Web Security Analysis for Java Using Conditional Weighted Pushdown System

By HUA, Vy Le Thanh

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Mizuhito Ogawa

September, 2012

Web Security Analysis for Java Using Conditional Weighted Pushdown System

By HUA, Vy Le Thanh (1010225)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Mizuhito Ogawa

and approved by
Professor Mizuhito Ogawa
Associate Professor Xavier Defago
Associate Professor Kazuhiro Ogata

August, 2012 (Submitted)

Acknowledgements

Admittedly, this thesis would not have been possible without Research Assistant Professor Xin Li and Professor Mizuhito Ogawa. I sincerely appreciate their valuable advice and consistent encouragement since the early stages of my study. In particular, I am deeply indebted to Xin Li for broadening my horizons and supervising major parts of my thesis. Her dedication in teaching and research has always been a rich source of inspiration. I would like to thank Associate Professor Yasuhiko Minamide at University of Tsukuba for his fruitful discussion on evaluation and performance of our approach. Definitely, it is my honor to have this thesis reviewed by Associate Professor Xavier Defago and Associate Professor Kazuhiro Ogata. I have also greatly profited from Associate Professor Nao Hirokawa's comments on improving the presentation. My gratitude is furthermore sent to all colleagues at Ogawa Lab for their kindness to help me get into new environment seamlessly. In addition to all the foregoing, I am much grateful to my beloved family and friends for keeping their faith in me throughout my time at Master course.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This introduction comprises four parts. The first part paints in broad strokes the big picture of Java Web security and unveils motivation of our study. The second part brings out an overview of the approach proposed to work out a challenge in the field. The third part provides a concise view of our contributions; while that last one shows the outline of this thesis.

1.1 The Motivating Big Picture

In light of recent development in cybertechnologies and automation, our critical systems increasingly relies on computers and networks. In spite of the vast benefits automated services bring to daily life, the pervasiveness of computer and network usage gradually exposes our critical infrastructures to a wide variety of threats and malicious attacks. Accidental failures and disruption yielded by such unintended events can trigger disastrous effects in financial and reliable aspects of a service. According to Symantec report published in April 2012 [12], in 2011 there are about 4 595 Web attacks blocked per day and over one million identities exposed per security breach. These dire consequences have highlighted the substance of Web security.

Notwithstanding numerous technologies introduced lately, Java is still one of the best vehicles to program Web applications. Analogous to such tremendous rate of adoption is a high expectation of a safe Web programming platform. Thereupon, building more trust to Java Web applications is an inherent mission not only to software developers but also to researchers in related fields.

In the overall security jigsaw puzzle from cryptography to firewalls, from antivirus software to privacy protocols, access control is a crucial piece. Access control mechanisms are born to be the fundamental protection for systems with shared resources. There are three main issues in an access control mechanism, each of which corresponding to a particular security question:

(i) *Authentication* – Who is a principal P in the system? Assume that attributes of P are known.

(ii) *Authorization* – When a principal P makes a request, should it be allowed?

(iii) *Enforcement* – How to implement the authorization during an execution?

Permission grant is a tough work, since it requires heavy domain-specific knowledge and continuous refinement to adapt to real-world situations. Too restrictive policy may cause authorization failures at runtime, whereas granting too many permissions may open a security hole. The burden of authorization assignment can be moderated if some parts of the work is automated. Herein, our study contributes an automation of authorization test which would provide at compile-time the test result of a given security policy.

In software development terms, detecting vulnerable components by testing can harm the underlying system. Furthermore, when the system grows in complexity, it is non-trivial to determine correct behaviors of a program at runtime. Static analysis techniques are therefore taken to settle the problem. These techniques offer a way to safely approximate the set of program behaviors arising dynamically at runtime. A large volume of literatures has been published with aim at access control analysis for Java. Nevertheless, there remains roadblock to a precise analysis when context of permission requirements are under consideration. This challenge is the critical issue we manage to lift while constructing our permission analysis framework.

1.2 Overview of Permission Analysis Framework

The ultimate target of our framework is to *check whether access control against a given trusted domain always succeed or may fail.* As demonstrated in Fig. 1.1, our framework could be seen via two stages:

(i) *Prerequisites.* Along with the Java program of concern, it is required to provide a security policy represented by a policy file. Besides, two assumptions of points-to analysis and string analysis are made to ensure the precision of analysis result.

(ii) *Analysis and Output.* First, we will abstract the given Java program and extract essential policy details which help constructing the model, namely, inter-procedural control flow graph, analysis points, and permission requirements. Second, we perform analysis to analyze result of permission checks. Finally, a deduction is made to output a proper analysis result which is either "ALWAYS SUCCEED" or "MAY FAIL".

In the nature of formalizing sequential programs, *pushdown system* has been considered most appropriate. Having equipped with weights and conditions, it becomes sufficient to model the stack-based access control mechanism in Java 2 platform. Thus, *conditional weighted pushdown system* is our choice of formalism in the framework.

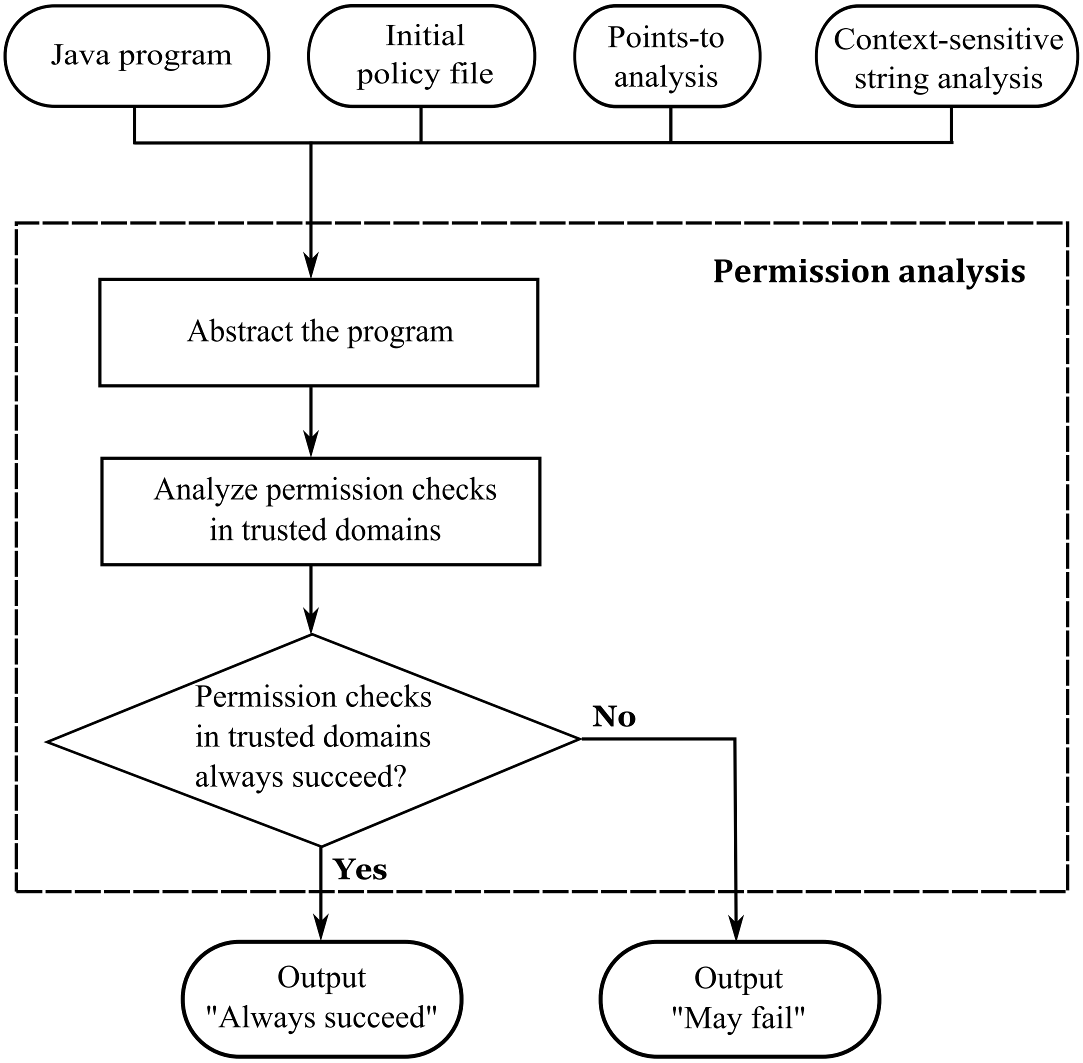


Figure 1.1: Overview of Permission Analysis Framework

1.3 Contributions

This thesis investigates and suggests an approach to automate a specific authorization test. First, we take into account possible optimization of the framework approach. Next, we systematically conduct the permission analysis.

In sum, the list of main contributions is as follows:

1. ***On-the-fly algorithm for conditional weighted pushdown systems.*** We propose an on-the-fly algorithm that efficiently reduces the state space in practice. This algorithm has been implemented in our open-source package cpwds and experimentally evaluated to prove its effectiveness. Additionally, we specify our approach with a declarative language, Datalog, and discuss its possibility to enhance performance in large-scale applications.

2. **Permission analysis.** We apply weighed pushdown system and conditional weighted pushdown system to formalize the permission checks. The analysis result of our framework provides a clue to improve Java Virtual Machine runtime performance. In sense that, one can remove redundant access-control checks that always succeed, to moderate overhead cost of program execution. Moreover, it could be also a guide to refine cumbersome security policies. Preliminarily, we demonstrate how the framework works with sample programs.

1.4 Thesis Outline

The rest contents of this thesis is organized along these lines.

- **Stack-based access control (Chapter 2).** The ultimate target of our study is to formulate and analyze the access control mechanism of Java. Thereby first of all, Chapter 2 is devoted to provide a gentle view of primitives and essential concepts involved in Java stack inspection.

- **Pushdown systems and rechability problem (Chapter 3).** The key factor of our framework formalism is the forward reachability problem in conditional weighted pushdown system. Chapter 3 recalls foundational concepts and discusses techniques that are mainly investigated in our study.

- **On-the-fly conditional weighted pushdown system algorithm and data structure (Chapter 4).** From a performance point of view, the effectiveness of realizing conditional weighted pushdown system is critical to our framework. Hence, a concerted effort has been put into creating an efficient algorithm to solve forward reachability problem in conditional weighted pushdown system. Chapter 4 presents our proposed on-the-fly algorithm as well as experience in selecting data structure for its implementation. A preliminary experimental result is furthermore reported to show the efficiency of our method.

- **Permission analysis (Chapter 5).** Admittedly, a sound solution depends upon a well-defined problem and the precision in tackling every single parts of the challenge. In Chapter 5, we first restate the problem in a more formal way and discuss obstacles to be encountered. Then the permission analysis will be partitioned into smaller analyses which are solvable by weighted pushdown system and conditional weighted pushdown system. Even more, we provide fruitful examples to demonstrate how to apply package cwpds in the framework and a blueprint of framework design through possible technical choices for implementation.

- **Related work (Chapter 6).** A literature survey on related studies is carried out in Chapter 6 along with discussion on their advantages and drawbacks.

- **Conclusion and future work (Chapter 7).** Finally, a summary of presented results will be shown in Chapter 7. Besides, we further sketch the road map of possible improvement and extension of the framework.

Chapter 2

Stack-Based Access Control

To fulfill the promise as a safe Web programming platform, rich security features have been integrated into Java ([22], [42], [27]). Among all, our interest is held by access control mechanism – the key ingredient of this thesis. Access control is a necessary approach offered to address a basic security question: *Which part of a program is allowed to access restricted resources?*

Section 2.1 lays out the design and implementation of access-control mechanism Java environment. Section 2.2 covers concepts of crucial primitives attached in Java Development Kit (JDK). The last section 2.3 describes and illustrates the base of Java access control: *Stack Inspection*. It is worth emphasizing that, although our framework is designed for *Java 2 platform*, it is also applicable for non-Java systems with similar access control model.

2.1 Access Control Mechanism in Java

Security model plays vital role among all architectural features that make Java an appropriate technology for Web environment [42].

Java 2 security is developed to moderate limitations on the sandbox security model which was employed in original release of JDK 1.0. Roughly speaking, the sandbox security model turns out to severely restrict access to valuable resources. While in real-world applications, it is expected to provide a flexible access control. In other words, the sandbox should have adjustable boundaries. In the old versions of JDK, theoretically it is possible to implement a more flexible and finer-grained access control on Java platform. However, a significant amount of effort is required to accomplish such task. Java 2 architecture aims to ease work of developers, since it provides a simpler and safer way to write security code [22].

The effort devoted to improve Java platform involves revising the design and introducing new primitives, namely, security policy, protection domain, access permission, and privileged action. These new concepts make the access control mechanism in Java 2 more robust, in terms that it supports fine-grained access control by a configurable security policy [23]. The following section focuses primarily on how the primitives are employed.

2.2 Primitives

2.2.1 Security Policy

The **security policy** of a Java system specifies its behavior in runtime. For the sake of simplicity, a security policy is an access-control table. For instance, each entry in Table 2.1 informs that "when running a program from code source X, allow it to do the actions listed in corresponding column Permissions".

Code	Permissions
Alice's applets	read "/test/question.txt"
Bob's applets	read and write "/test/answer.txt"
Local applications	read and write to "/test"

Table 2.1: A Typical Access-Control Table

From an abstract viewpoint, a security policy conceptualizes a correlation between a set of properties characterizing a code and a set of permissions granted to the code. Specifically, *code base* (location where the code comes from), *code signer* (alias of entity that signed the code) and *a list of principals* (people or any entities that may have right to take actions) constitute the set of properties used to characterize a piece of code. Such constitution is captured in the class java.security.CodeSource.

CodeSource A CodeSource object is comprised of a URL location and an array of certificates. Two CodeSource objects are considered equal if their URL locations are identical and the set of certificates attached to them are identical. As a matter of convenience, one can identify a CodeSource p is more general than a CodeSource q by checking whether $p.implies(q)$ is *true*. For example, CodeSource of http://www.jaist.ac.jp/tmp is more general than CodeSource of http://www.jaist.ac.jp/tmp/foo.jar.

2.2.2 Permissions

A **permission** represents an approval of access to concerned resources. Java 2 embraces the implementation of permission in the class java.security.Permission. Every new permission is defined by extending the Permission class, to represent a specific kind of access. To illustrate, the following Java code can be used to produce a permission to a read a file named abc.txt in the /home/admin directory.

```
perm = new java.io.FilePermission("/home/admin/abc.txt", "read");
```

In general, subclassed permissions are located in their own packages; such as, FilePermission is found in package java.io which manages the file system access. Moreover, most Permission objects include a target name and a list of *actions* permitted to do with the target. For instance, to declare a permission FilePermission, one should specify its target name by the pathname of a file or directory, and grant possible actions such as: read, write, execute, delete.

Permission implication In essence, comparing permissions is central to an access control mechanism deployment. To facilitate such comparison, Java 2 requires every permission class to implement an implies() method that define the relationship between the particular permission class and other permission classes. In more detail, given two permission objects, a and b, if $a.implies(b)$ returns *true* then any code granted a is also automatically granted b. To take an example, java.io.FilePermission("/home/admin/-", "read,write") implies java.io.FilePermission("/home/admin/abc.txt", "write"), but it does not imply java.io.FilePermission("/home/admin/xyz.txt", "execute") nor java.net.SocketPermission("jaist.ac.jp", "connect").

Permission assignment Generally, permissions are granted before the class is defined in Java runtime [22]. In the default Java 2 implementation, permissions are not altered once they are granted. It is worth noting that permissions are granted to classes, not to instances of classes.

2.2.3 Protection Domain

To put forward the extensibility, Java 2 allows permissions to be granted to **protection domains** – a concept encapsulated in class java.security.ProtectionDomain. Furthermore, all classes of the concerned program should be assigned to a protection domain and implicitly inherit all permissions granted to that domain. Informally speaking, classes from the same location and signed by the same entities are treated alike. It should be noted that each class belongs to *one and only one* protection domain. The mapping from a class to its domain is set only once, before the class comes into use, and unchanged during its lifetime (Fig. 2.1).

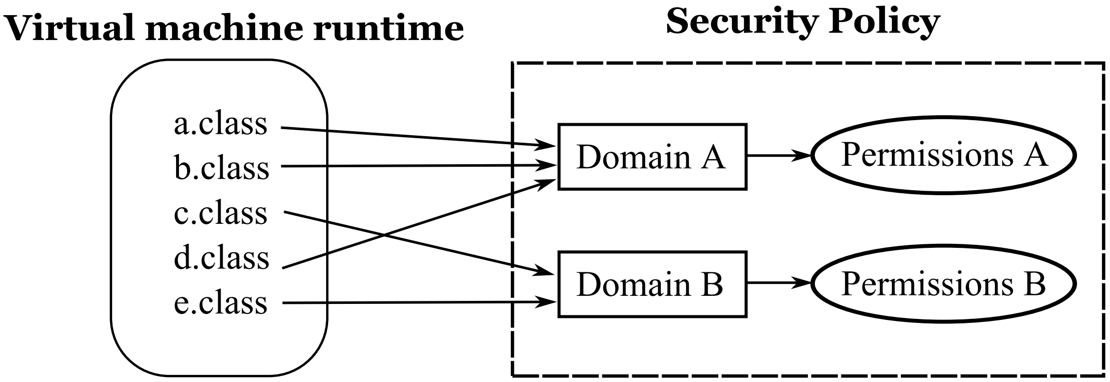


Figure 2.1: Protection Domains and Permission Grant

Java 2 describes protection domains in a fairly straightforward way, each ProtectDomain involves two parts: a CodeSource and a set of Permission. It also provides implies() method. However, unlike the purpose of comparison and testing equality in CodeSource or Permission, the implies() method in ProtectionDomain is to determine if a ProtectionDomain implies a given Permission.

2.2.4 Policy Configuration by Policy File

Policy file The default JDK implementation supports configuration of security policy via a flat-file format named **policy file**. A policy file associates permissions with protection domains and is read when the Java Virtual Machine (JVM) starts. Each policy file is made of one or multiple grant entries, every of which essentially consists of a CodeSource and its permissions. A list of signer names are included for the set of certificates that may be a part of the CodeSource. While the URL following keyword codeBase represents location where the CodeSource refer (Fig. 2.2).

```
grant signedBy "signer-name", codeBase "URL"
     principal Principal-class "principal-name",
     principal Principal-class "principal-name",
     .... {
     permission Permission-class-name "target-name "actions",
        signedBy "signer-name";
     permission Permission-class-name "target-name "actions",
        signedBy "signer-name";
     ....
};
```

Figure 2.2: Standard Format of a Policy File

Every entry in a policy file starts with keyword grant followed by optional information of codeBase, signers and a list of principals; its body is filled in a block containing one permission definition per line. Fig. 2.3 demonstrates a policy file entry granting codes from the /home/admin directory read access to the file /tmp/abc.txt and write access to the file /tmp/xyz.txt.

```
grant codeBase "file://home/admin/" {
  permission java.io.FilePermission "/tmp/abc.txt", "read";
  permission java.io.FilePermission "/tmp/xyz.txt", "write";
};
```

Figure 2.3: A Sample Policy File Entry

Configuring application-specific policy Java Development Kit (JDK) implementation obtains policy information from static locations where policy files can be found. The Java default system policy file (Fig. 2.4) is located at

```
<java.home>/lib/security/java.policy
```

Here, java.home is the directory into which JDK was installed. This java.policy is system-wide since it will be used when running any applet or application.

```
// Standard extensions get all permissions by default
grant codeBase "file:${{java.ext.dirs}}/*" {
  permission java.security.AllPermission;
};

// default permissions granted to all domains
grant {
  // Allows any thread to stop itself using the java.lang.Thread.stop()
  // method that takes no argument.
  // Note that this permission is granted by default only to remain
  // backwards compatible.
  // It is strongly recommended that you either remove this permission
  // from this policy file or further restrict it to code sources
  // that you specify, because Thread.stop() is potentially unsafe.
  // See the API specification of java.lang.Thread.stop() for more
  // information.
  permission java.lang.RuntimePermission "stopThread";

  // allows anyone to listen on un-privileged ports
  permission java.net.SocketPermission "localhost:1024-", "listen";

  // "standard" properties that can be read by anyone
  permission java.util.PropertyPermission "java.version", "read";
  permission java.util.PropertyPermission "java.vendor", "read";
  permission java.util.PropertyPermission "java.vendor.url", "read";
  permission java.util.PropertyPermission "java.class.version", "read";
  permission java.util.PropertyPermission "os.name", "read";
  permission java.util.PropertyPermission "os.version", "read";
  permission java.util.PropertyPermission "os.arch", "read";
  permission java.util.PropertyPermission "file.separator", "read";
  permission java.util.PropertyPermission "path.separator", "read";
  permission java.util.PropertyPermission "line.separator", "read";
};
```

Figure 2.4: A Portion of Default java.policy in Java Platform, Standard Edition 6

One can specify additional policy files when executing an application via the -Djava.security.policy command line argument:

```
java -Djava.security.manager -Djava.security.policy=myURL myApplication
```

where myURL is a URL specifying location of a policy file, and -Djava.security.manager is to invoke the default security manager so that the security policy will come into effect when running the application myApplication.

2.3 Access Control by Stack Inspection

In Java, **stack inspection** is the base of access control [23]. The big picture: *when a security-sensitive operation is invoked, all methods currently in the call stack will be inspected in top-down manner to see whether their declaration classes have required permission.* There are many strategies to implement stack inspection, two obvious ones are: eager evaluation and lazy evaluation [6].

Eager evaluation The set of effective permissions is updated dynamically at each method call and return.

Lazy evaluation The call stack is inspected on demand at authorization checkpoints. Java 2 adopted this lazy semantics for performance and scalability.

An elaborate Application Programming Interface (API), first introduced in JDK 1.2, triggers stack inspection via method java.security.AccessController.checkPermission(Permission). The method returns silently if the specified Permission is allowed in the current execution context. Otherwise, it throws a java.security.AccessControlException. This new API is known as a comprehensive implementation since it fully embraces both the basic access control algorithm, and the extended one with privileged operations.

2.3.1 Basic Algorithm

The basic access-control algorithm can be intuitively expressed as in Fig. 2.5 [22].

```
for each caller in the current execution context {
    if the caller does not have the requested permission
       throws a java.security.AccessControlException
}
return normally
```

Figure 2.5: The Basic Access-Control Algorithm

In the obsolete versions of JDK, when a code evaluated an access control decision, it ought to know the status of its callers as well as preceding callers in the call chain. This burden is now relieved for programmers, since in new implementation of java.security.AccessController the access checking process or stack inspection is automated. Let us illustrate the algorithm by Example 2.3.1.

Example 2.3.1. *Given a policy configuration as Table 2.2.*

All codes in package App.Faculty and App.Student are assigned to domain FACULTY and domain STUDENT respectively. Besides, codes that belong to Java system libraries are associated with domain SYSTEM, whereas codes in App.App stay in domain APP. Fig. 2.6 shows the Java codes of two classes in App.Faculty and App.Student and how they invoke permission check.

Protection domain	Permissions
SYSTEM	all existing permissions
FACULTY	read and write "/test/abc.txt"
STUDENT	read "/test/abc.txt"
APP	read, write and delete "/test/abc.txt"

Table 2.2: A Sample Policy Configuration

```
package App.Faculty;
public class Teacher {
  public static void foo() {
    ... some code ...
    FilePermission perm = new FilePermission("/test/abc.txt", "write");
    AccessController.checkPermission(perm);
    ... write something to /test/abc.txt ...
  }
}
```

```
package App.Student;
public class Student {
  public static void foo() {
    ... some code ...
    FilePermission perm = new FilePermission("/test/abc.txt", "write");
    AccessController.checkPermission(perm);
    ... write something to /test/abc.txt ..
  }
}
```

Figure 2.6: Sample Java Codes Invoking Permission Check

Suppose an application class MainApp in domain APP calls method foo in class Teacher to write the file /test/abc.txt. The execution context will look like Fig. 2.7. To give a clean intuition, we eliminated intermediate system calls from the figure. In this case, three distinct domains appear within the execution context: SYSTEM (stack frame 3), TEACHER (stack frame 2) and APP (stack frame 1). The stack inspection returns silently as all mentioned domains in the call chain are granted permission of write access to /test/abc.txt.

While in case of class Student (Fig. 2.8), the stack walk stops at stack frame 2, and an exception java.security.AccessControlException will be thrown due to the lack of write permission in domain STUDENT.

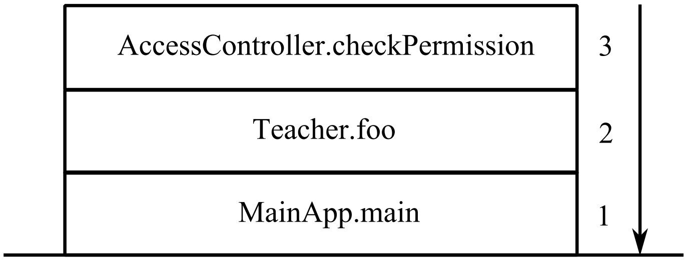


Figure 2.7: Snapshot of Call Stack When Method foo of Class Teacher is Invoked

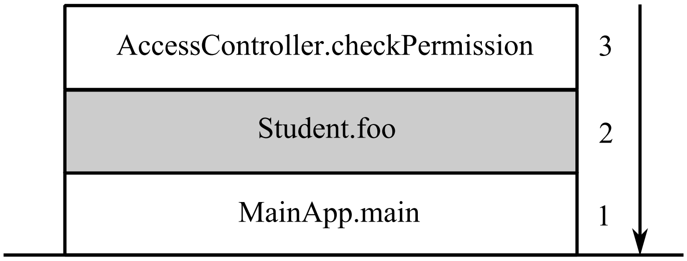


Figure 2.8: Snapshot of Call Stack When Method foo of Class Student is Invoked

2.3.2 Extended Algorithm with Privileged Operation

In practice, there are occasions where the codes closer to top of the call stack want to perform some actions that codes further down call stack are not allowed to do. A good example [42] is, suppose an untrusted applet, with no local file access permission, asks the Java API to render a string of text in font Helvetica. Apparently, the API needs to load the font file of Helvetica from local disk to render on behalf of the applet. In the process of opening font file, a file access permission check will be invoked. Although the class making explicit request to open file belongs to system domain with full permissions, the stack inspection still fails. Because the applet class sitting further down the call stack is inspected for no access permission to local files.

For such occasions, the basic algorithm appears too restrictive. To cope with exceptional cases, AccessController offers a static method doPrivileged(). A method M that calls doPrivileged() is telling the Java runtime to ignore the status of its callers since it will take responsibility to perform the action. In other words, a privileged method is allowed to execute any code with permissions granted to its caller, *regardless of the calling sequence.* Fig. 2.9 [22] describes the extended algorithm to handle the privilege status. In an attempt to avoid possible confusion, we stress that the method doPrivileged() being discussed here is implemented in recent API of Java 2, which wraps entire enable-disable cycle in a single interface [27] rather than keeping track of enable-disable processes separately as in previous versions of API [43].

```
for each caller in the current execution context {
    if the caller does not have the requested permission
       throws a java.security.AccessControlException;
    if the caller is privileged, return normally;
}
return normally
```

Figure 2.9: The Extended Access-Control Algorithm with Privileged Operation

To perform a privileged action A, method M invokes AccessControler.doPrivileged(A); this invocation involves calling method A.run() with *all the permissions* of M enabled. Java 2 API provides two interfaces to declare a privileged action: PrivilegedAction and PrivilegedExceptionAction. The difference between these two interfaces is the latter can throw checked exceptions.

Fig. 2.10 demonstrates a sample piece of code that calls doPrivileged with a PrivilegedAction.

```
methodM() {
  ... normal code ...
  AccessController.doPrivileged(new PrivilegedAction() {
    public Object run() {
      ... do some privileged action, e.g., opening font file ...
      return null;
    }
  });
  ... normal code ...
}
```

Figure 2.10: A Piece of Code that Calls doPrivileged

As a case in point, let us consider Example 2.3.2.

Example 2.3.2. *We extend the Example 2.3.1 with three new classes: Observer1 and Observer2 and AdvancedStudent (Fig. 2.11).*

```
package App.App;
public class Observer2 {
  public static void foo() {
    FilePermission perm = new FilePermission("/test/abc.txt", "write");
    AccessController.checkPermission(perm);
    ... write something to /test/abc.txt ...
  }
}
```

```
package App.App;
public class Observer1 {
  public static void foo() {
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        Observer2.foo();
        return null;
      }
    });
  }
}
```

```
package App.Student;
public class AdvancedStudent {
  public static void foo() {
     Observer1.foo();
  }
}
```

Figure 2.11: Sample Java Codes Invoking Privileged Action

In this case, when MainApp.main invokes AdvancedStudent.foo (Fig. 2.12), two new frames are pushed into call stack: AccessController.doPrivileged (domain SYSTEM) and run (the same domain APP as Observer1). The permission check will succeed even when AdvancedStudent does not have write access to file /test/abc.txt. Strictly speaking, frames 4-7 pass the inspection as they belong to domains that granted sufficient permission. When encountering AccessController.doPrivileged, the inspection process will examine the caller of AccessController.doPrivileged, i.e. Observer1.foo, and returns normally since Observer1 belongs to domain APP. The stack frame of AdvancedStudent.foo will never be examined overall.

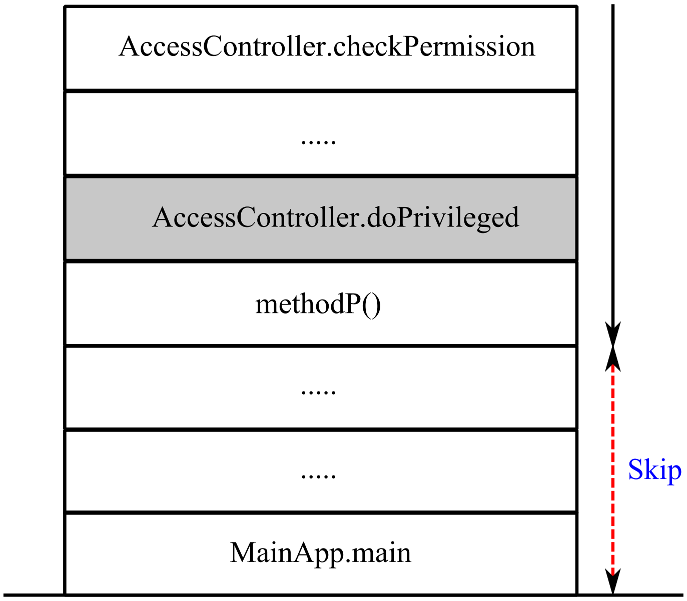

AccessController.checkPermission
.....
AccessController.doPrivileged
methodP()
.....
.....
MainApp.main
Skip

Figure 2.12: Snapshot of Call Stack When Method foo of AdvancedStudent is Invoked

One worth highlighting point of the extended access control algorithm is that the caller of AccessController.doPrivileged is always inspected; thereby an AccessController.doPrivileged invocation will be futile if its caller does not have permission to execute the expected action.

Chapter 3

Pushdown Systems and Reachability Problem

Model-checking pushdown systems (PDSs) have served the heart of sequential program analysis due to their natural model for handling (recursive) procedure calls by using stacks. Consequently, over the past decade, researchers have shown an increased interest in improving analysis techniques based on pushdown systems. In particular, Schwoon et al. [39] first introduced *weighted pushdown systems* (WPDSs) to address the authorization problems. Later, Reps et al. [36] made use of WPDS to tackle the generalized reachability problems and their application to interprocedural dataflow analysis.

When context-sensitive information is taken into account, a *valid path* essentially comprises pairs of procedure calls and returns. There are two approaches: *context-cloning* and *context-stacking*. The former bounds the depth of nested contexts, and the latter uses a pushdown stack as PDS. Generally, the latter approach is expected to be precise, but in some cases we need to look into the content of a stack. For instance, a method invocation in object-oriented languages depends on dynamic types, which is propagated in a sequence of invocations. More directly, stack inspection is frequently used in security applications. To alleviate the problem, [26] extended WPDS to *conditional weighted pushdown system* (CWPDS) by associating conditions to each transition rule. Theoretically, conditions described by regular sets are naturally obtained by enriching stack alphabet as products [26], which easily leads to stack alphabet explosion. Instead of improving theoretical complexity, our aim is to design a practically efficient algorithm.

This chapter recalls foundational concepts and discusses techniques that are mainly used in our study. Section 3.1 and Section 3.2 concern *pushdown systems* and *reachablility problem*, a formalism that facilitates interprocedural dataflow analysis. Section 3.3 and Section 3.4 explain the extension of pushdown system with weight and condition. To keep the content on point, we restrict us to essential definitions and properties. For comprehensive introduction to dataflow analysis and pushdown model checking, we refer to [19], [40], [38], and [30].

3.1 Pushdown System (PDS)

A ***pushdown system*** (PDS) is a formalism induced by a pushdown automaton. It is equipped with a finite set of *control locations* and *a stack*. Content of the stack is defined by a word of unbounded length over some finite *stack alphabet*. Consequently, a pushdown system is capable of verifying infinite-state systems [35].

Definition 3.1.1. *A* ***pushdown system*** $\mathcal{P} = (Q, \Gamma, \Delta, q_0, \omega_0)$ *features: a finite set of control location* Q*, a finite stack alphabet* Γ*, a finite set of rules* $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$*, the initial control location* $q_0 \in Q$ *and the initial stack contents* $\omega_0 \in \Gamma^*$*. A pair* $\langle p, \omega \rangle$*,* $p \in Q$*,* $\omega \in \Gamma^*$ *is called* ***configuration*** *of* $\mathcal{P}$*. A transition relation* $\Rightarrow$ *between configurations of* $\mathcal{P}$ *is defined so that* $\langle p, \gamma\omega' \rangle \overset{\langle r \rangle}{\Longrightarrow} \langle p', \omega\omega' \rangle$*, for all* $\omega' \in \Gamma'$ *if there is* $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle$*. The reflexive transitive closure of* $\Rightarrow$ *is denoted by* $\Rightarrow^*$*. Given a set of configurations* C*, we define* $post^*(C) \overset{def}{=} \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$*.*

3.2 Reachability Problem

Reachability problem is reduced from a natural class of dataflow analysis problems in which checking *safeness properties* is brought into focus, e.g. "Whenever a sorting procedure finishes, the input array is sorted correctly." [38] Given a set of configurations C (Fig. 3.1), there are two versions of reachability problem: *forward* (set $post^*(C)$) and *backward* (set $pre^*(C)$). In this study, our interest is mainly in the forward reachability.

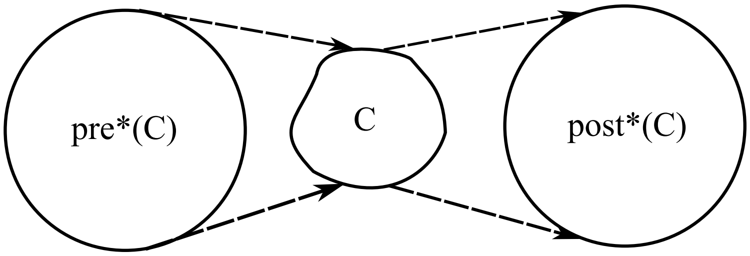


Figure 3.1: Two Sets $pre^*(C)$ and $post^*(C)$

Definition 3.2.1. *Given a pushdown system* P*, a set of configurations* C *and a configuration* c*. The* ***forward reachability (****$post^*$****) problem*** *is to check whether* $c \in post^*(C)$ *holds.*

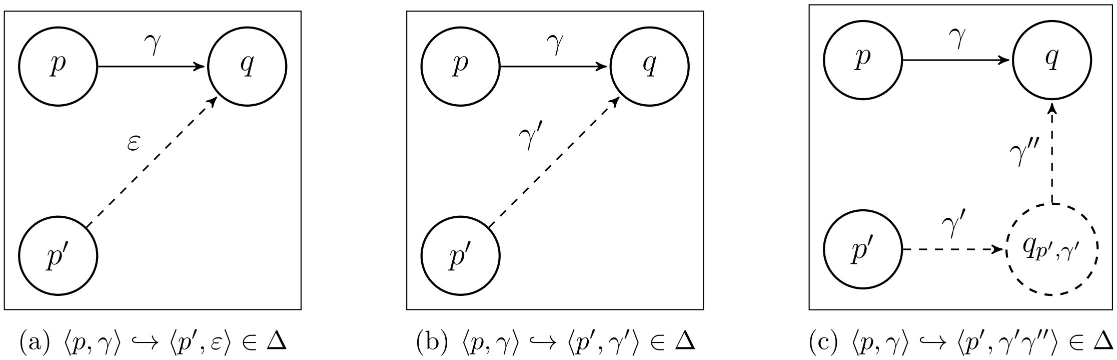
As mentioned, pushdown systems may have unbounded number of configurations. It is helpful to use finite automata, i.e., $\mathcal{P}$-automaton, to represent regular sets of configurations.

Definition 3.2.2. *Let a pushdown system* $P = (Q, \Gamma, \Delta)$*, and* C *be a set of configurations of* P*. A* ***$\mathcal{P}$-automaton*** *for* C *is a nondeterministic finite automaton that accepts from an initial state* $q \in Q$ *exactly the word* ω *such that* $p\omega \in C$*.*

Saturation algorithm Typically, $\mathcal{P}$-automaton [15] is used to tackle reachability problem because of its efficiency in practice [40]. The basic idea of $\mathcal{P}$-automaton algorithm is that:

1. Starting from a finite automaton representing a set of initial configurations C.
2. The algorithm finds all reachable states by adding transitions into the automaton until it is *saturated*.

Hence, $\mathcal{P}$-automaton algorithm is also known as *Saturation algorithm*. For instance, in the *post** problem, the saturation rule is: *If the left-hand side of the rule existed, make a path for the right-hand side.* In Fig. 3.2, the dashed nodes and dashed edges are added in the saturation process.



(a) $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$ (b) $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ (c) $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$

Figure 3.2: *post** Saturation Rule

3.3 Weighted Pushdown System (WPDS)

A **weighted pushdown system (WPDS)** is obtained by augmenting pushdown transition rules with a weight domain drawn from a *bounded idempotent semiring*. Instead of direct encoding as the product of control states and environments, such semirings are informative enough to enable us to describe dataflow as weights. Thus dataflow summary from a source to a destination can be computed by accumulating weights along the flow paths [36]. This approach gives opportunity to avoid state explosion.

Definition 3.3.1. *$\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is a **bounded idempotent semiring** iff:*
1. $(D, \oplus)$ is a commutative monoid with unit element $\bar{0}$, and $a \oplus a = a, \forall a \in D$.
2. $(D, \otimes)$ is a monoid with unit element $\bar{1}$.
3. $\otimes$ distributes over $\oplus$.
4. $\forall a \in D, a \otimes \bar{0} = \bar{0} = \bar{0} \otimes a$.
5. No infinite descending chains in the partial order $\sqsubseteq$ induced by $a \sqsubseteq b$ iff $a \oplus b = a$.

Definition 3.3.2. *A **weighted pushdown system** is a triplet $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ features a pushdown system $\mathcal{P} = (Q, \Gamma, \Delta, q_0, \omega_0)$, a **bounded idempotent semiring** $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ and a valuation function $f : \Delta \to D$.*

To the extent of program analysis, $\bar{1}$ implies identity function, $\bar{0}$ deduces program interruption. Meanwhile, $\otimes$ indicates merge of two dataflows at the meet, and $\oplus$ represents compositions of two dataflows.

The algorithm to solve weighted case of *post** problem proposed in [36] has the same flavor as unweighted one, in sense that they all use $\mathcal{P}$-automaton technique. However, the saturation rule is slightly changed:

1. Starting with a $\mathcal{P}$-automaton $\mathcal{A}$ which accepts the set of configurations C.

2. All of the transitions in $\mathcal{A}$ are initially labeled with $\bar{1}$.

3. Applying the saturation rule of unweighted case to introduce new transitions (p, γ, q) to $\mathcal{A}$; if the transition (p, γ, q) already occurs in $\mathcal{A}$, its label will be updated by *combine* operation $\oplus$.

4. The weight of a path in $\mathcal{A}$ is computed by taking the *extend* operation $\otimes$ of the labels on the transitions along the path.

Reachability problem on PDSs can be generalized to WPDSs via *Meet-over-all-valid-paths (MOVP)* problem.

Definition 3.3.3. *Let $W = (P, \mathcal{S}, f)$ be a weighted pushdown system where $P = (Q, \Gamma, \Delta, q_0, \omega_0)$. Let $\sigma = [r_1, \ldots, r_k] \in \Delta^*$ denote a sequence of pushdown transition rules; and $\boldsymbol{v}(\sigma) = f(r_1) \otimes \cdots \otimes f(r_k)$. We write $c \overset{\sigma}{\Rightarrow}{}^k c'$ if $c \overset{r_1}{\Rightarrow} c_1 \overset{r_2}{\Rightarrow} \ldots \overset{r_k}{\Rightarrow} c'$ for any $k \in \mathbb{N}$ and $c, c', c_i \in Q \times \Gamma^*$ with $1 \leq i \leq k$. Let $S, T \subseteq Q \times \Gamma^*$ be regular sets of configurations. For any configuration $c \in Q \times \Gamma^*$, the* ***Meet-over-all-valid-paths (MOVP)*** *problem is to compute $MOVP(S, T) = \bigoplus\{\boldsymbol{v}(\sigma) \mid c \overset{\sigma}{\Rightarrow}{}^* c', c \in S, c' \in T\}$. To specify the weighted pushdown system W associated with an MOVP problem, we may use notation $MOVP(S, T, W)$.*

3.4 Conditional Weighted Pushdown System (CWPDS)

A **conditional weighted pushdown system (CWPDS)** lifts an underlying WPDS to a conditional system by associating conditions to each transition rule [26]. Herein, conditions are set on stack content ω except the topmost stack symbol γ. As the the stack is traversed in bottom-up order, ω is scanned in reverse order (Fig. 3.3).

Definition 3.4.1. *A* ***conditional pushdown system (CPDS)*** *is a 6-tuple $\mathcal{P}_c = (Q, \Gamma, \Delta_c, \mathcal{C}, q_0, \omega_0)$ features: a finite set of control locations Q, a finite stack alphabet Γ, a finite set of regular languages $\mathcal{C} = \{L_1, \ldots, L_n\}$ over Γ, $\Delta_c \subseteq Q \times \Gamma \times \mathcal{C} \times Q \times \Gamma^*$, the initial control location $q_0 \in Q$ and the initial stack contents $\omega_0 \in \Gamma^*$. A computation relation $\Rightarrow_c$ on configurations is defined as $\langle p, \gamma\omega' \rangle \Rightarrow_c \langle q, \omega\omega' \rangle$ for all $\omega' \in \Gamma^*$ if $\langle p, \gamma \rangle \overset{L}{\hookrightarrow} \langle q, \omega \rangle$ and $\omega' \in L$.*

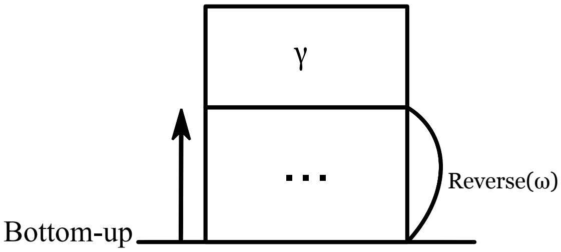


Figure 3.3: Conditions are Set on Stack Content

Given a regular language L, a **condition automaton** with respect to L is a *total deterministic finite automaton.*

Definition 3.4.2. *A* ***conditional weighted pushdown system (CWPDS)*** *is a triplet* $\mathcal{W} = (\mathcal{P}_c, \mathcal{S}, f)$ *features: a conditional pushdown system* $\mathcal{P}_c = (Q, \Gamma, \Delta_c, \mathcal{C}, q_0, \omega_0)$*, a semiring* $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ *and a valuation function* $f : \Delta \to D$*.*

Li et al. [26] showed that model checking problems on CWPDSs can be reduced to those on WPDSs, and further proposed an *offline algorithm* for translating CWPDSs to WPDSs. Given a set of regular languages $\mathcal{C} = \{L_1, \ldots, L_n\}$, the authors managed to settle the translation by synchronizing the underlying weighted pushdown system and the condition automaton $\prod_{1 \le i \le n} A_i$, which is a Cartesian product of all condition automata A_i with respect to conditions L_i. Fig. 3.4 is a summary for Algorithm TRANS demonstrated in [26].

- Set of condition DFA $A = \{A_i \mid 1 \le i \le n\}$ where $A_i = (S_i, \Gamma, \delta_i, \dot{\mathbf{s}}_\mathbf{i}, F_i)$.
- Constructing the Cartesian product $\prod_{1 \le i \le n} A_i = (\hat{S}, \Gamma, \hat{\delta}, \dot{\mathbf{s}}, \hat{F})$.
- Translating rules:

$$\begin{array}{lcl c lcl}
\langle p, \gamma \rangle & \overset{A_i}{\hookrightarrow} & \langle q, \varepsilon \rangle & & \langle p, (\gamma, r) \rangle & \hookrightarrow & \langle q, \varepsilon \rangle \\
\langle p, \gamma \rangle & \overset{A_i}{\hookrightarrow} & \langle q, \gamma' \rangle & \Longrightarrow & \langle p, (\gamma, r) \rangle & \hookrightarrow & \langle q, (\gamma', r) \rangle \\
\langle p, \gamma \rangle & \overset{A_i}{\hookrightarrow} & \langle q, \gamma' \gamma'' \rangle & & \langle p, (\gamma, r) \rangle & \hookrightarrow & \langle q, (\gamma', t)(\gamma'', r) \rangle
\end{array}$$

  where $r \in \hat{S}$, $r_i \in F_i$ and $\hat{\delta}(r, \gamma'') = t$.
- Translating weights: $f'(r') = f(r)$.

Figure 3.4: Translating CWPDSs to WPDSs by Algorithm TRANS [26]

One should remark here, that above algorithm would produce entire set of stack alphabet at once to complete the outcome WPDS. Inevitably it involved redundant stack symbols and transition rules which may never be used in later saturation process. Moreover, as the extended stack alphabet Γ' is a product of Γ and $\hat{S}$, a blow-up in the number of states of $\prod_{1 \le i \le n} A_i$ easily leads to explosion of Γ'. Hereafter, we design a new approach to realize CWPDS more efficiently in practice.

Chapter 4

On-The-Fly CWPDS Algorithm and Data Structure

This chapter is devoted to describe our contribution in realizing conditional weighted pushdown and enabling its application for large-scale programs. Section 4.1 is covered by details of our solution to enhance performance of conditional weighted pushdown system. Section 4.2 discusses a specification of our algorithm written in the declarative language Datalog. Lastly, Section 4.3 reports a preliminary experiment we conduct to evaluate performance of our method.

4.1 On-The-Fly Algorithm

Intuitively speaking, to reduce space consumption and runtime, one should generate stack alphabet and examine conditions only when necessary. We interleave this on-the-fly manner with the weighted $\mathcal{P}$-automaton construction in [36]; that means for a CWPDS transition rule, the condition associated with it will be checked only when the left-hand-side of the rule is being considered in the saturation process.
In sum, the effectiveness of *post*$*$ algorithm for CWPDS case is achieved via two phases:

1. Translating initial $\mathcal{P}$-automaton of Conditional pushdown system to one in pushdown system.

2. Applying *post*$*$ algorithm for WPDS case with on-the-fly manner.

Translating initial $\mathcal{P}$-automaton The idea formalized in Alg. 1 contains two steps as follows.

1. Given a $\mathcal{P}$-automaton $\mathcal{A}_c$ of Conditional PDS, starting from a pair consisting of a final state in $\mathcal{A}_c$ and the initial state of $\prod_{1\leq i\leq n} A_i$.

2. Extending and adding rules to the outcome $\mathcal{P}$-automaton $\mathcal{A}_t$ until it is saturated. To illustrate, let $(p, \mathbf{s})$ be a considered pair, add $(q, \mathbf{t}) \xrightarrow{(b,\mathbf{s})} (p, \mathbf{s})$ iff there exists $q \xrightarrow{b} p$ in $\mathcal{A}_c$ and $\mathbf{s} \xrightarrow{b} \mathbf{t}$ in $\prod_{1\leq i\leq n} A_i$. See Example 4.1.1 for better understanding.

The intuition behind is that, we mimic traversal of the product of condition automata.

Algorithm 1 $\mathcal{P}$-automaton translation

Input: a conditional pushdown system $P_c = (Q, \Gamma, \mathcal{C}, \Delta, q_0, \gamma_0)$ with $\mathcal{C} = \{L_1, \ldots, L_n\}$;
a set of condition automata $A = \{A_1, \ldots, A_n\}$ with $A_i = \phi(L_i^R) = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$ for $1 \leq i \leq n$;
a $\mathcal{P}$-automaton $\mathcal{A}_c = (Q', \Gamma, \rightarrow_c, Q, F_c)$ that accepts $C_s \subseteq Q \times \Gamma^*$, such that $\mathcal{A}_c$ has no transitions into Q states and has no ε-transitions.

Output: a $\mathcal{P}$-automaton $\mathcal{A}_t = (H, \Gamma', \rightarrow_0, Q, F)$ that accepts C'_s where $C'_s = \{\rho^{-1}(c) \mid c \in C_s\}$, such that $\mathcal{A}_t$ has no transitions into Q states and has no ε-transitions.

```
function buildPA($P_c, A, \mathcal{A}_c$)
begin
    let $\hat{\delta}(s_1, s_2, \ldots, s_n, \gamma) = (\delta_1(s_1, \gamma), \delta_2(s_2, \gamma), \ldots, \delta_n(s_n, \gamma))$ for $s_i \in S_i$
      with $i \in \{1, 2, \ldots, n\}$ and $\gamma \in \Gamma$
    $\dot{\mathbf{s}} := (\dot{s}_1, \dot{s}_2, \ldots, \dot{s}_n)$
    $\Gamma' := \Gamma \times (S_1 \times S_2 \times \cdots \times S_n)$
    $H := \emptyset$
    for each $q \in F_c$ do
        $H := H \cup \{(q, \dot{\mathbf{s}})\}$

    $F := H$
    $workset := H$
    $\rightarrow_0 := \emptyset$
    while $workset \neq \emptyset$ do
        select and remove a state $u = (p, \mathbf{s})$ from $workset$
        for each $r = (q, \gamma, p) \in \rightarrow_c$ do
            $\mathbf{t} := \hat{\delta}(\mathbf{s}, \gamma)$
            $H := H \cup \{(q, \mathbf{t})\}$
            $\rightarrow_0 := \rightarrow_0 \cup \{((q, \mathbf{t}), (\gamma, \mathbf{s}), (p, \mathbf{s}))\}$        ▷ add transition $(q, \mathbf{t}) \xrightarrow{(\gamma,\mathbf{s})} (p, \mathbf{s})$
            $workset := workset \cup \{(q, \mathbf{t})\}$
    for each $r = ((q, \mathbf{t}), (\gamma, \mathbf{s}), (p, \mathbf{s})) \in \rightarrow_0$ do
        if $q \in Q$ then
            $H := (H \setminus \{(q, \mathbf{t})\}) \cup \{(q)\}$        ▷ replace $(q, \mathbf{t})$ by $(q)$
            $\rightarrow_0 := (\rightarrow_0 \setminus \{((q, \mathbf{t}), (\gamma, \mathbf{s}), (p, \mathbf{s}))\}) \cup \{((q), (\gamma, \mathbf{s}), (p, \mathbf{s}))\}$
    return $(H, \Gamma', \rightarrow_0, Q, F)$
end
```

On-the-fly *post*∗ algorithm for Conditional weighted pushdown system It is listed in Alg. 2. Line 19, 22, 25 describe the on-the-fly checking of conditions.

Algorithm 2 On-the-fly *post*∗ algorithm for Conditional weighted pushdown system

```
Input: a conditional weighted pushdown system $W_c = (P_c, S, f)$,
       where $P_c = (Q, \Gamma, \mathcal{C}, \Delta, q_0, \gamma_0)$ with $\mathcal{C} = \{L_1, \ldots, L_n\}$ and $S = (D, \oplus, \otimes, \bar{0}, \bar{1})$;
       a $\mathcal{P}$-automaton $\mathcal{A}_c = (Q', \Gamma, \rightarrow_c, Q, F_c)$ that accepts $C_s \subseteq Q \times \Gamma^*$, such that $\mathcal{A}_c$
       has no transitions into $Q$ states and has no $\varepsilon$-transitions.
Output: a $\mathcal{P}$-automaton $\mathcal{A}_{post^*} = (H', \Gamma', \rightarrow, Q, F)$ with $\varepsilon$-transitions that accepts
       $post^*(C'_s)$ where $C'_s = \{\rho^{-1}(c) \mid c \in C_s\}$;
       a function $l$ that maps every $(h, \gamma, h') \in \rightarrow$ to a weight taken from semiring $S$.

 1: procedure UPDATE$(t, v)$
    begin
 2:     $\rightarrow := \rightarrow \cup \{t\}$
 3:     $newValue := l(t) \oplus v$
 4:     $changed := (newValue \neq l(t))$
 5:     if $changed$ then
 6:         $workset := workset \cup \{t\}$
 7:         $l(t) := newValue$
    end
 8:
 9: let $A = \{A_1, \ldots, A_n\}$ with $A_i = \phi(L_i^R) = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$ for $1 \leq i \leq n$
10: let $\hat{\delta}(s_1, s_2, \ldots, s_n, \gamma) = (\delta_1(s_1, \gamma), \delta_2(s_2, \gamma), \ldots, \delta_n(s_n, \gamma))$ for $s_i \in S_i$
        with $i \in \{1, 2, \ldots, n\}$ and $\gamma \in \Gamma$
11: $\mathcal{A}_t :=$ BUILDPA$(P_c, A, \mathcal{A}_c)$   ▷ call function in Alg. 1 to obtain $\mathcal{A}_t = (H, \Gamma', \rightarrow_0, Q, F)$
12: $\rightarrow := \rightarrow_0$; $workset := \rightarrow_0$; $l := \lambda t.\bar{0}$
13: for all $t \in \rightarrow_0$ do $l(t) := \bar{1}$
14: $H' := H$
15: while $workset \neq \emptyset$ do
16:     select and remove a transition $t = (p, (\gamma, \mathbf{u}), q)$ from $workset$
17:     if $\gamma \neq \varepsilon$ then
18:         for all $r = \langle p, \gamma \rangle \overset{A_i}{\hookrightarrow} \langle p', \varepsilon \rangle \in \Delta$ with $A_i = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$ do
19:             if $\mathbf{u}_i \in F_i$ then                          ▷ Condition $A_i$ is satisfied
20:                 UPDATE$((p', \varepsilon, q), l(t) \otimes f(r))$
21:         for all $r = \langle p, \gamma \rangle \overset{A_i}{\hookrightarrow} \langle p', \gamma' \rangle \in \Delta$ with $A_i = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$ do
22:             if $\mathbf{u}_i \in F_i$ then
23:                 UPDATE$((p', (\gamma', \mathbf{u}), q), l(t) \otimes f(r))$
24:         for all $r = \langle p, \gamma \rangle \overset{A_i}{\hookrightarrow} \langle p', \gamma'\gamma'' \rangle \in \Delta$ with $A_i = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$ do
25:             if $\mathbf{u}_i \in F_i$ then
26:                 let $\mathbf{z} = \hat{\delta}(\mathbf{u}, \gamma'')$
```

```
                $H' := H' \cup \{h_{p',(\gamma',\mathbf{z})}\}$
                UPDATE$((p', (\gamma', \mathbf{z}), h_{p',(\gamma',\mathbf{z})}), \bar{1})$
                UPDATE$((h_{p',(\gamma',\mathbf{z})}, (\gamma'', \mathbf{u}), q), l(t) \otimes f(r))$
                if $changed$ then
                    for all $t' = (p'', \varepsilon, h_{p',(\gamma',\mathbf{z})})$ do
                        UPDATE$((p'', (\gamma'', \mathbf{u}), q), l(t) \otimes f(r) \otimes l(t'))$
    else
        for all $t' = (q, (\gamma', \mathbf{w}), q') \in \rightarrow$ do UPDATE$((p, (\gamma', \mathbf{w}), q'), l(t') \otimes l(t))$
return $((H', \Gamma', \rightarrow, Q, F), l)$
```

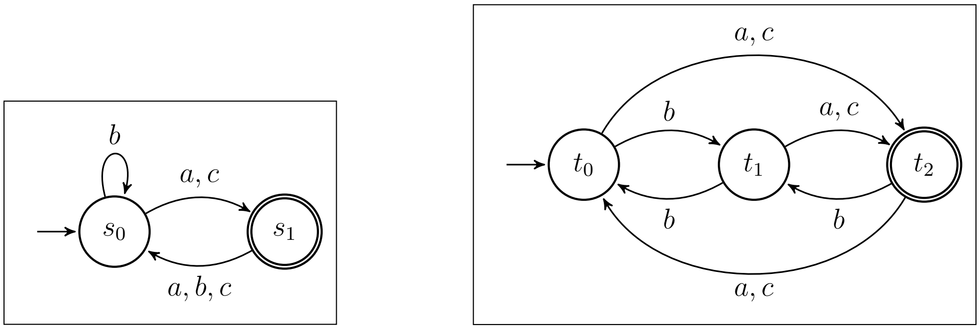
Performance of the offline algorithm (Fig. 3.4) and on-the-fly algorithm (Alg. 2) are implemented and experimentally evaluated in Section 4.3.

Example 4.1.1. *Consider a conditional weighted pushdown system with:*

- *Stack alphabet* $\Gamma = \{a, b, c\}$
- *The initial* $\mathcal{P}$*-automaton with two control locations* p, p' *(Fig. 4.3).*
- *Semiring* $\mathcal{S} = (\mathbb{N}, min, +, \infty, 0)$
- *Set of pushdown transition rules with condition and weight associated to each rule:*

$$\Delta = \left\{ \begin{array}{lcl} \langle p, b \rangle & \overset{A_0,1}{\hookrightarrow} & \langle p', c \rangle \\ \langle p', c \rangle & \overset{A_1,1}{\hookrightarrow} & \langle p', ab \rangle \\ \langle p', c \rangle & \overset{A_0,1}{\hookrightarrow} & \langle p, \varepsilon \rangle \end{array} \right\}$$

The regular languages representing the conditions are described by two deterministic finite automata A_0, A_1 *(Fig. 4.1) and their Cartesian product* A *is depicted in Fig. 4.2.*
The result after applying Alg. 1 and Alg. 2 are shown in Fig. 4.4 and Fig. 4.5 respectively.



(a) A_0

(b) A_1

Figure 4.1: Condition DFA A_0 and A_1

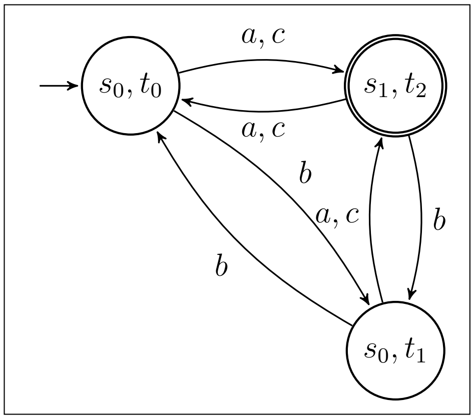


Figure 4.2: Product of Condition DFA $A = \{A_0, A_1\}$

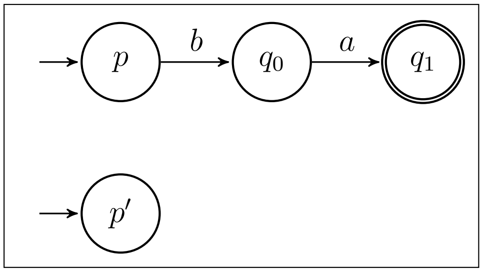


Figure 4.3: Initial $\mathcal{P}$-automaton

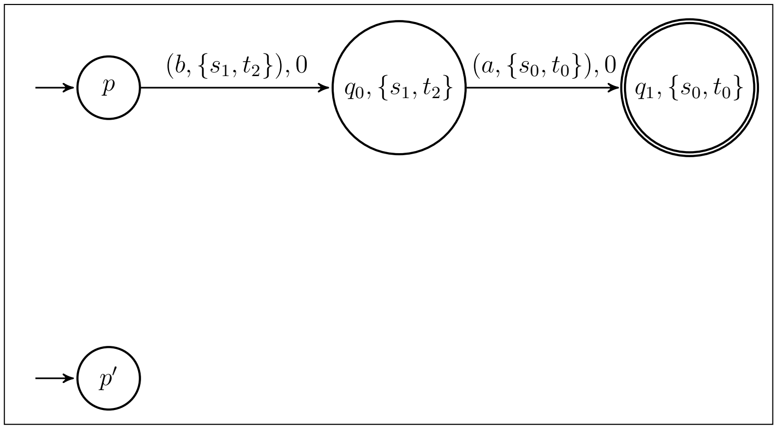


Figure 4.4: The Initial $\mathcal{P}$-automaton after Applying the Translation

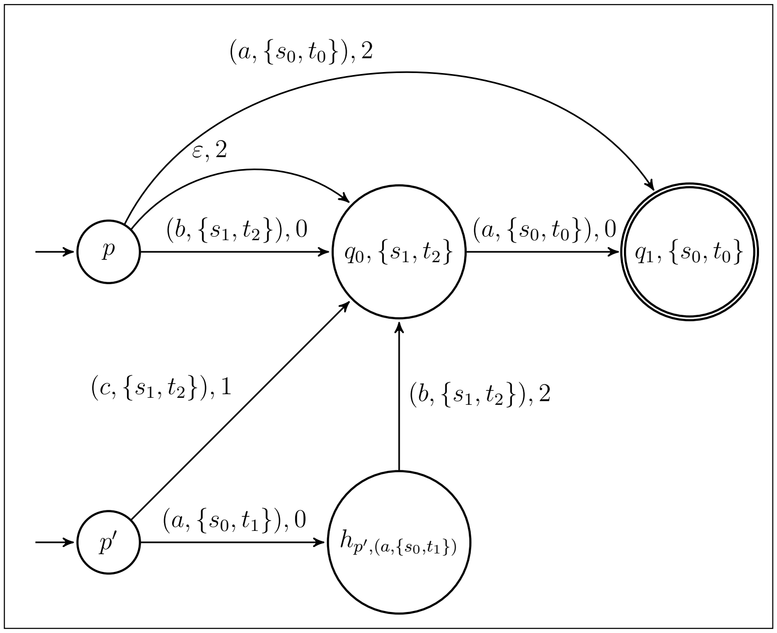


Figure 4.5: Final Result of Alg. 2.

By generating stack alphabet on-demand, we have avoided redundant symbols, such as: $(b, \{s_0, t_0\})$, $(c, \{s_0, t_0\})$, $(a, \{s_1, t_2\})$, $(b, \{s_0, t_1\})$, $(c, \{s_0, t_1\})$.

4.2 Data Structure for Conditional Pushdown System

Dealt with large-scale applications, what to seek is a good data structure for symbolic representation. It should be compact and provides efficient operations for program-analysis algorithms. Binary Decision Diagram (BDD) [1] is a structure widely used for this purpose. As our investigation into popular BDD implementations, bddbddb (**BDD**-**B**ased **D**eductive **D**ata**B**ase) library provides the most natural and concise front-end ([44], [45]). Specifically, a program analysis can be developed simply by writing a specification in a declarative programming language *Datalog*. Later, bddbddb will automatically transform the specification into efficient BDD operations. [46] reported that bddbddb was able to analyze applications containing up to 4×10^{14} context-sensitive paths in under 20 minutes. With the potential application of bddbddb, we make an effort into specifying conditional pushdown system and its *post*∗ problem by Datalog.

For conditional pushdown system (CPDS), one can easily apply Alg. 2 to solve its *post*∗ problem by modifying the saturation convergence condition. That is, in unweighted case, the saturation process will stop when *there are no new transitions introduced to the $\mathcal{P}$-automaton.*

Meanwhile, the specification described below could be extended to use in weighted case where the weight domain is finite; roughly, the extension is possible by integrating a more powerful representation Algebraic Decision Diagrams (ADDs) [3] to bddbddb. Nevertheless, it is beyond the scope of this thesis.

Datalog In deductive databases, Datalog is a query and rule language [34]. Syntactically, it is a restricted form of the programming language Prolog. In terms of query representation, Datalog operates on *relations* (two-dimensional tables) which consists of *attributes* (columns) and a *domain* defining the set of possible attribute values. A Datalog *inference rule* contains two parts: head and subgoals separated by an implication symbol.

$$\underbrace{D(w,z)}_{Head} \underbrace{:-}_{implication} \underbrace{A(w,x), B(x,y), C(y,z)}_{Subgoals}$$

indicates:

- **IF** $A(w,x)$ **AND** $B(x,y)$ **AND** $C(y,z)$ are `true`
- **THEN** $D(w,z)$ is `true`.

A Datalog *program* may contain many relations and inference rules.

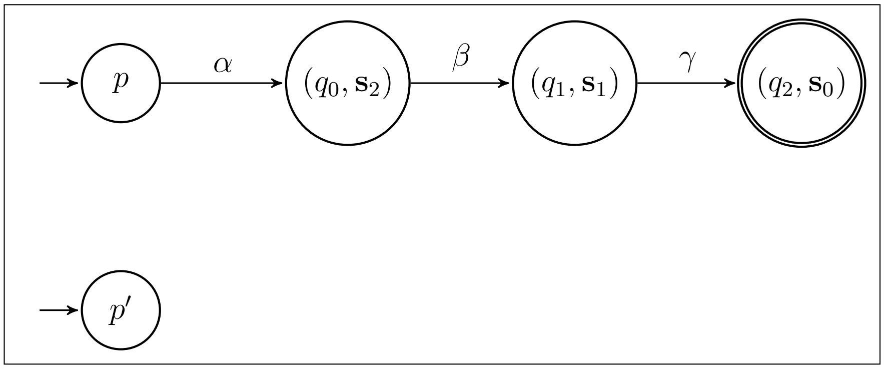
Encoding conditional pushdown system to Datalog program Table 4.1 and Table 4.2 describe the definition of domains and relations accordingly. Fig. 4.6 illustrates how to generate relations for a given $\mathcal{P}$-automaton. Last but not least, Table 4.3 shows all necessary inference rules to solve *post*∗ problem in CPDS. The relation *reachable* should be declared when applying the encoding to a specific analysis.

Declaration	**Remark**
P	control locations
I	internal states of initial $\mathcal{P}$-automata
$Index$	indexes of condition automata
$IndexStates$	indexes of states in condition automata
$A \subseteq Index \times IndexStates$	states of condition automata
$\hat{S}$	states of product automata
Γ	stack alphabet
$\Gamma' = \Gamma \cup \{-\}$	extended stack alphabet with special symbol
$Q \subseteq (P \cup I) \times \Gamma' \times \hat{S}$	internal states of *post** $\mathcal{P}$-automata

Table 4.1: Datalog Domains for On-The-Fly *post*∗ Algorithm in CPDS

Declaration	**Remark**	**Flags**
$push : Index \times P \times \Gamma' \times P \times \Gamma' \times \Gamma'$	Push rules $\langle p, \gamma\rangle \overset{A_i}{\hookrightarrow} \langle p', \gamma'\gamma''\rangle$	input
$pop : Index \times P \times \Gamma' \times P$	Pop rules $\langle p, \gamma\rangle \overset{A_i}{\hookrightarrow} \langle p', \epsilon\rangle$	input
$normal : Index \times P \times \Gamma' \times P \times \Gamma'$	Normal rules $\langle p, \gamma\rangle \overset{A_i}{\hookrightarrow} \langle p', \gamma'\rangle$	input
$cpa1 : P \times \Gamma' \times Q$	Transitions from control locations to internal states in *post** $\mathcal{P}$-automata	input
$cpa2 : Q \times \Gamma' \times Q$	Transitions of internal states to internal states in *post** $\mathcal{P}$-automata	input
$fa : A \times \Gamma' \times A$	Transitions of condition automata	input
$finalFA : A$	Final states of condition automata	input
T $reachable : P$	control locations of interest reachable control locations	input output

Table 4.2: Datalog Relations for On-The-Fly *post*∗ Algorithm in CPDS



Relations describe above $\mathcal{P}$-automaton
$cpa1(p, \alpha, q_0, -, (\mathbf{s}_2)).$ $cpa2(q_0, -, (\mathbf{s}_2), \beta, q_1, -, (\mathbf{s}_1)).$ $cpa2(q_1, -, (\mathbf{s}_1), \gamma, q_2, -, (\mathbf{s}_0)).$

Figure 4.6: A Sample Initial $\mathcal{P}$-automaton Defined by Datalog

$cpa1(p', \varepsilon, q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n}))$	:-	$cpa1(p, \gamma, q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n})),$ $pop(k, p, \gamma, p'),$ $finalFA(s_k^{j_k}).$
$cpa1(p', \gamma', q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n}))$	:-	$cpa1(p, \gamma, q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n})),$ $normal(k, p, \gamma, p', \gamma'),$ $finalFA(s_k^{j_k}).$
$cpa1(p', \gamma', p', \gamma', (s_1^{t_1}, \ldots, s_k^{t_k}, \ldots, s_n^{t_n}))$	:-	$cpa1(p, \gamma, q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n})),$ $push(k, p, \gamma, p', \gamma', \gamma''),$ $finalFA(s_k^{j_k}),$ $fa(s_1^{j_1}, \gamma'', s_1^{t_1}),$ $\ldots,$ $fa(s_k^{j_k}, \gamma'', s_k^{t_k}),$ $\ldots,$ $fa(s_n^{j_n}, \gamma'', s_n^{t_n}).$
$cpa2(p', \gamma', (s_1^{t_1}, \ldots, s_k^{t_k}, \ldots, s_n^{t_n}), \gamma'',$ $q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n}))$	:-	$cpa1(p, \gamma, q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n})),$ $push(k, p, \gamma, p', \gamma', \gamma''),$ $finalFA(s_k^{j_k}),$ $fa(s_1^{j_1}, \gamma'', s_1^{t_1}),$ $\ldots,$ $fa(s_k^{j_k}, \gamma'', s_k^{t_k}),$ $\ldots,$ $fa(s_n^{j_n}, \gamma'', s_n^{t_n}).$
$cpa1(p'', \gamma'', q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n}))$	:-	$cpa1(p, \gamma, q, \alpha, (s_1^{j_1}, \ldots, s_k^{j_k}, \ldots, s_n^{j_n})),$ $push(k, p, \gamma, p', \gamma', \gamma''),$ $finalFA(s_k^{j_k}),$ $cpa1(p'', \varepsilon, p', \gamma', (s_1^{t_1}, \ldots, s_k^{t_k}, \ldots, s_n^{t_n})),$ $fa(s_1^{j_1}, \gamma'', s_1^{t_1}),$ $\ldots,$ $fa(s_k^{j_k}, \gamma'', s_k^{t_k}),$ $\ldots,$ $fa(s_n^{j_n}, \gamma'', s_n^{t_n}).$
$cpa1(p, \gamma', q', \beta, (s_1^{t_1}, \ldots, s_k^{t_k}, \ldots, s_n^{t_n}))$	:-	$cpa1(p, \varepsilon, q, \alpha, (s_1^{j_1}, \ldots, s_n^{j_n})),$ $cpa2(q, \alpha, (s_1^{j_1}, \ldots, s_n^{j_n}), \gamma', q', \beta, (s_1^{t_1}, \ldots, s_n^{t_n})).$

Table 4.3: Datalog Inference Rules for On-The-Fly *post*$*$ Algorithm in CPDS

4.3 Performance of On-The-Fly Algorithm

Implementation of package cwpds Since the target usage is in Java programs analysis, we implement the whole framework and all related algorithms by Java language to maintain the consistency and highest possible efficiency. Our package cwpds offered for conditional weighted pushdown system is extended from the weighted pushdown system library jMoped [40]. In practice, the conditions associated with transition rules are typically defined by regular expressions. To get a fast manipulation and easy convert from regular expression to deterministic automata, the Java automaton library developed by Anders Moller [29] is our choice.

Experiment setup Comparison between the offline algorithm (Fig. 3.4) and the on-the-fly algorithm (Algo. 2) is conducted with test cases obtained from a study of HTML5 parser [28] in which conditional pushdown systems are constructed to formalize a non-trivial phase of the HTML5 specification.

Details of the input conditional pushdown systems are as follows:

- Initial configuration $\langle 323, X \rangle$.
- Size of stack alphabet $|\Gamma| = 25$.
- Number of control locations $|Q| = 487$.
- Number of pushdown transition rules $|\Delta| = 19679$.
- Each condition L_i in the system is described by a regular expression, e.g., `(~(((Div|(Optgroup|(Option|(P|(Rp|Ruby))))))*Li(@)))`. The corresponding condition DFA A_i may contain upto 6 states.

The result reported in Table 4.4 are from executing our implementation on a system with Intel®Core(TM) 2 Duo CPU P8400 @2.26GHz, 2.26GHz, Windows XP SP3 with 1GB RAM and Java Development Kit (JDK) 1.6.

- The first column shows number of conditions in the conditional pushdown system in increased order. As a recall, number of conditions is an aforementioned factor that causes the explosion of extended stack alphabet.
- The second column informs size of the outcome $\mathcal{P}$-automaton after applying *post** algorithm.
- Runtime and memory consumption are estimated in second (s) and Kilobyte (KB).
- ∞ means either taking longer than 24 hours or getting Out-of-memory.

Result When number of conditions $|\mathcal{C}|$ is upto 5, the offline algorithm is still able to handle stack-alphabet explosion, but the resource and overhead cost taken is at least two times more than that of the on-the-fly algorithm. When number of conditions $|\mathcal{C}|$ is over 5, theoretically product of condition DFA could explode exponentially as much as $6^{|\mathcal{C}|}$ in size, where 6 indicates the maximum number of states in a single condition DFA. Table 4.4 proves efficiency of the on-the-fly algorithm towards performance.

Number of conditions	**Size of output**	**Offline algorithm**		**On-the-fly algorithm**	
		Runtime	***Memory***	***Runtime***	***Memory***
1	18 229 transitions	1.313s	26 302 KB	0.953s	16 520 KB
2	31 414 transitions	2.187s	45 810 KB	1s	21 804 KB
5	31 412 transitions	25.438s	671 699 KB	1.047s	17 156 KB
10	109 960 transitions	∞	∞	1.875s	31 577 KB
15	211 368 transitions	∞	∞	2.391s	45 506 KB
20	418 363 transitions	∞	∞	5.11s	79 124 KB
60	3 445 895 transitions	∞	∞	32.594s	610 628 KB
70	3 244 153 transitions	∞	∞	32.062s	562 998 KB
80	3 331 394 transitions	∞	∞	35.094s	564 417 KB

Table 4.4: Runtime and Memory Consumption for Offline and On-The-Fly Algorithms

Chapter 5

Permission Analysis

This chapter is set out to draw attention to another major contribution of our study: How we apply conditional weighted pushdown system to settle a specific problem of stack-based access control in Java 2 platform. First, Section 5.1 clarifies the problem to be addressed and challenges that should be carried out to obtain a precise analysis. Then, the rest sections present our approach in detail.

5.1 Problem Statement and Challenges

The problem to be addressed by our permission analysis is described in Fig. 5.1. The original goal of permission analysis is to improve the runtime performance of Java Virtual Machine (JVM). By knowing which permission checkpoints always pass the stack inspection, one can either remove redundant checks in the program or refine the policy file accordingly.

Given:
(1) A Java program to be analyzed.
(2) A policy file stored at a static location.
(3) A list of trusted protection domains selected among all protection domains mentioned in the policy file.

Problem: Determine whether all permission checks invoked in trusted domains will always succeed. The output will be either ”Always succeed” or ”May fail”.

Figure 5.1: The Problem to be Addressed by Permission Analysis

Example 5.1.1. *Suppose an application* MainUI *gets from program arguments a file name* fn *(without extension) located in directory* home*. It counts number of available bytes in the file* fn.txt*, then write the result to an output file* fn.out*.*

```
   class A {
01:   public static String DIR = "home" + File.pathSeparator;
02:   public void doYourThing(String fileName) {
03:     try {
04:       // read access to file
05:       FileInputStream f = new FileInputStream(DIR+fileName+".txt");
06:       int cnt = f.available();
07:       // write access to directory
08:       FilePermission fp = new FilePermission(DIR+"*", "write");
09:       AccessController.checkPermission(fp);
10:       // write access to file
11:       PrintWriter out = new PrintWriter(DIR+fileName+".out");
12:       out.println("Total available bytes = " + cnt);
13:     } catch (AccessControlException e) {
14:         e.printStackTrace();
15:     } catch (FileNotFoundException e) {
16:         e.printStackTrace();
17:     } catch (IOException e) {
18:         e.printStackTrace();
        }
      }
    }
    public class MainUI {
19:   public static void main(String[] args) {
20:     new A().doYourThing(args[1]);
      }
    }
```

Figure 5.2: A Code Fragment that Involves Tricky Cases of Permission Check

Herein, challenges to be encountered while performing the permission analysis are investigated through Example 5.1.1.

Complicated context of constructing Permission objects From the perspective of developers, Java 2 platform has successfully encapsulated obscure parts of authorization test in its API; this helps release concerns over trivial permission requirements, e.g., asking for read access when you want to open a file. From the perspective of static analyses, such features make the circumstance more sophisticated. Because, without knowledge of how AccessController.checkPermission(perm) is actually invoked and how its parameter perm is instantiated, the analysis result may become overly conservative. To illustrate a tricky context of Permission object construction, suppose one needs to open a text file a.txt by instantiating an object FileInputStream("a.txt") (Line 05 in Fig. 5.2).

The constructor of FileInputStream will invoke the SecurityManager.checkRead() method, then a FilePermission("a.txt", "read") object named perm is instantiated and passed to SecurityManager.checkPermission(perm) method, which subsequently uses the parameter perm to call AccessController.checkPermission(perm).

That means, in a Java program, there may be more than one single access-control checkpoint (explicit invocation of the AccessController.checkPermission method); other checkpoints are perhaps wrapped in system calls. Therefore, not only the program needs to be modeled but also the underlying Java system libraries. Without careful treatment, size of the model to analyze could grow up indiscriminately.

Computation for permission requirements Beside the aforementioned context of Permission object, another problem is that, AccessController.checkPermission(perm) is a static method. Particularly, its invocations may occur in different parts of the program with the same allocation site and the same type of permission (Line 05, 08, 11, 20 in Fig. 5.2). A natural question thereby arises: How to distinguish calling contexts of AccessController.checkPermission and instantiating contexts of Permission object?

Propagation of string constants Every Permission object is made of zero or more parameters of String objects. As mentioned in Chapter 2, permission requirement represented by a Permission object perm can be uniquely identified via the Permission type along with its String parameters as below.

```
perm = new PermissionType(target, action)
```

The String expressions describe target and actions may be created and passed through a sequence of manipulation, e.g., concatenation (Line 05, 08, 11 in Fig. 5.2). Hence, for analyses aiming at Java 2 access control, it is crucial to keep track of string constants.

The remainder of this chapter sequentially reveals prerequisites and the way we cope with above challenges. From now on, checkPermission will stand for AccessController.checkPermission.

5.2 Prerequisites

In our analysis, we have assumption of the following context-sensitive *points-to analysis* [2] and *string analysis* [10]. These two extra analyses are responsible for computing type of perm at every invocation checkPermission(perm) along with (zero or more) String parameters when instantiating the perm object:

- Given a reference v, $pta(v)$ returns the finite set of abstract objects that v may refer to at runtime under certain calling contexts. Each element in $pta(v)$ is represented as a triplet $(type, loc, c)$, where $type$ is the object type, loc is the allocation site, and c is the calling contexts through which the object is constructed.

- Given a string variable v, $sa(v)$ 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 is the calling contexts through which sv is constructed.

The precision of our analysis depends on the precision of points-to analysis and string analysis.

5.3 Abstraction of Java Programs

Definition 5.3.1. *Let $Methods$ denote the set of all methods in the given program. We denote by $G_m = (N_m, E_m, e_m, x_m)$ the control flow graph (CFG) for any method $m \in Methods$, where N_m is the set of nodes that corresponds to program points, and $E_m \subseteq N_m \times N_m$ is the set of edges that represents the program control flow, and $e_m, x_m \in N_m$ are unique entry point and exit point of m, representatively.*

For the method AccessController.checkPermission that triggers stack inspection, we define its CFG as $G_{cp} = (\{e_{\mathbf{cp}}, x_{\mathbf{cp}}\}, \{(e_{\mathbf{cp}}, x_{\mathbf{cp}})\}, e_{\mathbf{cp}}, x_{\mathbf{cp}})$.

An inter-procedural CFG $G = (N, E)$ is a directed graph where N is the set of nodes and $E \subseteq N \times N$ is the set of edges. Given a collection of CFGs for all methods, G is built as follows. Initially $N = \bigcup_{m \in Methods} N_m$ and $E = \bigcup_{m \in Methods} E_m$. For any $n \in N$, if n is an invocation point for "m calls m'", then

- *a call edge $(n, e_{m'})$ and a return edge $(x_{m'}, n^R)$ are added into E, where $n^R \in N$ is a fresh node representing the return point, and*
- *new edges $\{(n^R, n') \mid (n, n') \in E\}$ are added into E, and the original edges $\{(n, n') \in E \mid n' \in N\}$ are removed from E.*

Note that,

- *Since our permission analysis is concerned with method invocations, nodes and edges in G that are not relevant to method invocations and the control flow structure of the program, such as program branching points, are abstracted away.*
- *Treatment for AccessController.doPrivileged: According to Java 2 semantics, each invocation of AccessController.doPrivileged(action) involves an underlying call to action.run(). To abstract such relation, one edge should be added between the entry point of AccessController.doPrivileged and that of the corresponding action.run().*

Definition 5.3.2. *Given an inter-procedural CFG $G = (N, E)$ of the given program, we define a pushdown system $P = (\{\cdot\}, \Gamma, \Delta, \gamma_{initial})$ that models the program, where the set of control locations is singleton, $\Gamma \subseteq N$, $\gamma_{initial} = e_m$ with m being the program's entry point, and Δ is constructed as follows, for each edge $(n, n') \in E$,*

- $\langle \cdot, n \rangle \hookrightarrow \langle \cdot, n' n^R \rangle \in \Delta$, *if $n' = e_m$ for some method m;*
- $\langle \cdot, n \rangle \hookrightarrow \langle \cdot, \epsilon \rangle \in \Delta$, *if $n = x_m$ for some method m and $n' = u^R$ is the return point of node u;*
- $\langle \cdot, n \rangle \hookrightarrow \langle \cdot, n' \rangle \in \Delta$ *otherwise.*

5.4 Abstraction of Policy System

The second analysis step is to abstract the domain-specific security policy of Java 2 platform. Recall that, all classes in a protection domain are granted the same set of permissions. Consequently, we can reasonably suppose all methods in the same class are also granted the same set of permissions.

Definition 5.4.1. *Let Domain denote a finite set of protection domains, and Perms denote the universe of all permissions involved in the given program.*

- $\texttt{dom} : Methods \longrightarrow Domain$ *is a mapping from methods to protection domains. This mapping is proper for Java semantics where domains are assigned to classes and all methods in classes.*

- *Each control flow graph is associated with a* security policy $\texttt{perm} : Domain \longrightarrow 2^{Perms}$*, which grants a set of permissions to each protection domain. Especially, all methods in system domain, e.g., method AccessController.doPrivileged, are granted all permissions in Perms.*

- *The policy system in our approach is method-wise; so all entry, exit, and invocation points associated with a method m will inherit all permissions granted to m and also belong to the protection domain of m.*

Consider `perm` is extended element-wise. Hereafter, we use $\texttt{policy}(m)$ as the abbreviation for $(\texttt{perm} \circ \texttt{dom})(m)$.

Definition 5.4.2. *Let $CheckPoints = \{n \in N \mid (n, n') \in E, n' = e_{cp}\}$, i.e., CheckPoints is the set of call sites that invokes the AccessController.checkPermission method.*

For each checkpoint $n \in CheckPoints$ that is supposed to contain expression "`checkPermission(perm)`", we first call points-to analysis $pta(\texttt{perm})$. For each $(\texttt{Type}, \texttt{loc}, c) \in pta(\texttt{perm})$, the allocation site referred to by l is supposed to contain expressions in one of the following form

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

We add each new permission $perm$ to $Perms$, where $perm =$

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

5.5 Permission Analysis

Our approach consists of two steps:

1. **Determining analysis points**: within programs of a given trusted domain, all trigger points of checkPermission will be captured and named *AnalysisPoints*.

2. **Performing permission analysis**: an analysis will be performed to decide whether all permission checks invoked in the trusted domain always succeed.

5.5.1 Determining Analysis Points

Definition 5.5.1. *Given an inter-procedural CFG $G = (N, E)$ (Def. 5.3.1), and a mapping $l : N \to Domain$ from nodes to their belonging protection domains. A **boundary** of a given domain $dm \in Domain$, denoted by $\mathcal{B}(dm)$, is defined as*

$$\mathcal{B}(dm) = \{u \in N \mid (u, v) \in E, l(u) = dm, l(u) \neq l(v)\}$$

The boundary of a domain dm refers to nodes from dm with outgoing edges to nodes from different domains (that are typically Java libraries).

The first step of our analysis is to determine program points from the given domain dm, denoted by $AnalysisPoints(dm)$, that may (indirectly) trigger stack inspection at runtime and will be examined in permission analysis. Assume the pushdown system encoded by Def. 5.3.2. Such checkpoints are defined as an approximation by

$$AnalysisPoints(dm) = \{n \in N \mid \langle \cdot, n\omega \rangle \in R \text{ for some } \omega \in \Gamma^*\}$$

where $R = \{\langle \cdot, n\omega \rangle \mid \omega \in \Gamma^*, n \in \mathcal{B}(dm)\} \cap pre^*(\{\langle \cdot, e_{\mathsf{cp}}\omega \rangle \mid \omega \in \Gamma^*\})$.

5.5.2 Permission Check

Definition 5.5.2. *Given a domain $dm \in Domain$, for each analysis point $n \in AnalysisPoints(dm)$, we denote by $PermReqs(n)$ permissions that may be checked for stack inspection at runtime triggered by the statement at n.*

We leave the realization of Def. 5.5.2 as future work. We adopt the semiring $\mathcal{S}_{perm} = (D_{perm}, \oplus_{perm}, \otimes_{perm}, \bar{0}, \bar{1})$ in [24], given a PER-based abstraction with 2-point domain $\{ANY, ID\}$, where $D_{perm} = \{\lambda x.ANY, \lambda x.ID, \bar{0}, \bar{1}\}$ with the ordering $\lambda x.ANY \sqsubseteq \bar{1} \sqsubseteq \lambda x.ID \sqsubseteq \bar{0}$.

We adapt the pushdown system $P = (\{\cdot\}, \Gamma, \Delta, \gamma_{initial})$ encoded from the given program (Def. 3.1.1) to a CWPDS $W_{perm} = (P_{perm}, \mathcal{S}_{perm}, f_{perm})$, where $P_{perm} = (\{\cdot\}, \Gamma, C_{perm}, \Delta_{perm}, \gamma_{initial})$.

We write $\langle p, \alpha \rangle \overset{C,w}{\hookrightarrow} \langle q, \omega \rangle$ if $r : (p, \alpha, C, q, \omega) \in \Delta_c$ and $f(r) = w$. For each $r : \langle \cdot, \alpha \rangle \hookrightarrow \langle \cdot, \omega \rangle \in \Delta$, if $\alpha = e_{\mathsf{cp}}$ and $\omega = x_{\mathsf{cp}}$, then for each $perm \in Perms$, we have

$$\begin{aligned} \langle \cdot, e_{\mathsf{cp}} \rangle &\overset{L\&C,\bar{1}}{\hookrightarrow} \langle \cdot, x_{\mathsf{cp}} \rangle \in \Delta_{perm}, \text{ and} \\ \langle \cdot, e_{\mathsf{cp}} \rangle &\overset{(!L)\&C,\lambda x.ANY}{\hookrightarrow} \langle \cdot, x_{\mathsf{cp}} \rangle \in \Delta_{perm} \end{aligned}$$

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

- $\alpha = \{(l, m) \in \Gamma \mid perm \in \texttt{policy}(m)\}$,
- $\beta = \alpha \cap \{(l, m) \in \Gamma \mid m = \textsf{AccessController.doPrivileged}\}$.
- $C = \Gamma^*(n_1^R + n_2^R + \cdots + n_k^R)\Gamma^*$ where $\{n_1^R, n_2^R, \ldots, n_k^R\} = \{n^R \mid perm \in PermReqs(n)\}$

Definition 5.5.3. *Given a domain $dm \in Domain$. For any analysis point $n \in AnalysisPoints(dm)$, we say stack inspections invoked by n may fail if $MOVP(S, T, W_{perm}) = \lambda x.ANY$ and always succeed otherwise, where $S = \{\langle \cdot, \gamma_{initial} \rangle\}$, $T = \{\langle \cdot, n^R\omega \mid \omega \in \Gamma^*\}$. We say access control invoked from the domain dm always succeed if each analysis point in this domain always succeeds for stack inspection, and may fail otherwise.*

Example 5.5.4 illustrates the control flow graph and conditional weighted pushdown system in case there are two different invocations of checkPermission.

Example 5.5.4. *Consider a code fragment (Fig. 5.3) and its control flow graph (Fig. 5.4).*

```
    package test.App;
    public class MyApp() {
l0:   public static void main(String[] args) {
l1:     boolean canOpenFile = false;
l2:     checkPermission(new FilePermission("a.txt", "read"));
l3:     canOpenFile = true;
l4:     A(canOpenFile);
l5:     checkPermission(new FilePermission("b.jar", "execute"));
    }
```

Figure 5.3: A Program Fragment with Permission Checks for File Access

The encoding to conditional weighted pushdown system is:

- *Stack alphabet:*

$$\Gamma = \{e_{main}, x_{main}, e_{cp}, x_{cp}, n_1, n_1^R, n_2, n_2^R, n_3, n_3^R\}$$

where $n_1 = (l2, main)$, $n_2 = (l4, main)$, $n_3 = (l5, main)$.

- *Pushdown transition rules:*

$$\Delta = \left\{ \begin{array}{lll}
\langle \cdot, x_{main} \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, \epsilon \rangle \\
\langle \cdot, x_{\boldsymbol{cp}} \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, \epsilon \rangle \\
\langle \cdot, x_A \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, \epsilon \rangle \\
\langle \cdot, e_{main} \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, n_1 \rangle \\
\langle \cdot, n_1 \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, e_{\boldsymbol{cp}}\, n_1^R \rangle \\
\langle \cdot, e_{\boldsymbol{cp}} \rangle & \overset{L_1 \& C_1,\bar{1}}{\hookrightarrow} & \langle \cdot, x_{\boldsymbol{cp}} \rangle \\
\langle \cdot, e_{\boldsymbol{cp}} \rangle & \overset{(!L_1) \& C_1,\lambda x.ANY}{\hookrightarrow} & \langle \cdot, x_{\boldsymbol{cp}} \rangle \\
\langle \cdot, e_{\boldsymbol{cp}} \rangle & \overset{L_2 \& C_2,\bar{1}}{\hookrightarrow} & \langle \cdot, x_{\boldsymbol{cp}} \rangle \\
\langle \cdot, e_{\boldsymbol{cp}} \rangle & \overset{(!L_2) \& C_2,\lambda x.ANY}{\hookrightarrow} & \langle \cdot, x_{\boldsymbol{cp}} \rangle \\
\langle \cdot, n_1^R \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, n_2 \rangle \\
\langle \cdot, n_2 \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, e_A\, n_2^R \rangle \\
\langle \cdot, e_A \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, x_A \rangle \\
\langle \cdot, n_2^R \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, n_3 \rangle \\
\langle \cdot, n_3 \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, e_{cp}\, n_3^R \rangle \\
\langle \cdot, n_3^R \rangle & \overset{\Gamma^*,\bar{1}}{\hookrightarrow} & \langle \cdot, x_{main} \rangle
\end{array} \right\}$$

  where:

  - $L_1 = (\alpha_1^*) + (\alpha_1^*)\beta_1\alpha_1(\Gamma^*)$,
  - $\alpha_1 = \{\gamma = (l, m) \in \Gamma \mid FilePermission(”a.txt”, ”read”) \in \boldsymbol{\mathtt{policy}}(m)\}$,
  - $\beta_1 = \alpha_1 \cap \{\gamma = (l, m) \in \Gamma \mid m = \textsf{\textit{AccessController.doPrivileged}}\}$,
  - $C_1 = \Gamma^*(n_1^R)\Gamma^*$,
  - $L_2 = (\alpha_2^*) + (\alpha_2^*)\beta_2\alpha_2(\Gamma^*)$,
  - $\alpha_2 = \{\gamma = (l, m) \in \Gamma \mid FilePermission(”b.jar”, ”execute”) \in \boldsymbol{\mathtt{policy}}(m)\}$,
  - $\beta_2 = \alpha_2 \cap \{\gamma = (l, m) \in \Gamma \mid m = \textsf{\textit{AccessController.doPrivileged}}\}$,
  - $C_2 = \Gamma^*(n_3^R)\Gamma^*$.

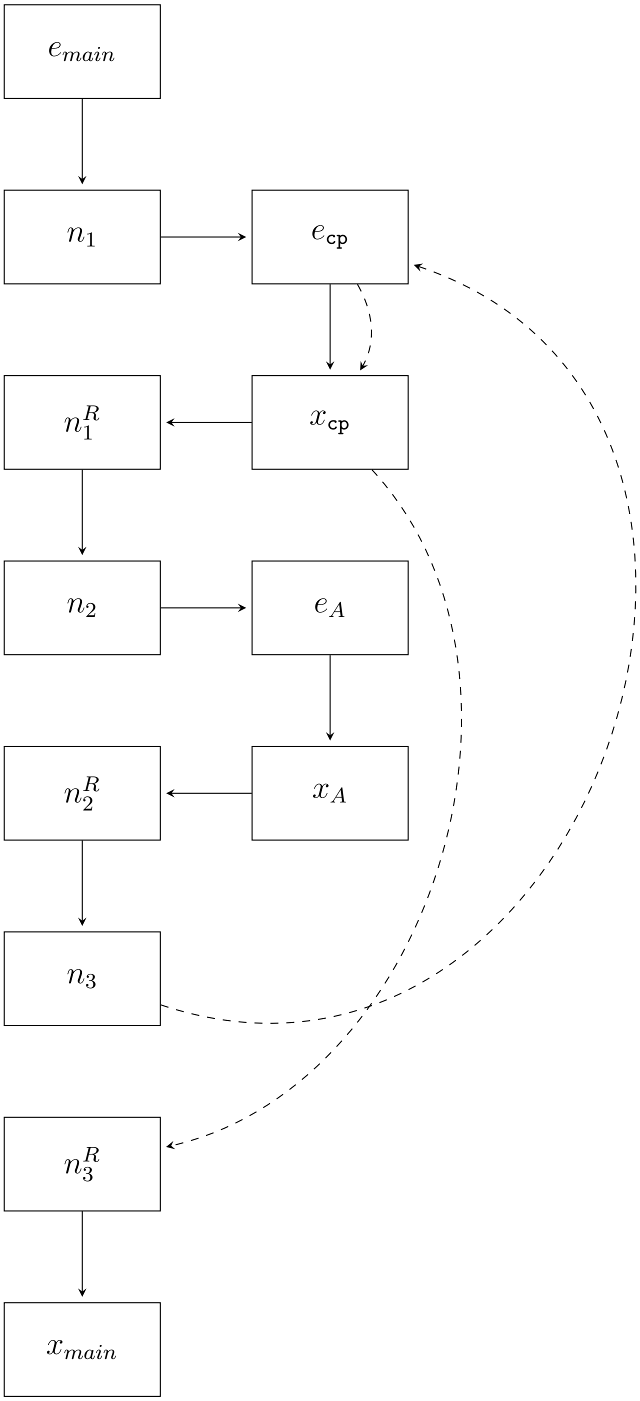


Figure 5.4: The Control Flow Graph of Program Fragment in Fig. 5.3.

5.6 Demonstration of Permission Analysis Framework

5.6.1 Sample Usage of Package cwpds for Permission Check

As described in Chapter 5, conditional weighted pushdown system is the key formalism in our permission analysis framework. In more detail, the on-the-fly algorithm in package cwpds will be used to compute *post** $\mathcal{P}$-automaton for the permission check. Here we demonstrate how to use package cwpds for such purpose via the Example 5.5.4; full source code of this demonstration could be found in Appendix A.

Suppose conditions of pushdown transition rules shown in Example 5.5.4 are:

- $L_1 = (\alpha_1^*) + (\alpha_1^*)\beta_1\alpha_1(\Gamma^*)$, where $\alpha_1 = \{e_{main}, x_{main}, e_{\mathsf{cp}}, x_{\mathsf{cp}}, n_1, n_1^R, n_2, n_2^R, n_3, n_3^R\}$, and $\beta_1 = \emptyset$,
- $C_1 = \Gamma^*(n_1^R)\Gamma^*$,
- $L_2 = (\alpha_2^*) + (\alpha_2^*)\beta_2\alpha_2(\Gamma^*)$, where $\alpha_2 = \{e_A, x_A, e_{\mathsf{cp}}, x_{\mathsf{cp}}\}$, and $\beta_2 = \emptyset$,
- $C_2 = \Gamma^*(n_3^R)\Gamma^*$.

Before invoking the on-the-fly algorithm, one should construct a conditional weighted pushdown system in the following way:

(i) Defining regular expressions and conditional DFA (Line 30 – 60 in Fig. A.2). A_0 describes condition Γ^*. Condition DFA A_1, A_2 are corresponding to L_1 and $!L_1$; while A_3, A_4 represent L_2 and $!L_2$.

(ii) Computing Cartersian product $A = \prod_{0 \leq i \leq 4} A_i$ (Line 63 – 68 in Fig. A.2).

(iii) Creating a conditional weighted pushdown system by setting Semiring (PERSemiring), and adding pushdown transition rules (Line 71 – 102 in Fig. A.2, and Fig. A.3).

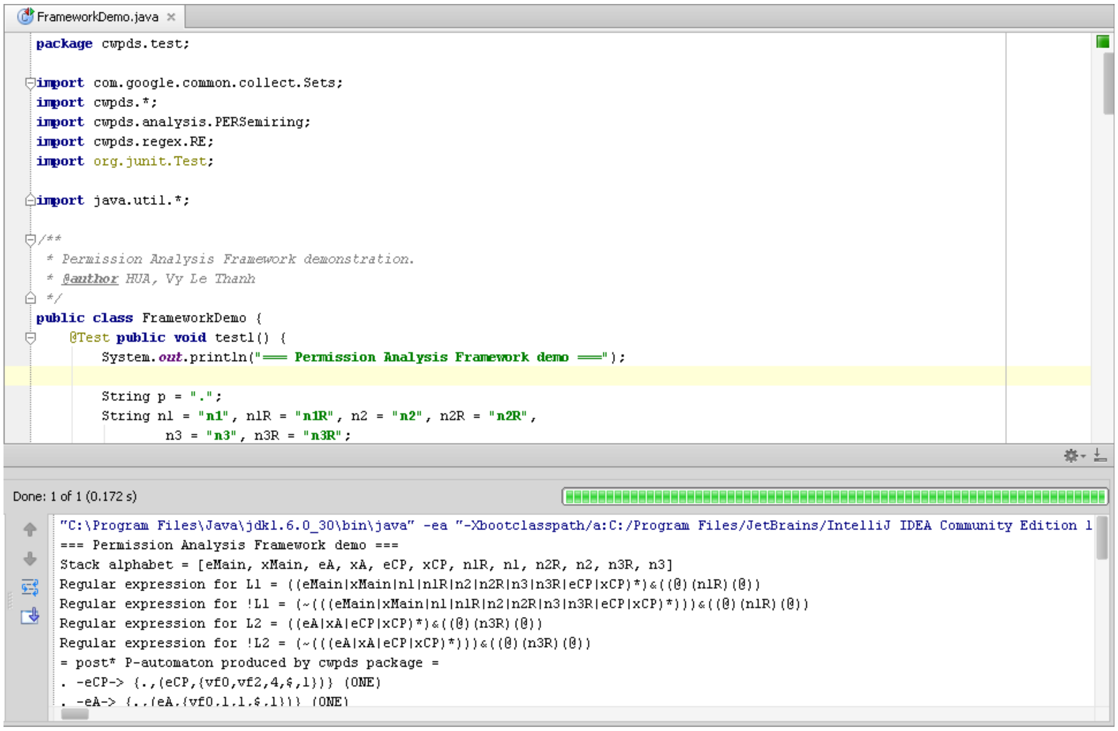
(iv) Initializing $\mathcal{P}$-automaton with a given program entry point, e.g. e_{main} (Line 105 – 109 in Fig. A.3).

(v) Asking the on-the-fly saturation algorithm to obtain *post** $\mathcal{P}$-automata (Line 113 – 118 in Fig. A.3).

(vi) Finally, stack inspections invoked by a checkpoint n is determined by the analysis result or solution of $MOVP$ (Def. 5.5.3) at n. This value can be computed by Algorithm 4 Fig. 19 in [36] (Line 119 – 122 in Fig. A.3).

Fig. 5.5 illustrates a successful run of package cwpds. Format of transitions in the outcome $\mathcal{P}$-automata is:

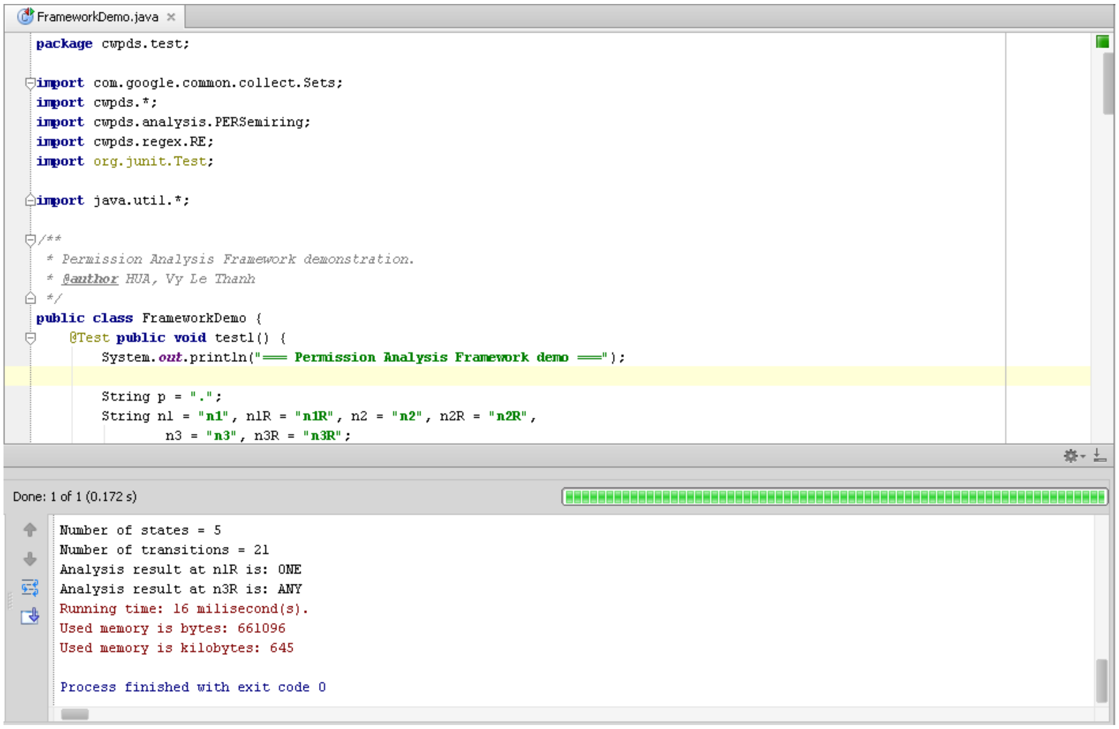
```
<start state> -<symbol>-> <end state> (weight of transition)
```



```java
package cwpds.test;

import com.google.common.collect.Sets;
import cwpds.*;
import cwpds.analysis.PERSemiring;
import cwpds.regex.RE;
import org.junit.Test;

import java.util.*;

/**
 * Permission Analysis Framework demonstration.
 * @author HUA, Vy Le Thanh
 */
public class FrameworkDemo {
    @Test public void test1() {
        System.out.println("=== Permission Analysis Framework demo ===");

        String p = ".";
        String n1 = "n1", n1R = "n1R", n2 = "n2", n2R = "n2R",
                n3 = "n3", n3R = "n3R";
```

```
"C:\Program Files\Java\jdk1.6.0_30\bin\java" -ea "-Xbootclasspath/a:C:/Program Files/JetBrains/IntelliJ IDEA Community Edition 1
=== Permission Analysis Framework demo ===
Stack alphabet = [eMain, xMain, eA, xA, eCP, xCP, n1R, n1, n2R, n2, n3R, n3]
Regular expression for L1 = ((eMain|xMain|n1|n1R|n2|n2R|n3|n3R|eCP|xCP)*)&((@)(n1R)(@))
Regular expression for !L1 = (~(((eMain|xMain|n1|n1R|n2|n2R|n3|n3R|eCP|xCP)*)))&((@)(n1R)(@))
Regular expression for L2 = ((eA|xA|eCP|xCP)*)&((@)(n3R)(@))
Regular expression for !L2 = (~(((eA|xA|eCP|xCP)*)))&((@)(n3R)(@))
= post* P-automaton produced by cwpds package =
. -eCP-> {.,(eCP,{vf0,vf2,4,$,1})} (ONE)
. -eA-> {.,(eA,{vf0,1,1,$,1})} (ONE)
```

Figure 5.5: Screen-shot of Package cwpds Usage in Java



```java
package cwpds.test;

import com.google.common.collect.Sets;
import cwpds.*;
import cwpds.analysis.PERSemiring;
import cwpds.regex.RE;
import org.junit.Test;

import java.util.*;

/**
 * Permission Analysis Framework demonstration.
 * @author HUA, Vy Le Thanh
 */
public class FrameworkDemo {
    @Test public void test1() {
        System.out.println("=== Permission Analysis Framework demo ===");

        String p = ".";
        String n1 = "n1", n1R = "n1R", n2 = "n2", n2R = "n2R",
                n3 = "n3", n3R = "n3R";
```

```
Number of states = 5
Number of transitions = 21
Analysis result at n1R is: ONE
Analysis result at n3R is: ANY
Running time: 16 milisecond(s).
Used memory is bytes: 661096
Used memory is kilobytes: 645

Process finished with exit code 0
```

Figure 5.6: Screen-shot of Analysis Result Using Package cwpds

Fig. 5.6 displays the analysis result at two points n_1^R and n_3^R, along with a running profile of the demonstration (Line 124 – 135 in Fig. A.4). According to above conditional pushdown system, the checkPermission invoked at n_1 always succeeds, while at n_3 it may fail.

5.6.2 Blueprint for Realization of the Framework

Realization of the framework is underway. Our design for abstracting a Java program and constructing control flow graph can be sketched as follows.

Front-end engine Among all strategies to implement program analysis algorithms, the primary concern is ability to construct control flow graph for Java method invocation, tools for points-to analysis, and the possibility to integrate with our back-end libraries. After careful consideration, our choice is Soot – a Java Optimization Framework from McGill University [41]. It not only provides the benefits of being well-established for research projects but also offers well-documented source code [13] [37]. Soot can be used as either a stand-alone tool or embedded instructions in Java programs.

Intermediate representation The versatility of Soot is remarkable since it supplies four intermediate representations for code: Baf, Jimple, Shimple and Grimp. Within the scope of our analysis, Jimple is the most suitable to represent the expected level of abstraction. Jimple (**J**ava's s**imple**) is a stackless, typed 3-address, and statement-based intermediate representation. It breaks compound statements into more atomic ones so that each statement can be described as a 4-tuple: *(result, operand1, operator, operand2)*. For instance, $z = a + b + c$ will become $z0 = a + b$ and $z = z0 + c$. This feature is essential to simplify dataflow analysis [13]. Jimple has 15 kinds of statements. For the purpose of interprocedural control-flow analysis, we utilize only InvokeStmt which represents method invocation.

Control flow graph creation Among several kinds of control flow graphs (CFGs) provided by Soot, BriefUnitGraph sufficiently meets our requirement. From a complete CFG generated by Soot, we simplified the model by abstracting away irrelevant nodes and edges as described in Chapter 5. A technical issue worth noting is treatment for the method AccessController.doPrivileged. Each CFG in Soot is built for a given method body, this construction thereby cannot be applied for a non-concrete method with no body like AccessController.doPrivileged. In other words, whenever encountering an invocation AccessController.doPrivileged(action), we need to perform a points-to analysis to obtain proper object action, then manually create edges for the relationship between AccessController.doPrivileged and action.run() method.

Points-to analysis tool Of concern over large-scale applications, Japot – a stacking-based context-sensitive points-to analysis tool for Java, is reported to be well-scaled with benchmarks of significant size [25].

String analysis tool Java String Analyzer (JSA) [11] is a tool for static string analysis which assesses possible values of string expressions. Its underlying technique was proposed by Christensen et al. [10]. For a gentle manual of installation and usage, we refer to [16]. The first phase in the front-end of JSA is the Jimple code generation implemented by Soot, while the outcome automata are represented using the Anders Moller's package [29]. They conjointly make our structure in harmony.

Identifying permission requirements Each analysis point n will be matched with a set of permissions which may be checked for stack inspection triggered from n. This step could be done by one extra analysis on the calling contexts.

Chapter 6

Related Work

This chapter reports a literature survey on related studies by three aspects: applications of conditional pushdown system, analyses aimed to stack inspection, and other approaches for access control systems.

Applications of conditional pushdown system Minamide et al. [28] suggested an interesting application of reachability analysis for parsing specification of HTML5. To avoid blowup of the size of stack alphabet while translating the problem to conditional pushdown system, the authors extended $\mathcal{P}$-automata to describe a set of configurations to automata with regular lookahead.

Analyses related to stack inspection and available tools Inspecting call stack in runtime can be costly from a performance point of view. Hence, a considerable amount of literature has been published on static analysis for stack inspection.

Namely, Banerjee et al. [4] pointed to a denotational semantics that proved the equivalence of eager and lazy evaluation for stack inspection, proposed a static analysis with full-proof for safety, and also identified program transformations that help to remove redundant runtime access control checks. Meanwhile, [21] proposed a technique using a context-sensitive, flow-sensitive, interprocedural 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 a practical matter, as it supposed parameters to Permission constructors were string constants.

From theoretical side, given a program which may contain stack inspection, the verification problem to decide whether it satisfies a given policy properties, was proved intractable in general [31]. Nonetheless, there still existed a solvable subclass of programs which precisely model programs containing checkPermission of Java 2 platform. Moreover, the study concluded computational complexity of problem for the subclass is linear time in the size of the given program.

On the other hand, in light of practical circumstances, [9] 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 [20]. Controversially, the tool made no attempt to relieve users from the burden of deciding access rights; more specifically, along with a given policy file, users were also required to explicitly specify which methods and permissions to check.

Conversely, being a module of privilege assertion in a popular tool – IBM Security Workbench Development for Java (SWORD4J) [18], the interprocedural analysis for privileged code placement [33] tackled three 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 Access-Rights Invocation Graph (ARIG) [21], the analysis enabled propagation of string constants to acquire more precise constructor of Permission object. Even so, having built on the expensive context-sensitive call graph, ARIG, scalability becomes its major disadvantage.

Of primary concern over implementation of the stack inspection algorithm, two control flow forward analyses, Denied Permission Analysis and Granted Permission Analysis, were defined by Bartoletti et al. [5] [6] to approximate the set of permissions denied or granted to a given Java bytecode at runtime. Outcome of the analyses were then used to eliminate redundant permission checks and relocate others to more proper places in the code.

Briefly, [21] [6] [20] approximated permissions required to allow the code to execute at each checkpoint. While with conditional weighted pushdown systems, we do not need to explicitly compute such information.

To the best of our knowledge, the modular analysis proposed in [17] is the most relevant to our work. The authors employed Automated Authorization Analysis (A3) tool to assess the precision of permission requirements for stack inspection. Strictly speaking, the study managed to identify authorization objects by a combination of string analysis and program slicing. However, it shed light only on analyzing for necessary permissions rather than checking an existing policy. That is to say, the effectiveness of its analysis relied on whether the permissions it detected were actually required or not.

Other approaches for access-control Since the inception of stack-based access control systems, stack inspection is widely adopted as a simple and practical model for Web security. Still a number of inherent flaws have been discovered; for example, an unauthorized code which is no longer in the call stack may be allowed to affect the execution of security-sensitive code [33] [32]. As a consequence, it makes an appeal to attempts for better alternate models or remedies to the known limitations.

Instead of analyzing security policies associated with existing code, Erlingsson and Schneider [14] composed new notion for enforcing security policies on runtime platforms like Java Virtual Machine. They constructed a system that inserted an inlined reference monitor and authorization checkpoints into the code to reduce redundant tests.

Likewise, targeting at a central problem of stack inspection: "To what extent local checks are sufficient to guarantee enforcement of global security properties?" [8] created a constraint-based static program analysis for inferring a secure calling context for a stack-inspecting method. It made a substantial step towards a secure interface for stack inspection [7].

A worth highlighting alternate model for Stack-based access control (SBAC) is the Information-based access control (IBAC) which was recently introduced in [32]. IBAC brought out a novel security model that could verify all and only the code responsible for a security-sensitive operation was sufficiently authorized. Notwithstanding it surpassed SBAC in many case studies, IBAC was quite complex for a complete implementation and applications.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

With a given program and its security policy, our framework performs a static analysis to analyze whether all permission checks in concerned domains always succeed. The analysis result indeed embraces the original motivation of program analysis; that is, to improve program performance with results already known at compile time. Specifically, with the awareness of always-succeed permission checks, developers can eliminate superfluous computational cost at runtime by removing those redundant access-control checks or further cleaning up bulky parts of the security policy.

We build up the framework by bringing two key factors into focus: the efficiency of framework approach and the precision of core analyses. Of concern over framework approach, we propose an on-the-fly algorithm to solve the forward reachability problem in conditional weighted pushdown system. In contrast to existing approach, experiment described in Chapter 4 Section 4.3 shows potential efficiency of our algorithm. Implementation of on-the-fly algorithm is packed in our open-source package cwpds. Furthermore, we pave the way for possible large-scale applications by providing a Datalog specification of the algorithm in scope of conditional pushdown system.

The calling context of a permission check and its relevant parameters are difficult to evaluate at compile time, in general; it requires meticulous analysis to deal with various situations of runtime data. By designing the framework in conditional weighted pushdown systems, we can manage the calling contexts in a precise way. Furthermore, it leads to an automatic realization of the analysis algorithm by model checker. As described in Chapter 5, we divide the problem into two analyses for: determination of analysis points and permission check. The first analysis is undertaken by applying weighted pushdown system, whereas the latter takes advantage of conditional weighted pushdown system with the proposed on-the-fly algorithm.

7.2 Future Work

Exploring the following matters promises to be a worthwhile line in future work:

- *Completing realization of the permission analysis framework.* As the framework turns out to be a helpful tool for Web security, topmost priority is putting it into practice and conducting empirical study on performance.

- *Investigating more efficient algorithms for problem instances in practice.* Despite of the potential efficiency of the on-the-fly algorithm, it is still appealed for an improvement so that the adaptation of conditional pushdown system are enabled in large-scale applications.

- *Investigating case studies for Datalog encoding.* With an implementation in Binary Decision Diagrams, e.g. bddbddb, the Datalog encoding appears promising once it benefits are taken for problem themes in which number of attributes is sufficiently solvable.

- *Extending scenario for the automation of authorization test.* An interesting foresight of our framework analysis result is that one could make use of it to generate a policy file. Composing a maintainable policy file is typically a manual work in reality. It requires domain-specific knowledge and hands-on experience in security, especially when one composes from scratch. Therefore, an initial policy file would help reduce such kind of burden.

Bibliography

[1] H. Andersen. An introduction to binary decision diagrams. In *Lecture Notes, http://www.cs.auc.dk/˜kgl/VERIFICATION99/mm4.html*, 1997.

[2] L. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Copenhagen, 1994.

[3] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their Applications. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 188–191, Los Alamitos, CA, USA, 1993. IEEE Comput. Soc. Press.

[4] A. Banerjee and D. A. Naumann. A simple semantics and static analysis for Java security. Technical report, Stevens Institute of Technology, 2001.

[5] M. Bartoletti and P. Degano. Static analysis for stack inspection. *Electronic Notes in Theoretical Computer Science*, 54:706–80, Aug. 2001.

[6] M. Bartoletti and P. Degano. Stack inspection and secure program transformations. *International Journal of Information*, 2004.

[7] F. Besson. Interfaces for stack inspection. *Journal of Functional*, 2005.

[8] F. Besson, T. de Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 76–87. ACM, 2002.

[9] B. Chang. Static check analysis for Java stack inspection. *ACM SIGPLAN Notices*, 41(3):40, Mar. 2006.

[10] A. S. Christensen and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the 10th international conference on Static analysis*, number February, pages 1–18, San Diego, CA, USA, 2003. Springer-Verlag.

[11] A. U. Department of Computer Science. Java String Analyzer. http://www.brics.dk/JSA/, 2003.

[12] G. Egan, K. Haley, D. Mckinney, T. Millington, J. Mulcahy, T. Parsons, A. Watson, M. Nisbet, N. Johnston, and S. Hittel. Internet Security Threat Report 2011. Technical Report April, Symantec, 2012.

[13] A. Einarsson and J. D. Nielsen. A survivor's guide to Java program analysis with soot. Technical report, McGill University, 2008.

[14] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, number May, pages 1–27, 2000.

[15] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, volume 1, pages 232–247. Springer, 2000.

[16] A. Feldthaus and A. Moller. The Big Manual for the Java String Analyzer. Technical report, Department of Computer Science, Aarhus University, 2009.

[17] E. Geay, M. Pistoia, B. G. Ryder, and J. Dolby. Modular string-sensitive permission analysis with demand-driven precision. *2009 IEEE 31st International Conference on Software Engineering*, pages 177–187, 2009.

[18] T. Habeck, L. Koved, M. Pistoia, and Y. Heights. SWORD4J : Security WORkbench Development environment 4 Java. Technical report, IBM, 2008.

[19] S. Jha, A. Datta, N. Li, D. Melski, and T. Reps. *Analysis Techniques for Information Security.* Morgan and Claypool Publishers, 2010.

[20] Y. Kim. Visualization of permission checks in java using static analysis. *Information Security Applications*, pages 133–146, 2007.

[21] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 37, pages 359—-372. ACM, Nov. 2002.

[22] G. Li. *Inside Java 2 platform security architecture, API design, and implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[23] G. Li and M. Mueller. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, number December, Monterey, California, 1997.

[24] X. Li and M. Ogawa. Interprocedural program analysis for Java based on weighted pushdown model checking. In *The 5th International Workshop on Automated Verification of Infinite-State Systems (AVIS 2006), ETAPS (April 2006)*, pages 1–13, 2006.

[25] X. Li and M. Ogawa. Stacking-based context-sensitive points-to analysis for Java. *Hardware and Software: Verification and Testing*, pages 133–149, 2009.

[26] X. Li and M. Ogawa. Conditional weighted pushdown systems and applications. *Proceedings of the ACM SIGPLAN 2010 workshop on Partial evaluation and program manipulation - PEPM '10*, page 141, 2010.

[27] G. McGraw and E. Felten. *Securing Java: Getting Down to Business with Mobile Code*. Wiley, 2nd edition, 1999.

[28] Y. Minamide and S. Mori. Reachability Analysis of the HTML5 Parser Specification and its Application to Compatibility Testing. In *18th International symposium on formal methods*, 2012.

[29] A. Moller. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java, 2010.

[30] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[31] N. Nitta and Y. Takata. An efficient security verification method for programs with stack inspection. *Computer and Communications Security*, pages 68–77, 2001.

[32] M. Pistoia, A. Banerjee, and D. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. *Security and Privacy, 2007*, 2007.

[33] M. Pistoia, R. Flynn, and L. Koved. Interprocedural analysis for privileged code placement and tainted variable detection. *ECOOP 2005-Object-Oriented*, pages 362–386, 2005.

[34] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 3rd edition, 2002.

[35] T. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, pages 23–51, 2007.

[36] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206–263, Oct. 2005.

[37] Sable research group McGill University. Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/, 2012.

[38] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universitat Munchen, 2002.

[39] S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. On generalized authorization problems. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 202–216. IEEE, 2003.

[40] D. Suwimonteerabuth. *Reachability in Pushdown Systems: Algorithms and Applications.* PhD thesis, Technische Universität München, 2009.

[41] R. Vall, L. Hendren, P. Lam, and C. Phong. Soot - a Java Bytecode Optimization Framework . In *IBM Centre for Advanced Studies Conference*, pages 1–11, 1999.

[42] B. Venners. *Inside the Java 2 Virtual Machine*. McGraw-Hill Companies, 2nd edition, 2000.

[43] D. Wallach and E. Felten. Understanding Java stack inspection. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 52–63. IEEE, 1998.

[44] J. Whaley. BDD-Based Deductive DataBase. http://bddbddb.sourceforge.net/index.html, 2004.

[45] J. Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams.* PhD thesis, Stanford, 2007.

[46] J. Whaley, D. Avots, and M. Carbin. Using datalog with binary decision diagrams for program analysis. *Programming Languages and Systems*, 3780, 2005.

Appendix A

Sample Usage of Package cwpds

This appendix displays full sample program of using package cwpds. The conditional weighted pushdown system created by below code is one described in Example 5.5.4; more gentle explanation could be found in Chapter 5 Section 5.6.

```java
package cwpds.test;

import com.google.common.collect.Sets;
import cwpds.*;
import cwpds.analysis.PERSemiring;
import cwpds.regex.RE;
import org.junit.Test;

import java.util.*;

/**
 * Permission Analysis Framework demonstration.
 * @author HUA, Vy Le Thanh
 */
public class FrameworkDemo {
  @Test public void test1() {
    System.out.println("=== Permission Analysis Framework demo ===");

    String p = ".";
    String n1 = "n1", n1R = "n1R", n2 = "n2", n2R = "n2R",
           n3 = "n3", n3R = "n3R";
    String eA = "eA", xA = "xA", eCP = "eCP", xCP = "xCP";
    String eMain = "eMain", xMain = "xMain";

    // Stack alphabet
    List<String> alphabet = Arrays.asList(eMain, xMain, eA, xA,
                                          eCP, xCP, n1R, n1, n2R, n2, n3R, n3);
    System.out.println("Stack alphabet = " + alphabet);
```

Figure A.1: Sample Usage of Package cwpds - Part I

```
    /* create regular expressions for pattern of stack condition */
    String CONDITION1 = "((eMain|xMain|n1|n1R|n2|n2R|n3|n3R|eCP|xCP)*)"
                        + RE.INTERSECTION + "((@)(n1R)(@))";
    // !CONDITION1 & C1
    String NOT_CONDITION1 = "(" + RE.COMPLEMENT + "("
                + "((eMain|xMain|n1|n1R|n2|n2R|n3|n3R|eCP|xCP)*)" + "))"
                + RE.INTERSECTION + "((@)(n1R)(@))";
    String CONDITION2 = "((eA|xA|eCP|xCP)*)" + RE.INTERSECTION
                         + "((@)(n3R)(@))";
    // !CONDITION2 & C2
    String NOT_CONDITION2 = "(" + RE.COMPLEMENT + "("
                             + "((eA|xA|eCP|xCP)*)" + "))"
                             + RE.INTERSECTION + "((@)(n3R)(@))";

    System.out.println("Regular expression for L1 = " + CONDITION1);
    System.out.println("Regular expression for !L1 = " + NOT_CONDITION1);
    System.out.println("Regular expression for L2 = " + CONDITION2);
    System.out.println("Regular expression for !L2 = " + NOT_CONDITION2);

    /* create condition DFA for Gamma* */
    DFA A0 = new RE(RE.ALL, alphabet).toDFA();

    /* create specific condition DFA for permission
       FilePermission("a.txt", "read") */
    DFA A1 = new RE(CONDITION1, alphabet).toDFA();
    DFA A2 = new RE(NOT_CONDITION1, alphabet).toDFA();

    /* create specific condition DFA for permission
       FilePermission("b.jar", "execute") */
    DFA A3 = new RE(CONDITION2, alphabet).toDFA();
    DFA A4 = new RE(NOT_CONDITION2, alphabet).toDFA();

    /* create product automaton */
    DFAProduct productDFA = new DFAProduct();
    productDFA.add(A0);
    productDFA.add(A1);
    productDFA.add(A2);
    productDFA.add(A3);
    productDFA.add(A4);

    /* create conditional weighted pushdown system */
    CPDS cpds = new CPDS();
    cpds.setSemiring(new PERSemiring(true, true));
    // <., xMain> -A_0, 1-> <., epsilon>
    cpds.add(0, PERSemiring.ONE, p, xMain, p);
```

Figure A.2: Sample Usage of Package cwpds - Part II

```
    // <., xCP> -A_0, 1-> <., epsilon>
    cpds.add(0, PERSemiring.ONE, p, xCP, p);
    // <., xA> -A_0, 1-> <., epsilon>
    cpds.add(0, PERSemiring.ONE, p, xA, p);
    // <., eMain> -A_0, 1-> <., n1>
    cpds.add(0, PERSemiring.ONE, p, eMain, p, n1);
    // <., n1> -A_0, 1-> <., eCP n1R>
    cpds.add(0, PERSemiring.ONE, p, n1, p, eCP, n1R);
    // <., eCP> -A_1, 1-> <., xCP>
    cpds.add(1, PERSemiring.ONE, p, eCP, p, xCP);
    // <., eCP> -A_2, ANY-> <., xCP>
    cpds.add(2, PERSemiring.ANY, p, eCP, p, xCP);
    // <., eCP> -A_3, 1-> <., xCP>
    cpds.add(3, PERSemiring.ONE, p, eCP, p, xCP);
    // <., eCP> -A_4, ANY-> <., xCP>
    cpds.add(4, PERSemiring.ANY, p, eCP, p, xCP);
    // <., n1R> -A_0, 1-> <., n2>
    cpds.add(0, PERSemiring.ONE, p, n1R, p, n2);
    // <., n2> -A_0, 1-> <., eA n2R>
    cpds.add(0, PERSemiring.ONE, p, n2, p, eA, n2R);
    // <., eA> -A_0, 1-> <., xA>
    cpds.add(0, PERSemiring.ONE, p, eA, p, xA);
    // <., n2R> -A_0, 1-> <., n3>
    cpds.add(0, PERSemiring.ONE, p, n2R, p, n3);
    // <., n3> -A_0, 1-> <., eCP n3R>
    cpds.add(0, PERSemiring.ONE, p, n3, p, eCP, n3R);
    // <., n3R> -A_0, 1-> <., xMain>
    cpds.add(0, PERSemiring.ONE, p, n3R, p, xMain);

    /* create initial P-automaton, given the entry point of program is
       eMain */
    PAutomaton pa = new PAutomaton();
    String finalState = "q";
    pa.setStartStates(Sets.newHashSet(p));
    pa.addTransition(PERSemiring.ONE, p, eMain, finalState);
    pa.setFinalStates(Sets.newHashSet(finalState));

    /* compute post* P-automaton by package cwpds */
    long startTime = System.currentTimeMillis();
    CPDSSat cpdsSat = new CPDSSat(cpds);
    PAutomaton paPost = cpdsSat.getPostStar(pa, productDFA);
    System.out.println("= post* P-automaton produced by cwpds package =");
    System.out.println(paPost);
    System.out.println("Number of states = " + paPost.getStates().size());
    System.out.println("Number of transitions = " + paPost.size());
    System.out.println("Analysis result at n1R is: "
                                               + cpdsSat.getValue(p, n1R));
    System.out.println("Analysis result at n3R is: "
                                               + cpdsSat.getValue(p, n3R));
```

Figure A.3: Sample Usage of Package cwpds - Part III

```
    /** Obtain performance statistics */
    long stopTime = System.currentTimeMillis();
    long elapsedTime = stopTime - startTime;
    System.err.println("Running time: " + elapsedTime + " milisecond(s).");

    // Get the Java runtime
    Runtime runtime = Runtime.getRuntime();
    // Run the garbage collector
    runtime.gc();
    // Calculate the used memory
    long memory = runtime.totalMemory() - runtime.freeMemory();
    System.err.println("Used memory is bytes: " + memory);
    System.err.println("Used memory is kilobytes: " + (memory / (1024L)));
  }
}
```

Figure A.4: Sample Usage of Package cwpds - Part IV