PLAS'12, ACM, pp 3:1-3:11. 2012

Security-Policy Monitoring and Enforcement with JavaMOP

Soha Hussein Patrick Meredith Grigore Roşu
University of Illinois at Urbana-Champaign
soha@illinois.edu/pmeredit@illinois.edu/grosu@illinois.edu

Abstract

Software security attacks represent an ever growing problem. One way to make software more secure is to use Inlined Reference Monitors (IRMs), which allow security specifications to be inlined inside a target program to ensure its compliance with the desired security specifications. The IRM approach has been developed primarily by the security community. Runtime Verification (RV), on the other hand, is a software engineering approach, which is intended to formally encode system specifications within a target program such that those specifications can be later enforced during the execution of the program. Until now, the IRM and RV approaches have lived separate lives; in particular RV techniques have not been applied to the security domain, being used instead to aid program correctness and testing. This paper discusses the usage of a formalism-generic RV system, JavaMOP, as a means to specify IRMs, leveraging the careful engineering of the JavaMOP system for ensuring secure operation of software in an efficient manner.

1. Introduction

Assuring that programs comply to desired security policies is crucial to avoiding security vulnerabilities. Schneider [30] proposes inlining *Security Automata* into a target program to monitor a set of events for security violations. A series of security systems [6, 15–17, 20] followed, which were built and designed specifically for the sake of monitoring security policies at runtime. These monitors are referred to as *Inlined Reference Monitors (IRMs)* [14, 30].

One of the motivations behind this research is that an inlined monitor is more easily able to enforce behavioral guarantees within a program than an external monitor because a richer set of observable program events is available. Additionally, this allows for easy customization of the varying security concerns of different target programs, which works to satisfy the basic principle of *least privilege*[29]: the minimal level of privileges needed for a given program is granted via a custom tailored set of IRMs.

On the other hand, *Runtime Verification (RV)* [4, 22, 31] aims to combine testing with formal methods in a mutually beneficial way. The idea underlying Runtime Verification is that system requirements specifications, typically formal and referring to temporal behaviors and histories of events or actions, are rigorously checked at runtime against the current execution of the program, rather than statically against all hypothetical executions.

JavaMOP [27] is a formalism-generic RV monitoring framework for Java. JavaMOP provides a rich and expressive specification language incorporating many complex features such as policy parameterization, corrective actions upon policy-defined conditions, and usage of an extensive repository of logical formalisms. JavaMOP supports these features with a low performance overhead: 10% or less. While JavaMOP does not provide a rewriter of its own, its output is an AspectJ [23] file that can be weaved into a target program using any standard AspectJ compiler.

As an effort to establish a tight connection for IRM as a form of the RV approach, we explore the usage of JavaMOP as a means to specify and enforce security policies. Additionally, we extend JavaMOP, where necessary, in order to express all of the desired security policies. This is, then, an application paper, which focuses on showing that an RV approach (JavaMOP) is not only quite capable of expressing security policies, but is able to do so in highly expressive[1] and more efficient manner.

Figure 1 shows an example of a classic security policy known as the Chinese wall [10], specified using JavaMOP. It will be described in more detail in Section 3.4, but briefly, the policy attempts to keep users in a system from accessing objects of different datasets that are in the same *Conflict Class*. As an example of this policy, consider institutions providing corporate business services where financial analysts (subjects) advise each business (dataset) based on its sensitive information (objects). Such an analyst must keep the confidentiality of the information of corporations he has accessed. On the other hand, the analyst can not advise another corporation that competes with companies they have already advised. Such competing corporations are said to have conflict of interest, and they are grouped into the same Conflict Class.

The JavaMOP policy shown in Figure 1 provides a powerful way to enforce the policy. It does not look for direct accesses of objects by subjects; rather, the policy tracks all accesses of objects that lie in the call stack of the subject, thus tracking all possible object accesses done directly or indirectly. Note that this specification catches violations *before* they occur. This is guaranteed by the `CFG` pattern, which describes accesses that occur outside a matching pair of method call and returns. The pattern can only fail if the access happens *inside* matched pairs of calls and returns (i.e., inside a method call). Because the handler fires *before* the access of an object is granted, illegal accesses will be denied.

The paper is organized as follows: Related work is discussed in Section 1.1. A quick outline of JavaMOP's main features is presented in Section 2. Section 3 discusses the usage of JavaMOP for expressing security policies as well as policy composition. Section 4 addresses different security features in JavaMOP. Performance overhead of security policies in JavaMOP is discussed in Section 5. Finally we conclude our work and outline future work in Section 6.

[1] JavaMOP supports multiple logical formalisms.

```
ChineseWall(Subject S) {
  SubjectWall monitoredSubjectWall;
  Obj readObject;
  event methodCall before(Subject S):
   call(* Subject.*(..)) && target(S) {
     if (monitoredSubjectWall == null)
     monitoredSubjectWall =
       new SubjectWall(S);
  }
  event methodReturn after(Subject S) :
   call(* Subject.*(..))&&target(S) {}
  event access before(Obj O):
    call(* Obj.Read()) && target(O) {
      readObject = O;
  }
  cfg: S -> S access | S M | epsilon,
       M -> M methodCall M methodReturn
          | epsilon
  @fail{
    SubjectWall sw =
        __MONITOR.monitoredSubjectWall;
    Obj o = __MONITOR.readObject;
    if(sw.conflictClassContains(o)
       && !sw.dataSetContains(o)){
      System.out.println(
  "Chinese Wall is violated. Halting..");
      Runtime.getRuntime().halt(1);
    }
    sw.addToConflictClass(o);
    sw.addToDataSet(o);
  }
}
```

Figure 1. The Chinese Wall Policy in JavaMOP using Context Free Grammar (cfg)

1.1 Related Work

IRM Systems. Since Schneider introduced the theory of Execution Monitors (EM) [30], a number of systems have been designed to implement the theory.

For example, SASI [14, 16] is a low level, platform specific system that supports two prototypes, one for x86 and another for Java bytecode. SASI uses Security Automata written in the SAL (language specification for security automata), which in turn uses ISA (Instruction Set Architecture) to express security relevant events. Using ISA instructions to express the security policy enriches the vocabulary that is used by SASI, and it enables its users to write low level security policies such as SFI (Software Fault Isolation). However, it makes it harder to express application-level security policies since a single application-level event might map to multiple instructions in the ISA. In addition to that, SASI does not support other important features such as parameterization of policies, general computation states and the specification of arbitrary code within the policies.

PoET [14, 15] provides a platform-independent toolkit that uses different libraries to expose both ISA instructions as well as application APIs. This allows both application-level and low level policies to be expressed. However it does not resolve conflicts between potentially conflicting policies nor does it provide a declarative parameterizations of policies.

Naccio [17], on the other hand, defines resource library methods, and it uses its writer to create wrappers around original method calls such that security checks are first done before the invocation of a target program method is allowed. However, it does not support arbitrary code or parameterization of policies. While Naccio supports parameters that can be bound statistically to provide limits for policies, they cannot be used to support multiple monitors of the same policy.

Polymer [6, 24, 25] is a Java specific IRM that has two distinguished features: composition of policies (policies are first class objects) and support for multiple remedial actions upon violation, as well as the addition of arbitrary code inside the policies. While Polymer does not have explicit support for parameterization, one may manually program parameterization in Polymer because it allows arbitrary code in policies.

SPoX [20] is a declarative policy specification language that uses an XML-like structure to express invalid behavior for target programs. SPoX supports single parameter policies, but lacks other important features, such as general computation states and arbitrary code. Its only supported action on policy failure is termination.

The security policy checking and enforcement systems discussed above have made significant progress towards implementing the IRM paradigm. Like mentioned above there are, however, aspects that still need to be further addressed in order for these systems to offer the functionality needed to express and enforce complex security policies and compositions of them, such as: generality in specification formalisms, so one can chose the best formalism for any given policy; powerful composition of policies, so that one can check and enforce multiple, possibly interacting policies at the same time against a target system; parametricity of security policy specifications, so that various instances of the same policy can co-exist, one per group of interacting subjects; finally, efficient implementations of all the above.

RV Monitoring Systems. Many approaches have been proposed to monitor program executions against formally specified properties. These different systems were all designed with the idea of monitoring safety properties in mind for the purposes of either enforcing the *functional correctness* of post-production target programs or debugging and testing target programs during the production phase. These safety properties tend to focus on the correct usage of various APIs, e.g., ensuring that iterators are used in a manner that will not cause the target program to crash, rather than the security.

All runtime monitoring approaches [3, 5, 9, 12, 19, 26, 32] except for JavaMOP,the system used here, have their specification formalisms hardwired and *no two of them* share the same logic. This observation strengthens the notion that there is no *silver bullet* specification formalism for all purposes. Additionally, most systems focus on only the violation or validation of a pattern or formula, while JavaMOP allows any number of logical *categories* on which to trigger handlers. This can be anything from violation or validation, to arbitrary sets of states in a finite state machine. This is particularly important for non-finite logics like the context-free grammars (CFGs) supported by JavaMOP, as they are not closed under complementation, that is, one could not simply invert the pattern and monitor for the opposite category. The generality of logical formalism and ability to monitor for multiple possible categories makes JavaMOP the most logical choice for a candidate RV monitoring system with which one can specify security policies. Another large benefit of JavaMOP over competing RV Monitoring systems is that it is far more efficient, as extensive experimentation has shown [27].

1.2 Assumptions and Limitations

Throughout this paper, we assume that the rewriter that inlines the JavaMOP generated AspectJ code, namely the AspectJ compiler, is *trusted*. We believe that this is a reasonable assumption because the role of JavaMOP ends with the generation of correct AspectJ code, which, if inlined correctly, would ensure the compliance of the program with the security policy of interest. Thus the certification of the AspectJ compiler, or of any other compiler that can do the same job, is beyond the role of JavaMOP. This is a long standing

```
<JavaMOP Specification>::=
  <Specification Header>
  <Event Declaration>
  <Property>
  <Handler>

<Specification Header}> ::=
<Modifiers><Spec.Id>
  "("<Java Parameters>")""{"
<Java Variables Declarations>

<Event Declaration}> ::=
  "event"<Id><Extended AspectJ advice>"{"
     [<Event Actions>]"}"

<Property>::=
  <Logic Name>":"<Logic Syntax>

<Handler> ::=
  "@"<Logic State>"{"
      <Java Statements>"}""}"
```

Figure 2. JavaMOP Syntax

problem that we do not discuss thoroughly here as it is beyond the scope of this paper.

1.3 Contributions

The contributions of this paper are as follows:

- We explore, for the first time, the usage of a generic software monitoring system, JavaMOP, as an IRM system to express and enforce security policies.
- We provide a novel, powerful, and efficient specification of the well known Chinese wall security policy.
- We highlight some important features of JavaMOP in the context of checking security policies, such as parameterization and the usage of the around joinpoint. The latter was added to the JavaMOP system specifically for monitoring and enforcing security policies.
- We introduce novel and flexible policy composition and manipulation using JavaMOP.
- We provide thorough experimental results for security policies specified using JavaMOP as well as Polymer and SPoX.

2. JavaMOP as a Generic RV System

Monitoring oriented programming (MOP) is a generic monitoring framework that integrates specification and implementation by checking the former against the latter at runtime. JavaMOP is the Java instance for MOP; for other MOP instances the reader is referred to [27]. To enforce JavaMOP specifications, the JavaMOP tool automatically synthesizes monitors from the desired property specifications as AspectJ code which may be weaved into a target program using any standard AspectJ compiler.

Figure 2 shows the syntax for JavaMOP specifications. It is composed of four main parts. The first part is the <Specification Header>. This is where a user provides one or more <Modifier(s)>[2], a specification name <Spec.Id>, a list of specification parameters (if any) <Java Parameters> written in Java syntax, and finally a list of local specification variables <Java Variables Declarations> which can be used and manipulated inside the specification.

[2] <Modifiers> are a selection of running options to the specification, i.e., parameter binding mode, synchronization mode between multiple monitors, suffix matching mode or perthread mode; detailed description of each modifier can be found in [27].

```
1  SafeLock(Lock l, Thread t){
2    int acquire_count, release_count;
3    event acquire before(Lock l, Thread t):
4      call(* Lock.acquire())&&
5      target(l)&&thread(t){++acquire_count;}
6    event release before(Lock l, Thread t):
7      call(* Lock.release())&&
8      target(l)&&thread(t){++release_count;}
9    event begin before(Thread t):
10     execution(* *.*(..))&&
11     thread(t)&&!within(Lock+){}
12   event end after(Thread t):
13     execution(* *.*(..))&&
14     thread(t)&&!within(Lock+){}
15   cfg: S ->  S begin S end
16           |  S acquire S release
17           |  epsilon
18   @fail {
19     System.out.println((
20     (__MONITOR.acquire_count
21       > __MONITOR.release_count)?
22      "not enough releases before"
23      + " end of method"
24    : "too many releases of lock"));
25   }
26 }
```

Figure 3. JavaMOP Specification (Safe Lock)

The second part of the JavaMOP specification is the <Event Declaration>, where the program points a user wishes to monitor are specified. The declaration starts with the keyword event followed by a unique event name <Id>, then the signature of the action to be monitored using a slightly extended AspectJ syntax <Extended AspectJ advice>. The last part in the event declaration is the <Event Actions>, which may optionally contain Java statements that will be executed upon matching the event signature.

The third part in a JavaMOP specification is the <Property> that the specification will monitor. JavaMOP is specification formalism independent, supporting various logical formalisms, such as finite state machines (FSM), extended regular expressions (ERE), context free grammars (CFG), linear temporal logic (LTL), and past time linear temporal logic (PTLTL) [27].

Finally, the fourth part of a JavaMOP specification is the <Handler>, that will be executed when a specified logic state is reached by the monitor, e.g., validation or violation. This may optionally contain any <Java Statements> extended with a few JavaMOP constructs.

Figure 3 shows an example of a JavaMOP specification. This specification describes a basic programming principle for correct usage of locks: methods within a thread should release each lock as many times as it is acquired.

The first part of this specification names the property SafeLock, defines the specification parameters: Lock l and Thread t (meaning that the specification will be monitored for each combination of Lock and Thread instances), and finally the local variable declarations: acquire_count and release_count, which are used to provide more meaningful error messages when locks are misused.

The second part in the specification is where the actions of interest are defined; for instance, this specification defines four events: acquire when a lock is acquired, release when a lock is released, begin to capture the beginning of a method call, and end to capture the end of a method call. As mentioned, JavaMOP borrows and extends the syntax of AspectJ in event declarations. For example, the acquire event is declared to occur before a function call to the acquire method of the class Lock. The target and thread clauses are used to bind parameters in the event. The release event

```
1 DisableSystemCalls() {
2   event systemCalls before() :
3     call(* Runtime.exec(..)) {
4       System.out.println(
5 "System Calls are not allowed. Halting");
6       Runtime.getRuntime().halt(1);
7     }
8 }
```

Figure 4. Disable System Calls

```
1 RestrictSystemCalls() {
2   event systemCalls Object around():
3     call(* Runtime.exec(..)){}
4   ere : systemCalls+
5   @match {
6       __SKIP; // action is bypassed
7    }
8 }
```

Figure 5. Restrict System Calls

has a similar declaration only modified to capture calls to release a lock and counting number of releases instead. begin and end are very similar, save that begin captures the beginning of a method call by using the before advice, while end uses the after advice. Note that the !within(Lock+) filters out all beginning (or end) of methods that are defined in Lock or any of its subclasses.

The fourth part of the specification is a formal description of the desired property. As mentioned before, JavaMOP is logical-formalism-independent. In this example, the property description begins with cfg, meaning that a context free grammar (CFG) is used, and continues with a CFG pattern that defines our desired property. Note that the terminals of the grammar are our defined events.

The last part of the specification consists of handlers to execute in different states of the corresponding monitor. Categories considered in this paper are @match and @fail, which trigger when a pattern language (e.g., ERE or CFG) match or fail to match, respectively, and @validation and @violation, which trigger when a logical language (e.g., LTL) evaluates to true or false, respectively. In Figure 3, the handler starts with @fail, printing an appropriate warning message according to the number of locks being acquired and released.

3. Security Policies in JavaMOP

In this section we address specifications of security policies in JavaMOP. Our presentation is organized into main security concerns, and we identify how JavaMOP can be used to address each of them. In each specification, the occurrence of "..."[3] denotes a place where code has been omitted due to space constraints or redundancy.

3.1 Access Control

One of the main security principles is the restriction of each program's accesses to available system resources. For example, Figure 4 shows a JavaMOP specification which can be used to prevent the target program from invoking system calls. This is a common type of access control where the restricted resource is the invocation of executable files.

This specification uses only the minimal structure of a JavaMOP specification (called a *raw specification*); it contains no logic formula to express the violation nor does it contain a handler section. In fact, the specification monitors a single event which is exactly before the invocation of a system call. Just before such an event is executed, the specification halts the program as a corrective action to avoid compromising the security of the system.

A more refined way of realizing the same above policy is shown in Figure 5: RestrictSystemCalls. In this variation of the specification, system calls are handled differently: the target is not halted instead it is allowed to proceed after skipping the current instruction of the system call event. The specification is still monitoring the occurrence of a single event, but there are three main differences.

[3] Note that this is not the same as "..", which is part of the AspectJ syntax.

```
1 SeperationOfDuties(
2         Subject S,
3         LoanRequest L){
4   event internalRating
5     before(Subject S, LoanRequest L):
6     call(* Subject.internalRate(
7           LoanRequest))
8     && args(L) && target(S) {}
9   event externalRating
10    before(Subject S, LoanRequest L):
11    call(* Subject.externalRate(
12          LoanRequest))
13    && args(L) && target(S){}
14  event loanApproval
15    before(Subject S, LoanRequest L):
16    call(* Subject.Approveloan(
17          LoanRequest))
18    && args(L) && target(S) { }
19  ere: internalRating+ | externalRating+
20     | loanApproval+
21  @fail{
22    System.out.println(
23      "Conflicting duties. Halting.");
24    Runtime.getRuntime().halt(1);
25  }
26 }
```

Figure 6. Separation of Duties

The first is that the monitoring is done around (line 2) the occurrence of the critical event (allowing it to be skipped later). The second difference is that an ERE (extended regular expressions) pattern (line 4) is used, systemCalls$^+$ (e^+ means one or more instances of e). This will match any occurrence of systemCalls. The third is that a match handler actually skips the event matched by the around joinpoint using the __SKIP keyword (line 9).

3.2 Separation of Duties

Separation of duties is also a main security principle that enforces internal controls to prevent frauds. Figure 6 shows an example of a possible SeparationOfDuties policy in JavaMOP. This policy must be customized according to the target program. This policy is borrowed from Ponder [11], and it disallows a single user to perform more than one action for a given loan request. This means that even though a user might have the privilege to perform a certain action for loans, they might be disallowed to perform the said action if it contradicts with another critical action they have performed for the same loan.

Note that this specification is parametric in the Subject and the Loan (line 1). This shows a powerful feature of JavaMOP that allows a specification to be parametric in multiple objects. A different monitor *instance* will be created for each combination of instantiated objects matching the parameters. This means the ERE pattern only fails to match if a given Subject, S, performs more than one type of action on a given LoanRequest, L. The pattern achieves this by only allowing each event to occur in conjunction with other instances of the same event. Note that S will not be

```
1  SqlInjection() {
2    Set<String> taintedString = new HashSet();
3    event userInput
4      after(String tainted):
5      call(* ServletRequest.getParameter(..))
6      && target(tainted) {
7        taintedString.put(tainted);
8      }
9    event propagate
10     after(StringBuffer SB, String S):
11     call(* StringBuffer.new(String))
12     && target(SB) && args(S)
13   || call(* StringBuffer.append(String))
14     && target(SB) && args(S) {
15       if(taintedStrings.contains(S))
16         taintedStrings.put(SB.toString());
17     }
18   event usage
19     before(String S):
20     call(* Statement.executeQuery(String))
21     && args(S) {
22       if(taintedStrings.contains(S))
23         util.checkSafeQuery(S);
24     }
25 }
```

Figure 7. SQL Injection

restricted from performing a *different* action on LoanRequest L_2 (assuming $L_2 \neq L$).

3.3 SQL Injection

Figure 7 shows another security policy that uses a raw JavaMOP specification. The policy detects SQL-injection attacks [2], where malicious users try to corrupt a database by inserting unsafe SQL statements through system input.

In SQL injection, a string is tainted when it depends upon some user input; when a tainted string is used as an SQL query, it should be checked to avoid potential attacks. In Figure 7, a HashSet is declared to store all tainted strings. Three types of events need to be monitored: userInput occurs when a string is obtained from user input (by calling ServletRequest.getParameter()); propagate occurs when a new string is created from another string via a StringBuffer; finally, usage occurs when a string is used as a query.

Appropriate actions are triggered at observed events: at userInput, the user input string is added to the tainted set; at propagate, if the new string is created from a tainted string then it is marked as tainted, too; at usage, if the query string is tainted then a provided method, called Util.checkSafeQuery, is called to check the safety of the query. Thus the safety check, which can be an expensive operation, is invoked dynamically, on a by-need basis.

3.4 Wall Policies

We discuss here two policies: the FileNetworkWall and the Chinese Wall policies, where the first is a computer specific variation of the second, for restricting conflicting uses of system resources for applications.

We start by showing in Figure 8 the FileNetworkWall policy, where the target program is allowed to either access the network or the file system, but not both. The policy is elegantly specified by grouping all file methods into a single fileAccess event (lines 3-4) and likewise grouping all network methods into a single networkAccess event (lines 6-7). The PTLTL formula will cause the handler to be triggered if there is an access to the network with a file access in the past, or vice versa. Here, the symbol <*> means, "eventually in the past."

```
1  FileNetworkWall() {
2    event fileAccess before():
3      call(* Runtime.exec(..))
4    || call(* File.creatNewFile())
5    || ... //other file access methods
6    {}
7    event networkAccess before():
8      call(URL.new(..))
9    || call(Socket.new(..))
10   || ... //other network access methods
11   {}
12   ptltl: (fileAccess => <*> netWorkAccess)
13        or (netWorkAccess => <*> fileAccess)
14   @validation {
15     System.out.println(
16   "Conflicting resource access. "
17 + "Halting..");
18     Runtime.getRuntime().halt(1);
19   }
20 }
```

Figure 8. File Network Wall

The classic ChineseWall [10] policy in terms of subjects and objects laying in different dynamically defined conflict classes is shown in Figure 1 from Section 1. This specification provides a novel handling for the Chinese wall: it does not look for direct accesses of objects by subjects. Rather, the policy tracks all accesses of objects that lie in the call stack of the subject. The approach was chosen because there may be accesses to objects that may be performed in several different methods of the subject where the reference to the subject is not immediately available to AspectJ. The policy performs stack inspection for each subject to ensure that accessed objects are not in conflict with previously accessed objects of different datasets.

The policy is parametric in the Subject, meaning that there will be a monitor instance created for each subject instance monitoring the validation and violation of the policy. The policy requires defining an auxiliary class which is defined in another file and is not shown here: SubjectWall. A SubjectWall object carries the list of conflict classes and datasets of accessed objects for each subject.

The policy also defines two local monitor variables, readObject and monitoredSubjectWall. The first carries the object accessed by the subject, the second is an instance of SubjectWall, where all datasets and conflict classes of accessed objects are stored.

The policy defines three events. The methodCall event denotes all method calls made by a subject instance; its action creates a monitoredSubjectWall for a given subject, if one does not already exist. The methodReturn event refers to the end of method calls made by subjects. The access event looks for any read of an object. Note that the read access is not parameterized by the subject, so it will match a read performed by any subject rather than just the subject of the current monitor instance.

A CFG pattern, which is matched for failure, is used to express the property. Here, the handler will be triggered whenever there is an access that occurs between at least one pair of methodCall and methodReturn events. An access will be seen outside of methodCall and methodReturn if the access belongs to a different Subject than the Subject of interest for a given monitor instance, and we do not wish the handler to be triggered in such a case.

The handler checks if the accessed object is in the subject's conflict class but not its data set. This is performed through the SubjectWall sw, which defines the wall of datasets and conflict classes accessed by the subject. Whenever there is an access to an object, the object's dataset value and conflict class value are validated against the subject's wall of previously accessed datasets and conflict classes using the method calls dataSetContains(O)

and conflictClassContains(O). If the object's conflict class value is found but the dataset value is not found within the subject's wall (implying that the subject has accessed another object of a different dataset that lies in the same conflict class of the object being accessed) then the target is halted, otherwise the subject wall is updated to reflect the accessed dataset and conflict class of the object and the access is allowed. The conflict classes are constructed by the target program using the SubjectWall class (and possibly other associated classes).

To our knowledge, no monitoring systems other than JavaMOP can even specify this property conveniently. For example, in SASI [14, 16] one needs to write a security automaton whose states represent only the set of events of the target program. There is no way in SAL to perform the general computations necessary to maintain conflict classes and data sets. The only way to specify a Chinese wall policy in SASI is to construct a fixed number of objects and their corresponding datasets, which is, of course, not amenable to the addition of new objects or datasets (see [14]).

It is also difficult to specify this policy in SPoX [20]. Even though SPoX supports general purpose states and supports parameter specifications, only states with integer data types can be represented (it cannot, for instance, represent list states). But a more serious problem in SPoX is that it does not support invocation of methods, for example to check if the accessed object lies in a conflict class or is already in an accessed dataset. SPoX also has no way to update the subject's accessed dataset or conflict class.

PoET on the other hand is more powerful in expressing security policies than SASI or SPoX; it provides global structured security states and allows the addition of states inside a certain class (thereby implementing an implicit parameter). With PoET, then, one can express the Chinese wall policy, but one also needs to define a variable for each subject instance where the depth inside a subject call is tracked. Only when a read is detected inside a subject call the required security checks for the Chinese wall are tested.

PoET [14, 15] and Polymer [6, 24, 25] are able to support the Chinese wall policy, but Polymer has a restriction: it does not support policy parameters. However, since Polymer is an imperative language that supports general purpose code, one may use data structures to keep different instances of the subject in order to emulate parameters. This, however, is error prone. Note also, that PoET would require complicated data structures to monitor policies with more than one parameter.

Naccio [17] cannot express this property because it does not support parameters, and it does not support general purpose code, making it impossible to emulate parameters.

3.5 Policy Composition

Once a number of policies are written and tested separately for correct operation, the next step is to enforce multiple policies concurrently. We refer to this as the *Policy Composition* problem. Regardless of the importance of policy composition, it has not received considerable attention in prior research except for in the Polymer [6, 7, 24, 25] project, which defines a number of *Policy Combinators* to enable different compositions of policies.

JavaMOP, by default, allows multiple policies to coexist within a given target program. However it makes no guarantees on how they will operate together if their events interfere with each other, that is, if they happen to select some of the same program points. By default, which policy will be triggered first when events interfere is decidable only by the order in which AspectJ [23] files are weaved into the program or by using an aspect precedence declaration, which is part of the AspectJ standard. These are, at best, an incomplete way to allow for policy composition. To ensure proper composition, some sort of coordination and management among different policies is necessary, in order to allow precedence as well as conflict resolution.

Consider the following concrete example in JavaMOP that illustrates policies conflict and composition. Suppose that one wants to enforce both the RestrictSystemCalls in Figure 5, and the FileNetworkWall policy in Figure 8. Now consider a situation where the target program accesses a network resource followed by a system call invocation; this violates both policies. Now we have a conflict between skipping the system call and halting.

Generally speaking the options available to JavaMOP handlers can be split into four basic groups: halting the target program, proceeding with the code that triggered the current event, skipping the code that caused the current event, or executing some arbitrary Java code. We will refer to these as **Halt** (Figure 4), **Skip** (Figure 5), **Proceed** (Figure 3, since execution is not stopped or altered instead it is allowed to continue), and **Exec** (Figure 1 since it shows arbitrary Java Code), respectively.

For any two Policies say A and B, JavaMOP creates two monitors[4] at runtime, one for each policy. If Policy A has a **Halt** statement inside the handler and so does Policy B. Then obviously their combined outcome should be to **Halt** the program.

However, a conflict occurs when different operations are used by the handlers of each policy (i.e., conflict between halting or skipping in the above mentioned example).

		A			
		Halt	**Skip**	**Proceed**	**Exec**
B	**Halt**	**Halt**	?	?	?
	Skip	?	**Skip**	?	?
	Proceed	?	?	**Proceed**	?
	Exec	?	?	?	?

Table 1. Combination of behavior of two policies, A and B, with different handlers

Table 1 shows the different categories of operations that a handler might contain. Cells marked with ? indicate a conflict between different handler actions that must be resolved. The types of handler operations can be grouped into a lattice, from the least restrictive to the most restrictive: **Proceed** $\sqsubset$ **Exec** or **Skip** $\sqsubset$ **Halt**.

The idea that we adopt in JavaMOP to resolve conflicts between two or more policies, is that instead of allowing the policies in conflict to execute their handlers statements, they instead delegate their operations to a higher JavaMOP specification. The job of this higher JavaMOP specification is to monitor the execution of other specifications, rather than monitoring the execution of the target, for the sake of coordination among them.

Reflecting that on our concrete example above, we will demonstrate the scenario where one wants to be less restrictive by not halting the target if the file access violating the FileNetworkWall policy is skipped by the RestrictSystemCalls. The first thing we need to do is to modify the handlers of FileNetworkWall and RestrictSystemCalls to call four special methods defined for a Manager class[5]. We show in Figure 9 the updated version of FileNetworkWall [6], the other policy is updated in a similar way.

The second thing we need to do is to create a new higher policy, called ConflictManager, that will monitor calls to methods of the

[4] Note that a given monitor can, as mentioned, contain multiple monitor instances, one for each combination of bound parameters.

[5] The Manager class is a helper class that contains no logic of its own, instead it contains a set of methods that are invoked by different policies where appropriate.

[6] we use the notation of _MONITOR in the handler part to refer to one of the local variables defined in the monitor.

```
import java.io.*;
import java.net.*;

FileNetworkWall() {
 Manager M1 = new Manager();

 event fileAccess before():
 call(* Runtime.exec(..))||
 call(* File.creatNewFile())
 || ... //other file access methods
 {}
 event networkAccess before():
 call(URL.new(..))||call(Socket.new(..))
 || ... //other network access methods
 {}
 ptltl : networkAccess => <*>fileAccess
 @match{ __MONITOR.M1.haltFN(); }

 ptltl: fileAccess => <*>networkAccess
 @match{ __MONITOR.M1.haltNF(); }
}
```

Figure 9. Delegation of the File Network Wall Policy

```
ConflictManager() {
 event haltFN
    after():call(void Manager.haltFN()){}
 event haltNF
    after():call(void Manager.haltNF()){}
 event proceed
   after():call(void Manager.proceed()){}
 event skip
   after():call(void Manager.skip()){}
 ptltl: haltFN => <*>proceed
 @validation{
    System.out.println(
 "FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
 }
 ptltl: haltNF => (*)proceed
 @validation{
    System.out.println(
 "FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
 }
}
```

Figure 10. Conflict Manager

Manager class, then decide whether the system really needs to be halted. The ConflictManager policy is shown in Figure 10.

It defines four events: the haltFN event, which is called by the FileNetworkWall policy whenever it wants to halt after detecting a file access followed by a network access, the haltNF event, which is similar to the above event except it is invoked when the violation pattern of network access followed by file access is detected, the proceed event, which is called whenever there is another file operation (this is done by adding a new event to RestrictSystemCalls), and the skip event, which is called whenever there is a system call that is going to be skipped.

The policy defines the valid trace using PTLTL. Unlike in previous policies, the events are generated by *other policies*, rather than the target program directly. The policy specifies two properties one for each case of violation for the FileNetworkWall. The first property states that, if there is a halt resulting from the pattern of: fileAccess networkAccess, then at least one of the fileAccess events must have been proceeded. While the second property states that, if there is a halt resulting from the pattern of: networkAccess fileAccess then the haltNF event must be preceded immediately by a skip event. Note that the skip event does not appear in the formula; instead, it causes the absence of the necessary proceed event that would have saved the policy from violation. This composition requires that RestrictSystemCalls be given a higher precedence or weaved first, to ensure that skip or proceed will occur immediately before haltFN and haltNF.

Moreover, suppose that we wish to enforce FileNetworkWall and RestrictSystemCalls per user or Subject within the program. To do this, all we need to do is to update the involved policies to declare a Subject parameter on the specification, and to make the events parametric in Subject. Figure 11 shows the necessary changes for the ConflictManager. Similar changes are required for the two composed properties.

```
ConflictManager(Subject S) {
 event haltFN before(Subject S):
   call(void Manager.halt1(Subject))
&& args(S){}
 event haltNF before(Subject S):
   call(void Manager.halt1(Subject))
&& args(S){}
 event proceed before(Subject S):
   call(void Manager.proceed(Subject))
&& args(S){}
 event skip before(Subject S):
   call(void Manager.skip(Subject))
&& args(S){}
 ptltl: haltFN => <*>proceed
 @validation{
    System.out.println(
 "FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
 }
 ptltl: haltNF => (*)proceed
 @validation{
    System.out.println(
 "FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
 }
}
```

Figure 11. Parametric Conflict Manager

4. Discussion

Now that we have seen how JavaMOP can elegantly specify and enforce different security policies that range from the simple to the complex, we devote this section to highlight other security relevant features of JavaMOP.

4.1 Level of Enforcement

Security policies can be expressed and inlined at different levels within a target program, e.g., *application-level* or *object code level*. JavaMOP, Polymer [6, 24, 25] and SPoX [20] use the Java API to mark certain security relevant application-level operations (events) inside the target program for monitoring. Thus JavaMOP should not be used, for example, to specify and enforce low level security policies like memory management and software fault tolerance isolation. Systems like SASI and PoET/PSLang [14, 15] are more suitable to specify such policies since the security relevant operations are specified with ISA instructions, which provide a richer vocabulary to express relevant low level operations. However, it is much more difficult, if not impossible, to use these latter systems to enforce application-level security.

4.2 Class of Policies Enforced

JavaMOP monitors a single execution trace as it occurs, thus it does not enforce policies that depend on observing multiple possible executions. For example, it cannot monitor *Information Flow* policies, which may require checking all possible execution traces. This would imply that JavaMOP enforces policies which are characterized as *properties*, that is, where the set membership for a set of executions is determined by each element alone and not by other members of the set; i.e., other executions [1].

Policies that check that no principal ever has access to a given resource are liveness properties, thus they cannot be expressed by any monitoring system because any partial execution may be extended to a invalid one. However if the availability is bounded to a limit then it can be expressed, e.g., a policy which monitors denials bounded within a set period of seconds [18]. Because of the time period, this property is actually a safety policy that specifies the set of traces in which access is not allowed during the specified time period [30].

And since JavaMOP (with AspectJ) is able to rewrite the target to enforce security policies and to trigger possibly different corrective handlers upon violation of a policy, JavaMOP is categorized as a *Program Rewrite* [21]. And thus JavaMOP can enforce RW-enforceable policies (policies that are enforceable by rewriting the program to an equivalent valid execution) including EM-policies (Execution Monitor policies) and satisfiable static policies [21].

4.3 JavaMOP versus AspectJ

Aspect Oriented Programming (AOP) is well known for best specifying security concerns, due to its modularity and the its ability to seperate concerns and crosscut them where needed. However, as shown in [28] , using AspectJ becomes a hard task when it is used to specify event-history based specifications, or moreover when used to enforce specifications on certain objects of classes (parametrized specifications). That is where a security enforcement mechanism, or more generally a runtime verification system such as JavaMOP, comes into action.

JavaMOP makes implementation details of AspectJ transparent to users (such as declaration of different security states, or creating the appropriate structure to deal with history based events), focusing only on the important parts for their specification to work, leaving the details for JavaMOP to handle.

In this paper we showed examples that support our claim that JavaMOP is a mature and a complete tool that better serves the specification of security policies.

4.4 AspectJ Correctness

As mentioned in the introduction, the process of inlining the generated AspectJ code into the program is outside the boundaries of JavaMOP, and thus JavaMOP makes no guarantees that the actual weaving of the AspectJ file is done correctly.

However, it is worth highlighting that there are some security concerns that arise in the AspectJ-weaved code (for example, in AspectJ5 for a privileged aspects to access private class members it introduces into the target class a public method to get and set the private member) [13]. This is an area of active research [13, 33].

4.5 Restricting Java Reflection

JavaMOP makes no restriction on the code written inside the event action or the property handlers. Additionally, JavaMOP makes no restrictions about using Java reflection within a program. There is a reason for this: JavaMOP is designed to be a general RV system that can be used for many purposes, not necessarily for security. However, since Java reflection can be used maliciously to circumvent the integrity of JavaMOP monitors, it is usually advised, as in Polymer or Naccio, that one should write a specific JavaMOP property that restricts the usage of Java reflection.

```
RestrictJavaReflection(){

Class c;

boolean isJavaMOPClass(java.lang.Class c)
{//check reflected object usage does not
//implement MOPObject interface
}


event fieldAccess
before(java.lang.reflect.Method method):
 call(* java.lang.reflect.Method.*(..))
 && target(method){
  c = method.getDeclaringClass();}

event classAccess
before(java.lang.reflect.Field field):
 call( * java.lang.reflect.Field.*(..))
 && target(field){
  c = field.getDeclaringClass();}

event constructorAccess
before(java.lang.reflect.Constructor constructor):
 call(* java.lang.reflect.Constructor.*(..))
 && target(constructor){
  c = constructor.getDeclaringClass();}


ere: fieldAccess | classAccess
    | constructorAccess

@match{
 if(__MONITOR.isJavaMOPClass(__MONITOR.c))
  System.out.println("Reflection used to access
  javamop class: "+ __MONITOR.c.getName());
      Runtime.getRuntime().halt(1);
 }
}
```

Figure 12. Restricting Java Reflection.

Figure 12 shows part of the policy that restricts the Java Reflection methods. Since all JavaMOP classes must implement MOPObject interface, restricting the usage of Java Reflection is accomplished by checking that any object (i.e., Method, Field or Constructor) using a reflection method does not directly or indirectly reside in a class that implements the MOPObject interface.

5. Evaluation and Performance

This section presents the results of three experiments regarding the performance of policy monitoring with JavaMOP. The first and the second use the DaCapo benchmark suite (version 9.12-bach) [8] and several Java API security policies. The first experiment shows the runtime overhead of monitoring different JavaMOP security policies, while the second experiment shows a comparison of the overheads of JavaMOP, SPoX and Polymer on DaCapo.

The third experiment shows the performance of the ChineseWall policy, which is not relevant to any programs in the DaCapo benchmark suite. All experiments were performed on a machine with 1.00 GB of RAM with a 3GHz Pentium® 4 processor, running Ubuntu Linux 9.10.

	Hidden FileAccess			Disable Network			File Creation			FileNetwork Wall			DisSys Calls		All Policies		
avrora	3	64	64	1	0	0	2	14	14	0	1388	0	1	0	2	1416	56
batik	-1	122	122	1	685	685	-1	0	0	0	1692	685	-1	0	1	2071	1542
eclipse	-1	642	642	1	438	438	2	28	28	1	1958	439	1	0	1	3047	1542
fop	1	121	121	0	0	0	0	0	0	1	667	0	1	0	1	548	49
h2	2	15	15	0	0	0	-1	0	0	1	73	0	-1	0	-1	70	9
jython	0	2726	2726	2	0	0	-1	2	2	1	7347	0	-2	0	-1	10049	2720
luindex	1	25	25	0	0	0	1	256	256	0	24534	0	-1	0	-2	16475	176
lusearch	-2	1549	1549	0	0	0	0	0	0	0	3807	0	0	0	0	3670	1033
pmd	-1	3138	3138	-1	0	0	-1	0	0	-3	6288	0	0	0	-1	7182	2242
sunflow	0	13	13	1	0	0	1	0	0	2	72	0	1	0	0	67	9
tomcat	1	37	37	2	3	3	1	0	0	2	20044	3	1	0	1	14680	39
tradebeans	0	13	13	0	0	0	1	0	0	0	3430	0	0	0	1	3427	9
tradesoap	0	15	15	0	0	0	1	0	0	0	2788	0	0	0	-1	2787	11
xalan	-1	23826	23826	0	1	1	-1	0	0	1	47741	1	-2	0	0	51144	17022

Table 2. DaCapo Results. For each policy, except DisSysCalls, three numbers are shown: average percent overhead within $\pm$ 3%, total number of events monitored, and number of triggered handlers. Sometimes

5.1 Performance Results for JavaMOP

The default input for DaCapo was used, and we use the -converge option to ensure the validity of our test by running each test multiple times, until the execution time converges. After convergence, the runtime is stabilized within 3%, thus average overhead numbers in Table 2 and 3 should be interpreted as $\pm$3%.

We measured the performance of each of the following security specifications separately then we measured the performance when all of them are instrumented simultaneously; the security policies measured are: RestrictHiddenFileAccess, a policy that simply restricts access to hidden files; DisableNetwork, which disallows all network traffic; RestrictFileCreation, which disallows the creation of files; and FileNetworkWall and DisableSystemCalls, which were defined earlier in the paper.

Table 2 shows three numbers for each policy: average percentage overhead within $\pm$ 3%, total number of monitored events, and number of times the handlers were triggered. The DisableSystemCalls shows only the first two numbers since it is a raw JavaMOP specification that does not contain a handler. The AllPolicies in table 2 shows the same figures when all the policies are enforced simultaneously on DaCapo. Note that when the policy's property is monitoring the occurrence of a single event, then the number of times the handlers were triggered is exactly the number of times the events were monitored, i.e., HiddenFileAccess, DisableNetwork and FileCreation.

JavaMOP shows no significant performance overhead when any of the above specifications are enforced on DaCapo independently or when they are all enforced at the same time. We attribute this result to a number of reasons. The first is that these specifications are, from the perspective of JavaMOP's capabilities, relatively simple specifications that are not parameterized. This means that for each policy, a single monitor will be synthesized, creating at most four monitor instances at the same time, when all the policies are enforced simultaneously (a small number of monitors compared to the much heavier workload for which JavaMOP has been optimized). Negative overheads are occasionally possible because additional code introduced by the AspectJ weaving process changes the program structure in DaCapo, sometimes causing the benchmark to run slightly faster due to better instruction cache layout.

5.2 JavaMOP Vs. SPoX Vs. Polymer on DaCapo

In this experiment we instrumented 4 policies: the discussed previously DisableSystemCalls and FileNetworkWall, in addition to LimitOpenedFiles which bounds the number of files being opened to a statically bound variable and NoWriteAfterClose a parametric policy that ensures that no file is being written after it was closed.

SPoX shows also no significant measurements for almost all DaCapo programs, except for fop (which takes an XSL-FO file, parses it and formats it, generating a PDF file), SPoX shows $\simeq$ 16% average overhead. We attribute that due to the fact that fop benchmark makes extensive usage of the file API methods that the policies used in the performance measurement are monitoring. Thus the load of usage of the APIs used in SPoX specifications increases, which increases in turn the crosscutting points and thus requires more weaving to be done.

On the other hand Polymer shows the worst percentage overhead with maximum of 128%. We attribute this to the fact that Polymer did not undergo any performance enhancements: Polymer's concern, while being built, was to ensure the security of the rewriter and the validity of the weaving process rather than the performance of the process.

The last specification NoWriteAfterClose measures the performance overhead for JavaMOP and SPoX for a parametric specification. We did not measure the specification on Polymer, since Polymer does not support parametric specifications. JavaMOP again shows better performance when an extensive usage of the file APIs are being used by DaCapo as in fop (1%), while SPoX shows average performance overhead 16%. For other DaCapo programs no significant overhead difference was observed.

5.3 JavaMOP Chinese Wall Performance

To test the ChineseWall security policy, we have customized a simulation program for stock work flow, where users (subjects) are allowed to access objects in datasets that do not lay in the same conflict class of previously accessed objects.

We tested the program at different numbers of *loads*: running subjects, datasets and conflict classes. We also tested multiple different call depths **(0, 25 and 50)** for the call depth of each subject. Table 4 summarizes the results in three groups of columns. The first group of columns (#Subjects, #Datasets and #Conflict classes) shows the load size of the test performed. It is worth noting that, for each load, a monitor is created for each subject, resulting in at most

	DisableSys Calls			FileNetwork Wall			LimitOpened Files			NoWrite AfterClose	
	MOP	SPoX	Polymer	MOP	SPoX	Polymer	MOP	SPoX	Polymer	MOP	SPoX
avrora	1	3	1	3	2	2	2	2	2	2	1
batik	-1	0	-1	1	-1	-1	-1	1	0	-1	-1
eclipse	-1	1	-8	1	1	-3	0	-5	-1	1	-2
fop	0	16	19	5	17	15	0	16	13	1	16
h2	1	1	-	2	0	-	-1	-1	-	-1	-1
jython	1	-2	-2	-1	-2	2	0	-2	1	-1	-4
luindex	0	3	-1	0	3	14	0	0	13	1	5
lusearch	0	0	0	0	1	2	-1	1	0	1	1
pmd	-2	-2	10	-2	-2	128	-2	-2	39	-1	-2
sunflow	1	0	0	0	0	1	1	0	-1	1	0
tomcat	2	0	7	1	0	126	1	-1	44	2	-1
tradebeans	1	1	2	1	0	1	1	0	1	1	1
tradesoap	0	-1	1	0	1	9	0	1	3	-1	0
xalan	-1	-1	4	-1	-3	63	1	-3	22	-1	-2

Table 3. DaCapo Results. Except for the last policy (which shows only JavaMOP and SPoX average overhead), three average overhead numbers are shown for JavaMOP, SPoX and Polymer respectively.

#Subjects	#Datasets	#Conflict	%Overhead	#Method call/return	#Access	Total events	#Trigger
100	1000	10	6	2000	1000	3000	51500
200	4000	20	9	8000	4000	12000	406000
300	9000	30	5	18000	9000	27000	1363500
400	16000	40	3	32000	16000	48000	3224000
500	25000	50	6	50000	250000	75000	6287500

Table 4. ChineseWall Results. The table shows test load size (#Subjects, #Datasets, #Conflict classes), average percent overhead(%Overhead), number of paired method call and return events monitored (#Method call/return), number of access events monitored (#Access), total number of monitored events (Total #events), and number of triggered handlers(#Trigger).

three hundred monitor instances (the highest load tested) running simultaneously.

The second group shows the percentage performance overhead between the original test program and one instrumented with the ChineseWall policy. As shown in the table, JavaMOP yields low performance overhead, $\simeq 9\%$ at the most, when the target is running a high load of datasets, despite the fact that the test program does very little that is not a policy event. This is shown by the last group of columns which summarizes the number of monitored pairs of method call and return events (#Method call/return), number of monitored access events (#Access), total number of monitored events (Total #events), and, finally, the number of times the handler for the Chinese Wall policy was triggered(#Trigger). The number of access events multiplied by the number of subjects constitutes the upper bound for number of times the handler is triggered. This number is significantly lower because JavaMOP does not create monitor instances for a given Subject until absolutely necessary.

6. Conclusion

In this paper, the relationship between Inlined Reference Monitors (IRMs) and the general purpose Runtime Verification (RV) was explored. We showed that, despite their different backgrounds, they actually overlap in similar functionality. In fact IRM can be considered as a specific instance of RV. We demonstrated how JavaMOP, an RV system, is able to effectively and efficiently specify and monitor security policies.

We discussed how JavaMOP can be used to resolve potential conflicts that might arise when multiple specifications are enforced. This is done by delegating handlers of the conflicting specifications to a higher JavaMOP specification, which in turn decides the appropriate action to be taken.

Finally we presented and discussed the results of our performance experiments for JavaMOP and two other IRM systems mainly SPoX and Polymer.

A formal framework for the composition of JavaMOP specifications is a direction for future research.

References

[1] B. Alpern and F. B. Schneider. Defining Liveness. Technical report, Ithaca, NY, USA, 1984.

[2] C. Anley. Advanced SQL Injection in SQL Server Applications. *NGSSoftware Insight Security Research*, 2002.

[3] P. Avgustinov, J. Tibble, and O. de Moor. Making Trace Monitors Feasible. In *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 589–608. ACM, 2007.

[4] H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma. *Runtime Verification (RV'05)*. Elsevier, 2005. ENTCS 144.

[5] H. Barringer, D. Rydeheard, and K. Havelund. Rule Systems for Runtime Monitoring: from EAGLE to RULER. *J. Logic Computation*, November 2008.

[6] L. Bauer, J. Ligatti, and D. Walker. A Language and System for Enforcing Run-time Security Policies. Technical Report TR-699-04, Princeton University, 2004.

[7] L. Bauer, J. Ligatti, and D. Walker. Composing Security Policies with Polymer. *SIGPLAN Not.*, 40:305–314, 2005.

[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss,

A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, pages 169–190. ACM, 2006.

[9] E. Bodden. J-LO, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.

[10] D. Brewer and M. Nash. The Chinese Wall security policy. In *IEEE Symposium on Security and Privacy (SOSP'89)*, pages 206–214. IEEE, 1989.

[11] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Workshop on Policies for Distributed Systems and Networks (POLICY'01)*, pages 18–38. Springer, 2001.

[12] M. d'Amorim and K. Havelund. Event-Based Runtime Verification of Java Programs. *ACM SIGSOFT Software Engineering Notes*, 30(4): 1–7, 2005.

[13] B. De Win, F. Piessens, and W. Joosen. How secure is AOP and What can we do about it? In *Proceedings of the 2006 international workshop on Software engineering for secure systems*, SESS '06, pages 27–34, New York, NY, USA, 2006. ACM. ISBN 1-59593-411-1. doi: 10.1145/1137627.1137633. URL `http://doi.acm.org/10.1145/1137627.1137633`.

[14] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

[15] U. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy (SOSP'00)*, pages 246–255. IEEE, 2000.

[16] U. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: A Retrospective. *ACM Transactions on Information and System Security*, 3:87–95, 2000.

[17] D. Evans. *Policy-Directed Code Safety*. PhD thesis, MIT, 2000.

[18] V. D. Gligor. A Note on Denial-of-Service in Operating Systems. *IEEE Trans. Softw. Eng.*, 10:320–324, May 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010241.

[19] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational Queries Over Program Traces. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 385–402. ACM, 2005.

[20] K. W. Hamlen and M. Jones. Aspect-Oriented In-lined Reference Monitors. In *Workshop on Programming Languages and Analysis for Security (PLAS'08)*, pages 11–20. ACM, 2008.

[21] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability Classes for Enforcement Mechanisms. *ACM Transactions on Programming Languages and Systems*, 28:175–205, 2006.

[22] K. Havelund and G. Roşu. *Runtime Verification (RV'01, RV'02, RV'04)*. Elsevier, 2001, 2002, 2004. ENTCS 55, 70, 113.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object Oriented Programming (ECOOP'01)*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[24] J. Ligatti, J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *Journal of Information Security*, 4:2–16, 2003.

[25] J. A. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, 2006.

[26] M. Martin, V. B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 365–383. ACM, 2005.

[27] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An Overview of the MOP Runtime Verification Framework. *Journal on Software Techniques for Technology Transfer*, 2011. to appear.

[28] P. H. Phung and D. Sands. Security Policy Enforcement in the OSGi Framework Using Aspect-Oriented Programming. In *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, COMPSAC '08, pages 1076–1082, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3262-2. doi: 10.1109/COMPSAC.2008.149.

[29] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63:1278–1308, 1975.

[30] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and Systems Security*, 1:30–50, 2000.

[31] O. Sokolsky and M. Viswanathan. *Runtime Verification (RV'03)*. Elsevier, 2003. ENTCS 89.

[32] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Runtime Verification (RV'05)*, volume 144 of *ENTCS*, pages 109–124. Elsevier, 2005.

[33] B. D. Win, W. Joosen, and F. Piessens. AOSD Security: A Practical Assessment. In *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT03*, pages 1–6, 2003.