Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2011) 000–000

Procedia Computer Science

The 8th International Conference on Mobile Web Information Systems (MobiWIS)

A Service-oriented Approach to Mobile Code Security☆

Micah Jones, Kevin W. Hamlen

Computer Science Department, The University of Texas at Dallas, Richardson, TX 75080

Abstract

Client software for modern service-oriented web architectures is often implemented as mobile code applets made available by service-providers. Protecting clients from malicious mobile code is therefore an important concern in these architectures; however, the burden of security enforcement is typically placed entirely on the client. This approach violates the service-oriented paradigm.

A method of realizing mobile code security as a separate service in a service-oriented web architecture is proposed. The security service performs in-lined reference monitoring of untrusted Java binaries on-demand for client-specified security policies. An XML format for specifying these policies is outlined, and preliminary experiments demonstrate the feasibility of the approach.

Keywords: in-lined reference monitors, runtime monitors, mobile code

1. Introduction

Mobile code is an important implementation component of many modern web services. For example, geospatial web servers provide clients mobile bytecode applets that filter, analyze, and graphically render geospatial data obtained from the service within a local web browser. The use of mobile code in these contexts is motivated by the need for high efficiency and quality of service in service-oriented architectures (SOAs). That is, mobile code can reduce communication and processing overhead for the service-provider by shifting some of that overhead to the client.

Mobile code security, however, is typically not provided as a distinct service; clients must protect themselves from malicious or misbehaved mobile code by implementing their own local protection systems. These are typically realized as a fixed part of the client's mobile code execution environment. For example, Java bytecode applets undergo both static validation and dynamic monitoring by the Java Virtual Machine (JVM) to protect the client from potentially malicious code provided by untrusted service-providers. This arrangement results in a relatively inflexible collection of security policies available to typical code-consumers. For example, the JVM enforces basic memory-safety and object-encapsulation properties, but it does not enforce user- or application-specific policies, such as a policy that prohibits untrusted applets from opening more than 3 pop-up windows per run, or prohibits them from sending data to the network after they have received private data as input from the user. While enforcing such custom policies is

☆This material is based upon work supported by the U.S. AFOSR under Young Investigator Award FA9550-08-1-0044.

Email addresses: micah.jones1@utdallas.edu (Micah Jones), hamlen@utdallas.edu (Kevin W. Hamlen)

possible, it typically requires developing and installing new client-side VM or OS extensions for each new policy to be enforced—an impractical undertaking for many organizations and users.

As a more flexible alternative, we consider a service-oriented approach to securing mobile binary code. Our approach is based on *in-lined reference monitors* (IRMs) [1], which implement security monitors by automatically in-lining runtime security checks into the untrusted application's binary text before it is executed. Code-consumers in our framework submit untrusted Java bytecode and a desired security policy to a trusted in-lining service that instruments the bytecode with an IRM that enforces the policy. The resulting *self-monitoring code* is then digitally signed and returned to the code-consumer, who can verify the signature and safely execute it.

Numerous IRM systems have been developed for traditional mobile code architectures (e.g., [2–4]), though not for SOAs to our knowledge. Extending the technology to SOAs allows mobile code security to become a separate service in such an architecture, rather than an obligation that each client must satisfy for itself individually. The significant complexity of policy enforcement implementation is shifted to a trusted third party, affording code-consumers the flexibility of enforcing a wide array of potentially organization- and application-specific policies without implementing that functionality themselves. Lightweight devices that lack the computational resources necessary to analyze and instrument arbitrary bytecode binaries can therefore enforce these rich policies by leveraging the power of the SOA.

The service-oriented approach has numerous potential security advantages, including improved deployment speed and wider client coverage than traditional patching approaches. For example, a web client that is configured by default to always pass untrusted bytecode through an in-lining service before execution receives the benefits of security updates implemented by that service instantly, without needing to download and install security updates or patches for its OS or VM. Moreover, as new vulnerabilities are discovered and new enforcement strategies are invented, third-party in-lining services can often be updated and adapted more rapidly than typical OS/VM patches can be developed. This is because software patches can usually only be developed by a relatively small collection of experts who have access to the OS/VM source code, whereas IRM implementations depend only bytecode language standards available to the public (e.g., [5]). The service-oriented approach therefore provides defenders a means to quickly and comprehensively react to zero-day attacks for which no patch yet exists, and to protect users of legacy software who may be slow to apply patches.

The remainder of the paper proceeds as follows. Section 2 provides a brief overview of our bytecode rewriting system architecture. Security policies for our system are expressed as SPoX (Security Policy XML) documents [6], whose syntax is described in §3. Finally, §4 describes a prototype deployment of our system as a Java web service and some preliminary performance statistics. Section 5 concludes.

2. System Overview

Our in-lining service accepts as input an untrusted Java bytecode binary and a client-specified security policy. The service returns a modified binary that has been instrumented with code that enforces the policy when the modified binary is executed. The in-lining service never executes the untrusted code directly (which would be both prohibitively inefficient and unsafe), and the transformation process consists of a purely linear traversal of the untrusted bytecode rather than a more sophisticated static code analysis (many of which are worst-case exponential). This keeps runtimes for the service tractable.

Security policies are specified by the client as SPoX (Security Policy XML) [6] documents, which express mobile code policies as temporal properties that define the set of all permissible *event sequences* that may be exhibited by the untrusted applet. Security-relevant *events* are typically Java system API calls and their arguments, which are the means by which Java programs read or affect the external computing environment. Other program operations can be identified as security-relevant as well, as discussed in §3.

The alphabet of all program events deemed security-relevant is specified in SPoX using *pointcuts*, a concept derived from aspect-oriented programming (AOP). A pointcut is a declarative expression that describes a set of matching program operations, similar to how a regular expression describes a set of string values. The SPoX pointcut language is a binary adaptation of the one implemented by AspectJ [7] for Java source code, allowing policy-writers to develop policies that statically and/or dynamically regard binary-level method calls and their arguments, object pointers, and lexical contexts, among other properties.

The language of permissible event sequences is expressed by encoding it as a *security automaton* [8]—a finite- or infinite-state machine that accepts all and only those sequences that satisfy the security policy. Security automata

```
1 <state name="s" />
2 <edge name="fileAccess">
3     <nodes var="s">0,1</nodes>
4     <and><call>java.io.File*.*</call>
5          <argval num="1">
6            <streq>.*windows\\.*</streq>
7          </argval></and>
8 </edge>
9 <edge name="illegalSocketOutputStream">
10    <nodes var="s">1,#</nodes>
11    <call>java.net.Socket.getOutputStream</call>
12 </edge>
```

Figure 1. Policy that prohibits network sends after sensitive file reads

express *safety policies*, which essentially say that some "bad thing" must not happen.[1] The "bad thing" can be a collection of individual prohibited events or it can include particular orderings of events, in which case the policy is said to be *history-based*. For example, a policy that permits at most 10 simultaneously open files is history-based because each file-open is permitted contingent on the history of events that preceded it.

SPoX policies can be enforced as binary-level IRMs [9, 10]. Security checks are inserted at appropriate points around security-relevant instructions in an untrusted target program, causing impending policy violations to be dynamically detected and prevented. This instrumentation process is similar to *aspect-weaving* in AOP, except that in AOP both aspects and target programs are typically specified as source code, whereas our rewriter synthesizes binary code from a purely declarative policy specification and injects it into an untrusted binary program. This allows SPoX policies to be enforced by code-consumers on binaries without source code.

SPoX does not specify *how* to enforce a given policy; instead, it declaratively describes how security-relevant events affect an abstract security automaton state. There are therefore many ways to implement any given SPoX policy, affording enforcement mechanisms a great deal of flexibility.

State-transitions can be specified in terms of information gleaned from the current program state, such as method argument values, the call stack, and the current lexical scope. For example, the SPoX policy in Figure 1 encodes a policy that prevents network send operations after a program has read a private file. There is one state variable `s`, declared on Line 1. The first of two automaton edges is described by the `fileAccess` definition on Lines 2–8, which matches calls to `java.io.File` methods whose first arguments are strings containing the substring "`windows\`". Such operations change security automaton state `s` from 0 to 1. The `illegalSocketOutputStream` edge beginning at Line 9 matches network send operations. When the current state satisfies `s` = 1, such operations violate the security policy, as indicated by postcondition `#`.

In order to instrument this policy in an untrusted target program as an IRM, our rewriter locates all code points that match pointcuts in the policy and injects guard code around them to simulate the two-state security automaton. There are many correct guard code implementations of any given policy, affording rewriters a broad array of enforcement strategies. Our rewriter takes the approach of *reifying* the security state `s` into the program itself as a global variable. Immediately before calls to member methods of any class beginning with `java.io.File`, it injects guard code that tests whether `s` = 0 and compares the first argument's `toString` form to regular expression `.*windows\\.*`. If these dynamic tests succeed then `s` is assigned 1. Likewise, it injects guard code immediately before each call to `java.net.Socket.getOutputStream` that checks whether `s` = 1. If not, the operation is permitted; otherwise the program self-aborts to prevent the impending violation.

The surrounding untrusted binary code must also be prevented from maliciously circumventing the new guards or corrupting reified state variable `s`. To achieve this, the rewriter redirects any jumps that previously targeted security-relevant instructions to the injected guard code that surrounds them.[2] In addition, reified security states are imple-

[1] This is in contrast to *liveness policies*, which say that some "good thing" must eventually happen (cf. [3]).

[2] The only form of computed jump in Java bytecode is dynamic method dispatch. Thus, unrestricted computed jumps need not be supported, greatly simplifying the control-flow analysis.

$n \in \mathbb{N}$	natural numbers	$c \in C$	class names
$sv \in SV$	state variables	$iv \in IV$	iteration variables
$id \in ID$	object identifiers	$ed \in ED$	edge names
$pcn \in PCN$	pointcut names		

pol ::=	$spc^* sd^* edg^*$	**policies**
spc ::=	`<pointcut name="`pcn`">`pcd`<pointcut>`	**stored pointcuts**
sd ::=	`<state name="`sv`">[`c`]</state>`	**state declarations**
edg ::=		**edges**
	`<edge name="`ed`">`$pcd\ ep^*$`</edge>`	edgesets
	`|<forall var="`iv`" from="`a_1`" to="`a_2`">`edg^*`</forall>`	iteration
ep ::=	`<nodes [obj="`id`"] var="`sv`">` a_1`,`a_2 `</nodes>`	**edge endpoints**
a ::=	$n \mid iv \mid a_1$`+`$a_2 \mid a_1$`-`$a_2 \mid a_1$`*`$a_2 \mid a_1$`/`$a_2 \mid$ `(`a`)`	**arithmetic**

Figure 2. SPoX core language syntax

mented as private class fields so that malicious code cannot corrupt the reified security state and thereby defeat the in-lined monitor.

The correctness (i.e., soundness) of the approach follows from an inductive argument that injected variable `s` correctly tracks the security automaton state throughout the execution of the rewritten code [11]. Since `s` is never assigned `#`, executing the rewritten code never violates the policy. The proof can be informally summarized as follows: Reified state `s` is initially equal to the security automaton's start state label. This value is only subsequently modified by in-lined IRM guard code (because `s` is implemented as a private field that is inaccessible by other code). Each such modification is immediately preceded or succeeded by a security-relevant operation that changes the security automaton state likewise. Thus, the two remain synchronized with respect to one another.

3. Security Policies

The general syntax of a SPoX specification is provided in Figures 2 and 3. Each specification consists of a list of security automaton edge declarations. Each edge declaration consists of three parts:

- *Pointcut expressions* (Figure 3) identify sets of security-relevant events that programs might exhibit at runtime.
- *Security-state variable* declarations (*sd* in Figure 2) are used to construct the overall security automaton state. The security state is defined by the set of mappings of all state variables to their integer[3] values.
- *Security-state transitions* (*ep* in Figure 2) describe how events cause the security automaton's state to change at runtime.

Edges are specified by `<edge>` structures, each of which defines a (possibly infinite) *set* of edges in the security automaton. Each `<edge>` structure consists of a pointcut expression and at least one `<nodes>` declaration. The pointcut expression defines a common label for the edges, while each `<nodes>` declaration imposes a transition precondition and postcondition for a particular state variable. The precondition constrains the set of source states to which the edge applies, and the postcondition describes how the state changes when an event matching the pointcut expression occurs.

In general, state variables come in two varieties:

[3] Binary operator `/` in Figure 2 denotes integer division.

$n \in \mathbb{N}$	natural numbers	$c \in C$	class names
$re \in RE$	regular expressions	$md \in MD$	method names
$fd \in FD$	field names	$id \in ID$	object identifiers
$pcn \in PCN$	pointcut names	$mo \in MO$	modifiers

pcd ::=		**pointcuts**
	<*mtag*>*mo** *c.md*</*mtag*>	method accesses
	|<get>*mo** *c.fd*</get> | <set>*mo** *c.fd*</set>	field accesses
	|<argval num="*n*" [obj="*id*"]>*vp*</argval>	stack args (values)
	|<argtyp num="*n*">*c*</argtyp>	stack args (types)
	|<handler>*c*</handler>	exception handler executions
	|<this [obj="*id*"]>[*c*]</this>	*this* pointer references
	|<target [obj="*id*"]>[*c*]</target>	target object references
	|<cflow>*pcd*</cflow>	control flows
	|<pointcutid name="*pcn*" />	predefined pointcuts
	|<and>*pcd**</and> | <or>*pcd**</or> | <not>*pcd*</not>	logical operators
	|<true /> | <false />	constants
mtag ::=	call | execution | withincode	
vp ::=	<isnull /> | <streq>*re*</streq>	**value predicates**
	|<inteq>*n*</inteq> | <intle>*n*</intle>	

Figure 3. SPoX pointcut syntax

- *Instance security-state variables* describe the security state of an individual runtime object. They can be thought of as hidden field members of security-relevant classes.
- *Global security-state variables* describe the state of the overall system. They can be thought of as hidden, global, static program variables.

Instance state variables allow SPoX specifications to express per-object security properties. For example, a policy can require that each `File` object may be read at most ten times by defining an instance security-state variable associated with `File` objects and defining state transitions that increment each object's security-state variable each time that individual `File` object is read (up to ten times). In contrast, global security-state variables are used to express instance-independent security properties. For example, a policy can require that at most ten `File` objects may be created during the lifetime of the program by defining a global security-state variable that gets incremented each time any `File` object is created (up to ten times).

Counters and other repetitive automaton structures can be succinctly expressed in SPoX as `<forall>` structures that introduce a set of edges for each integer state in a range. For example, a resource bound policy might be expressed a security state `s` with edges from $s = i$ to $s = i + 1$ for each integer i in some interval $[0, n]$, where n is the resource bound. In SPoX, such a policy could be succinctly expressed as follows:

```
<forall var="i" from="0" to="9">
  <edge name="count">
    <nodes var="s">i,i+1</nodes>
    p
  </edge>
</forall>
```

This specifies a policy that permits operations matching pointcut *p* to occur at most 10 times. More complex configurations can be achieved by nesting `<forall>` structures and by using more complex arithmetic expressions as pre- and post-conditions of the `<nodes>` elements within them.

3.1. Pointcuts

A syntax for SPoX's pointcut language is given in Figure 3. We support all regular expression operators available in AspectJ for specifying class and member names. Since SPoX policies are applied to Java bytecode binaries rather than to source code, the meaning of each pointcut expression is reflected down to the bytecode level. We here describe the language informally; a formal denotational semantics is provided in [6].

Atomic pointcuts (i.e., those that do not recursively contain other pointcuts) can be categorized into those whose criteria are statically decidable and those that can only be decided dynamically (in general). Statically decidable pointcuts, such as `<call>`, can be matched against code points through a direct, static inspection of the bytecode. For example, the set of all bytecode instructions that match a `<call>` pointcut are exactly the `call` instructions that specify a method name that matches the pointcut's regular expression content. The binary format of Java `call` instructions specify the method name statically. In contrast, dynamically decidable pointcuts, such as `<argval>`, cannot be matched through purely static analysis because they depend on information that can only be determined at runtime, such as the values of items on the stack. Practical policies invariably demand an in-lining service that supports both types, and that therefore implement a combination of static and dynamic code analysis.

Static pointcuts include method accesses (`<`*mtag*`>`), field accesses (`<get>` and `<set>`), argument type constraints (`<argtyp>`), and exception handlers (`<handler>`). Method accesses are usually of greatest practical significance in real policies, so we support several different kinds of pointcuts for matching them. Instructions that call methods are matched by `<call>` pointcuts, whereas the entry points of the methods they call are matched by `<execution>` pointcuts. This distinction is important because when a security-relevant method *m* is part of the untrusted binary itself, using `<execution>` leads to a more efficient IRM implementation because the guard code is injected at the entry point of *m* instead of at every instruction that calls *m*. However, when *m* is external to the binary (e.g., it is a system API method), `<call>` is required since *m* itself cannot be rewritten. The `<withincode>` pointcut matches all code points within a given method, and is typically used within a conjunction to identify all instances of a particular operation within a given lexical scope.

Dynamic pointcuts include argument value tests (`<argval>`), and dynamic typing tests (`<this>` and `<target>`). The first of these tends to be the most important, and allows dynamic checking of method call arguments and field assignments. Value predicates *vp* compare these values to integers, string regular expressions, or simply `null`. Types and subtypes of the target object of a method call or field access can be dynamically matched by `<this>` and `<target>` pointcuts, respectively.

We anticipate that realistic policies that exhaustively itemize all possible binary-level, security-relevant operations will become quite large documents over time. It is therefore important to be able to modularize such documents by assigning names to commonly used pointcut expressions and allowing those names to be used as references in larger pointcuts. The `<pointcutid>` pointcut allows such references, and can be thought of as a macro that expands to the referent's definition.

4. Web Service Implementation

We have implemented a version of our rewriter which acts as a Java web service application. Using an HTTP request with the POST method, a client uploads two files: a SPoX policy and an untrusted JAR (Java ARchive) file. The server then provides these two files as input to the rewriter, which creates a new, rewritten JAR that enforces the security policy. Finally, the rewritten JAR file is returned as an HTTP response to the client.

The server-side code consists of about 7400 lines of Java code, 7200 lines of which are devoted to Java binary parsing, rewriting, and code generation, and 200 lines of which implement the servlet that makes the rewriter accessible as a web service. We use the Apache BCEL library for low-level Java binary reading and writing. In order to manage the file uploads, we use the Apache Commons fileUpload library, which obtains Java `FileInputStreams` from the request data. The files are saved locally on the server in temp files, which are submitted as input to the rewriter. The newly rewritten JAR file is then copied to the HTTP response as a binary stream.

```
<state name="s" />
<edge name="badUpload">
  <nodes var="s">0,#</nodes>
  <or><and><set>Picasa.UploadPhoto.name</set>
           <not><argval num="1">
                  <streq>.*\.(jpg|jpeg|tif|tiff|png|bmp|gif)</streq>
                </argval></not></and>
      <and><set>GoogleDocs.UploadDoc.file</set>
           <not><argval num="1">
                  <streq>.*\.(doc|docx|txt|rtf)</streq>
                </argval></not></and></or>
</edge>
```

Figure 4. Policy that prohibits uploads of files with non-whitelisted extensions

We ran the rewriter service on three different applications, with different policies for each. All tests were performed on a Dell Studio XPS notebook computer running Windows 7 64-bit with an Intel i7-Q720M quad core processor, a Samsung PM800 solid state drive, and 4 GB of memory. The server-side code ran on an instance of GlassFish Server 3.1. Provided runtimes account only for time spent rewriting and performing related file I/O, and do not include any network operations such as uploading and downloading files.

jWeatherWatch. We enforced the policy from Figure 1 on a weather widget application called jWeatherWatch [12]. The policy prohibits the application from accessing files in the Windows directory. The application obeys this policy, so the rewritten program showed no observable change in behavior. The uploaded JAR was 140 KB in size and rewriting increased its size by 6K (4%). Total processing time was approximately 4.3 seconds.

Google.mE. Google.mE [13] is an open source Java application that acts as a client for various Google web applications, including GoogleDocs, Picasa, YouTube, and GMail. One of its features supports uploading of files to many of those services. We chose to enforce the policy in Figure 4, which prohibits uploads of non-picture files to Picasa and non-document files to GoogleDocs. The file type is identified by whitelisting permissible file extensions; filenames with extensions not explicitly listed in the respective `<streq>` elements are prohibited. When we attempted to upload a file with an unsupported extension to GoogleDocs, the rewritten application halted execution as expected.

The original JAR was 921 KB in size while the rewritten one was 513 KB—a size reduction of over 44%. The reduction is primarily to the lack of compression in the original JAR, whereas our rewriter compresses the output JAR by default. The runtime was reported as 21.7 seconds.

Jeti. Jeti [14] is a simple Jabber IM client. We enforced the policy in Figure 5, which limits the number of simultaneous socket connections to 5. Such a policy might be used to protect a user from DDoS malware disguised as legitimate web service clients. Socket connection events increment security state `s` until `s` = 5, at which point the next connection event signals a policy violation. Socket close events decrement `s`.

Interestingly, enforcement of this policy uncovered an apparent bug in the application. When a login is successful and the user later logs out, the connection is properly closed; however, connections for failed logins are never properly closed. Thus, six successive failed login attempts trigger a policy violation, and the rewritten program halts.

The original JAR file was 533 KB in size and rewriting decreased it by 59 KB (11%). In this case, the size reduction is a result of the rewriter stripping out unnecessary metadata in the JAR's internal class files. The total processing time was 20.4 seconds.

5. Conclusion

We implemented a web service that automatically in-lines security monitors into untrusted Java binary code in accordance with a client-specified, temporal security policy. The policy is specified using the SPoX XML format, which encodes temporal code properties as security automata. This allows client-specific security requirements for

```
<pointcut name="connect">
    <or><and><call>java.net.Socket.new</call>
             <argtyp num="2">int</argtyp></and>
        <call>java.net.Socket.connect</call></or></pointcut>
<state name="s" />
<forall var="i" from="0" to="4">
    <edge name="inc_connections">
        <nodes var="s">i,i+1</nodes>
        <pointcutid name="connect" /></edge>
</forall>
<forall var="i" from="1" to="5">
    <edge name="dec_connections">
        <nodes var="s">i,i-1</nodes>
        <call>java.net.Socket.close</call></edge>
</forall>
<edge name="six_connections">
    <nodes var="s">5,#</nodes>
    <pointcutid name="connect" /></edge>
```

Figure 5. Policy that prohibits more than 5 simultaneous connections

mobile code to be shifted to a separate, trusted security service in a service-oriented architecture. The separate service is lighter-weight than platform-as-a-service approaches because it does not actually execute the mobile code.

Processing time for rewriting averaged 0.02 seconds per kilobyte. This is acceptable for small applications, but would need to be improved for larger applications on the order of megabytes in size. Future work should also investigate certification approaches, such as those based on proof- or model-carrying code [15, 16], that could allow the in-lining service to remain untrusted by checking its output with a lighter-weight verifier [17].

References

[1] F. B. Schneider, Enforceable security policies, ACM Trans. Information and System Security 3 (1) (2000) 30–50.
[2] F. Chen, G. Roşu, MOP: An efficient and generic runtime verification framework, in: Proc. ACM Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), 2007, pp. 569–588.
[3] J. Ligatti, L. Bauer, D. Walker, Run-time enforcement of nonsafety policies, ACM Trans. Information and System Security 12 (3) (2009).
[4] I. Aktug, K. Naliuka, ConSpec - a formal language for policy specification, Science of Computer Programming 74 (1–2) (2008) 2–12.
[5] T. Lindholm, F. Yellin, The Java™ Virtual Machine Specification, 2nd Edition, Prentice Hall, 1999.
[6] K. W. Hamlen, M. Jones, Aspect-oriented in-lined reference monitors, in: Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS), 2008, pp. 11–20.
[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: Proc. European Conference on Object-Oriented Programming (ECOOP), Vol. 2072, 2001, pp. 327–355.
[8] B. Alpern, F. B. Schneider, Recognizing safety and liveness, Distributed Computing 2 (1986) 117–126.
[9] M. Jones, K. W. Hamlen, Enforcing IRM security policies: Two case studies, in: Proc. IEEE Intelligence and Security Informatics Conference (ISI), 2009, pp. 214–216.
[10] M. Jones, K. W. Hamlen, Disambiguating aspect-oriented security policies, in: Proc. Aspect-Oriented Software Development (AOSD), 2010, pp. 193–204.
[11] M. Sridhar, K. W. Hamlen, Model-checking in-lined reference monitors, in: Proc. International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), 2010, pp. 149–151.
[12] jWeatherWatch, `http://code.google.com/p/jweatherwatch` (2009).
[13] Google.mE, `http://sourceforge.net/projects/googleme` (2010).
[14] Jeti, `http://jeti.sourceforge.net` (2007).
[15] M. Dam, A. Lundblad, A proof carrying code framework for inlined reference monitors in Java bytecode, Computing Research Repository (CoRR) abs/1012.2995 (2010).
[16] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, D. DuVarney, Model-carrying code: A practical approach for safe execution of untrusted applications, in: Proc. ACM Symposium on Operating Systems Principles (SOSP), 2003, pp. 15–28.
[17] M. Sridhar, K. W. Hamlen, Flexible in-lined reference monitor certification: Challenges and future directions, in: Proc. ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV), 2011, pp. 55–60.