—*a precision improvement of* 77%. For the instantiated permissions, A3 identifies on average 21% false positives due to the conservativeness of the analysis. By our definition, these false positives are not WPIs because the permission class is correctly identified. Despite these false positives, the instantiated permissions reported by A3 are more precise and useful in 53% of the cases than the corresponding results reported by SWORD4J. For example, assume that A3 overapproximates the filename in a `FilePermission` with set of strings {`f1`, `f2`}, in which `f1` is an actual requirement and `f2` a false positive, against a $\star$ overapproximation by SWORD4J. It is easier for a system administrator to extract a minimal policy from {`f1`, `f2`} than from $\star$. Furthermore, even if the unnecessary `f2` permission were mistakenly granted, that would be less of a security exposure than granting the $\star$ permission.

Typically, the false negatives of a static analysis for Java arise from not modeling native methods. However, native methods are not involved in authorization requirements be-

Application	SWORD4J						A3					
	Nodes	Time	Permissions				Nodes	Time	Permissions			
		sec.	WMI	WPI	Detected	Inst.		sec.	WMI	WPI	Detected	Inst.
`JPDStore`	28524	117	0	14	18	-	482	8	0	1	13	8
`JPTApi`	26871	109	0	13	14	-	175	5	0	0	2	2
`Java Integrity`	27427	70	1	1	14	-	257	10	0	1	10	7
`JavaCup`	32893	128	0	1	5	-	636	13	0	0	7	5
`Ganymede`	35820	93	0	16	22	-	1167	19	0	4	11	4

Table 2. Experimental Results of A3 Analysis

cause they are executed outside of the Java Virtual Machine (JVM), so we expect neither A3 nor SWORD4J to exhibit any false negatives due to native methods.[7] However, for scalability reasons, SWORD4J unsoundly excludes from the analysis scope several packages, such as `java.awt` and `javax.swing`. No permissions required by those packages and their callers will be reported by SWORD4J. Consequently, SWORD4J may have false negatives. We detected four SWORD4J false negatives on `JavaCup`. Conversely, A3 does not exclude any libraries; it soundly models the execution of any application and we did not detect any false negatives.

Besides the precision gains, our analysis offers benefits in terms of running time and scalability, due to the modularity and context-insensitivity of A3. In terms of running time (excluding the time required to compute the summaries, which are then used by all the analyses) *A3 outperforms SWORD4J by an average factor of 12*. By using the callgraph size to measure the scalability of the analysis, we notice that *the callgraphs generated by A3 are smaller than those generated by SWORD4J by an average factor of 80.*

7 Related Work

The work related to this paper covers several areas: security analysis, modular static analysis, and string analysis.

In the area of security analysis, Wallach and Felten present a formalization of stack inspection that examines authorization based on the principals currently active in a thread stack at run time (*security state*) [28]. An optimization technique, called *Security-Passing Style* (SPS), encodes the security state of an application while the application is executing. Each method is modified to pass a security token as part of each invocation. The token represents an encoding of the security state at each stack frame, as well as the result of any authorization test encountered. Pottier *et al.* [19] extend and formalize the SPS via type theory using a λ-calculus, but do not handle incomplete programs. Jensen *et al.* [11] focus on proving that code is secure with respect to a global security policy. Their model employs operational semantics, using a two-level temporal logic, and shows how to detect redundant authorization tests. They assume that the whole program is available for analysis. Bartoletti *et al.* [1] are interested in optimizing performance of run-time authorization tests by eliminating redundant tests and relocating others. The reported results apply operational semantics to model the run-time stack. Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [4] describe a system that inlines reference monitors into the code to enforce specific security policies. The objective is to define a security policy and then inject authorization points into the code to reduce redundant tests. Conversely, this paper studies authorization from the perspective of an existing system containing authorization test points. Koved *et al.* [13] and Pistoia *et al.* [18] automate static security analysis for Java authorization and privilege assertion, respectively, in the SWORD4J tool [26]. Stack inspection is modeled as a backward dataflow problem on a context-sensitive callgraph, which often does not effectively disambiguate permission flows. Since an expensive context-sensitivity is applied indiscriminately, scalability is affected too. Conversely, A3 builds a relatively inexpensive 0-1-CFA callgraph [8, 23], and adds precision only where needed using more expensive analyses. Also, A3 summarizes libraries and uses string analysis to enhance precision. Pistoia *et al.* [17] identify flaws in stack inspection, which arise when code no longer on the stack is still capable of influencing a security-sensitive operation.

The modular analyses most closely related to our work are those which compute dataflow summaries for library methods. Rountev *et al.* describe a general approach for static analysis of program fragments [21]. The interactions of a program fragment with the rest of the program are modeled by summary values and functions. This formal analysis model is later instantiated in a points-to and side-effect analysis for C libraries and their clients [20]. If we consider the set of Java libraries in a benchmark to be the program fragment, then our modular permissions analysis of the libraries is a specific simple instance of this analysis model. Note that the permission dataflow solution for the Java Runtime is independent of the library client. Flanagan and Felleisen [5] present a general approach for componential set-based analysis of functional languages. The static analysis represents

[7] The only native methods that are important in authorization are `doPrivileged` and `Thread.start` (discussed in Section 3), and these methods are synthetically modeled by A3.

program properties by sets of constraints (set-based analysis). The main idea is to derive a separate simplified constraint system on each module for the dataflow problem being solved. These constraints are later combined to obtain the solution for the entire program. In some sense, the individual constraint files are analogous to the library summaries in our algorithm. Meunier *et al.* apply the ideas in the previous paper to PLT Scheme, in which programs consist of modules with contracts, that describe function inputs and outputs using predicates [14]. Zhang and Ryder describe a modular reachability analysis that summarizes the possible callbacks from a library associated with each of its public entries [31].

A Java String Analyzer (JSA) was first introduced by Christensen *et al.* [2, 12]: possible strings arising at run time are approximated by a regular language for statically checking errors in dynamically generated Structured Query Language (SQL) queries. Wassermann and Su [29] extend Minamide's algorithm [15] (discussed in Section 4.2) to syntactically isolate tainted substrings from untainted substrings. They label non-terminals in a CFG with annotations reflecting taintedness and untaintedness. Fang Yu et al. [30] propose a string-analysis algorithm for PHP that is based on finite automata. Their algorithm computes an automaton to model every string value in the program, whereas ours builds a CFG.

8 Conclusion

This paper has presented a novel modular static analysis for identification of permission requirements for stack-inspection systems. Furthermore, since strings are essential when defining permissions, string analysis is used to enhance precision, and a combination of string analysis and program slicing allows for disambiguating permission-propagation paths. The analysis has been implemented in a tool for automatic identification of Java permission requirements called A3. The effectiveness of A3 on public benchmarks shows outstanding improvement over previous work. In the future, we would like to extend this work to model more complex authorization systems, such as Information-Based Access Control (IBAC) [17].

References

[1] M. Bartoletti, P. Degano, and G. L. Ferrari. Static Analysis for Stack Inspection. In *ConCoord 2001*.

[2] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *SAS 2003*.

[3] Equinox Project, `http://www.eclipse.org`.

[4] U. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *S&P 2000*.

[5] C. Flanagan and M. Felleisen. Componential Set-based Analysis. *TOPLAS*, 21(2), 1999.

[6] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USITS 1997*.

[7] G. Grätzer. *General Lattice Theory*. Birkhäuser, 2nd ed., 2003.

[8] D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *TOPLAS*, 23(6), 2001.

[9] N. Hardy. The Confused Deputy (Or Why Capabilities Might Have Been Invented). *OSR*, 22(4), 1988.

[10] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI 1988*.

[11] T. P. Jensen, D. L. Métayer, and T. Thorn. Verification of Control Flow Based Security Properties. In S&P 1999.

[12] Java String Analyzer, `http://www.brics.dk/JSA/`.

[13] L. Koved, M. Pistoia, and A. Kershenbaum. Access Rights Analysis for Java. In *OOPSLA 2002*.

[14] P. Meunier, R. B. Findler, and M. Felleisen. Modular Set-based Analysis from Contracts. In *POPL 2006*.

[15] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW 2005*.

[16] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification. In *CSFW 2004*.

[17] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond Stack Inspection: A Unified Access Control and Information Flow Security Model. In *S&P 2007*.

[18] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *ECOOP 2005*.

[19] F. Pottier, C. Skalka, and S. F. Smith. A Systematic Approach to Static Access Control. In *ESOP 2001*.

[20] A. Rountev and B. G. Ryder. Points-to analysis and side-effect analysis for programs built with precompiled library modules. In *CC 2001*.

[21] A. Rountev, B. G. Ryder, and W. Landi. Data-Flow Analysis of Program Fragments. In *FSE 1999*.

[22] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, 63, 1975.

[23] O. Shivers. Control Flow Analysis in Scheme. In *PLDI 1998*.

[24] SourceForge.net, `http://www.sourceforge.net`.

[25] M. Sridharan, S. J. Fink, and R. Bodík. Thin Slicing. In *PLDI 2007*.

[26] IBM Java Security Workbench Development for Java (SWORD4J), `http://www.alphaworks.ibm.com/tech/sword4j`.

[27] T. J. Watson Libraries for Analysis (WALA), `http://wala.sourceforge.net`.

[28] D. S. Wallach and E. W. Felten. Understanding Java Stack Inspection. In *S&P 1998*.

[29] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *PLDI 2007*.

[30] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic String Verification: An Automata-Based Approach. In *SPIN 2008*.

[31] W. Zhang and B. G. Ryder. Automatic Construction of Accurate Application Call Graph with Library Call Abstraction. *Journal of Software Maintenance and Evolution*, 19(4), 2007.