Through the Lens of Code Granularity: A Unified Approach to Security Policy Enforcement

Shamaria Engram Jay Ligatti
Department of Computer Science and Engineering
University of South Florida
Tampa, FL, USA
Email: {sengram,ligatti}@usf.edu

***Abstract*—A common way to characterize security enforcement mechanisms is based on the time at which they operate. Mechanisms operating before a program's execution are static mechanisms, and mechanisms operating during a program's execution are dynamic mechanisms. This paper introduces a different perspective and classifies mechanisms based on the granularity of program code that they monitor. Classifying mechanisms in this way provides a unified view of security mechanisms and shows that all security mechanisms can be encoded as dynamic mechanisms that operate at different levels of program code granularity. The practicality of the approach is demonstrated through a prototype implementation of a framework for enforcing security policies at various levels of code granularity on Java bytecode applications.**

***Index Terms*—security mechanisms, enforcement, policies**

I. Introduction

Security mechanisms enforce policies on untrusted programs and are traditionally classified based on whether they operate statically or dynamically. Some policies, such as type safety, can be enforced either way. However, important tradeoffs exist between static and dynamic enforcement.

Static mechanisms (e.g., static type checkers or virus scanners) analyze all of a program's source [1], intermediate (e.g., [2]), or binary code (e.g., [3]) before its execution. Such mechanisms allow policy violations to be caught before code is put into production without inhibiting a program's runtime performance. However, it is undecidable to determine statically whether an arbitrary program satisfies a nontrivial policy [4]. Consequently, static code analzyers typically enforce policies conservatively and sometimes report false positives [1].

Dynamic, or runtime, mechanisms (e.g., firewalls or operating systems) monitor program events during a program's execution and intervene as necessary. These mechanisms can enforce some policies more precisely than static mechanisms because of available runtime information. However, mechanisms that only monitor the current execution are limited in the types of policies they can enforce [5]. Other runtime mechanisms [6] can enforce a larger class of policies when they can access auxiliary program information (e.g., results of a static analyzer). Nevertheless, dynamic mechanisms generally sacrifice runtime performance for precision.

Hybrid mechanisms combine a static and dynamic approach to policy enforcement and can enforce policies that are difficult to enforce by just a static or dynamic mechanism alone. Some program rewriters—mechanisms that modify programs prior to their execution to ensure policy satisfaction [7]—can be viewed as a particular type of hybrid that is able to perform static code analysis and inline checks to enforce policy-specific constraints at runtime for statically undecidable policies.

The static-dynamic view of security mechanisms is important for providing a foundation for characterizing the class of policies enforceable by both static and dynamic mechanisms. However, this perspective presents a problem in addressing two research questions:

1) Which policies are precisely enforceable by hybrid mechanisms? A precise characterization of the class of policies enforceable by such mechanisms has been difficult to derive because different proof techniques are used for the static and dynamic components [8].
2) Do additional mechanisms exist, beyond: static code analyzers, runtime mechanisms, and hybrids?

A first step to answering these questions might be to reason about security mechanisms more uniformly by casting existing mechanisms into a single framework. This paper introduces an alternative perspective for classifying security mechanisms based on the granularity of code that they monitor. The approach unifies existing classes of security mechanisms by encoding them as runtime mechanisms that operate at one or more levels of code granularity. The contributions of this paper are as follows:

1) We present a general model of security mechanisms parameterized by the granularity of program code that they analyze and show that this model unifies existing classes of security mechanisms (Section III).
2) We demonstrate the practicality of the model with a prototype implementation for enforcing security policies on Java bytecode applications. The implementation is based on a Java library, called JaBRO, that we have developed to enable runtime code analysis at various levels of code granularity (Section IV).
3) We evaluate the effectiveness and efficiency of the implementation by enforcing security policies on two popular, open-source applications at various levels of code granularity (Section V).

II. Related Work

Prior work on the unification of security mechanisms within a single framework is limited. Nonuniform runtime mechanisms, introduced in [6], have auxiliary knowledge about a program, such as the results of a static code analyzer that guarantees whether certain program events will occur. Formal characterizations of the class of policies enforceable by nonuniform mechanisms, with various capabilities, were presented in [9], [10], [11]. Such mechanisms can enforce policies that uniform runtime mechanisms—mechanisms that only monitor the current execution and have no a priori knowledge about the program—cannot enforce. However, the nonuniform runtime mechanism model is not general enough to capture all existing classes of security mechanisms because they do not have access to a program's code.

It was shown in [7] that any statically enforceable policy is enforceable by a runtime mechanism if code analysis can be performed immediately after the program is loaded. However, [7] did not explore the implications of this claim or whether runtime mechanisms can improve the precision at which statically undecidable policies may be enforced.

Hence, we conclude from prior work that runtime mechanisms can be viewed as generalizations of static mechanisms when runtime mechanisms can access a program's code, thus providing a basis for a unified approach to policy enforcement.

III. A Unified Approach

This paper encodes all security mechanisms as runtime mechanisms that operate at one or more levels of program code granularity, where the granularity is determined by well-defined, modular program constructs. This section presents a general model of runtime mechanisms and defines levels of granularites at which mechanisms may operate.

A. Mechanisms

Mechanisms in this paper are runtime mechanisms that enforce policies by intercepting security-relevant events just before they are about to execute. To capture realistic behaviors of security mechanisms, we model them as automata that can permit, deny [5], modify [7], suppress, or insert [12] program events based on the rules of a policy.

Definition. *A security mechanism is an automaton $M = (I, O, Q, Q_0, \delta)$, where I is the set of possible inputs to the mechanism, O is the set of possible outputs from the mechanism, Q is a countable set of automaton states, $Q_0 \subseteq Q$ is a countable set of initial automaton states, and δ is a deterministic or nondeterministic transition function. A deterministic δ is a function $\delta : Q \times I \to Q \times O$, and a nondeterministic δ is a function $\delta : Q \times I \to 2^{Q \times O}$.*

An input $i \in I$ to or an output $o \in O$ from a mechanism can be any well-defined program construct. For example, Figure 1 shows how an example Java program consists of different granular program constructs. At the coarsest granularity are whole programs, which decompose into packages, packages decompose into classes, classes decompose into methods, and methods decompose into fine-grained program instructions.

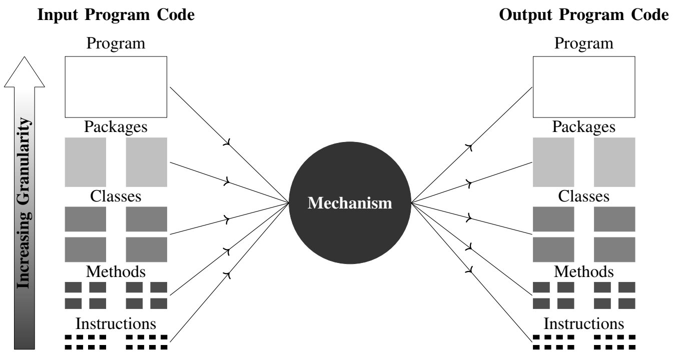


Fig. 1: A general security mechanism capable of inputting and outputting program code at any level of granularity.

B. Fine-Grained Policy Enforcement

Program instructions and values are at the finest level of granularity. Instructions might be low-level assembly or micro-code instructions, or high-level language program statements (e.g., method invocations or assignment statements). Values can be primitive or reference types or a collection of values stored in a data structure.

A *fine-grained policy* can be enforced at runtime by monitoring individual events as they are attempted by the target program and do not require mechanisms to access the program's code. For example, consider the policy that prevents target programs from writing to files using `FileWriter.write()` after reading from files using `FileReader.read()`. A run-time mechanism enforcing this policy can monitor the method invocations attempted by the target program. If the program attempts to execute `FileWriter.write()` after executing `FileReader.read()`, the mechanism can enforce such a policy by outputting `System.exit(1)` to halt the target program. It is important to note that the mechanism only monitors the invocations of the `read()` and `write()` methods and does not analyze the code body of these methods; therefore, a mechanism enforcing such a policy in this way operates at a fine-grained level.

Fine-grained policy enforcement is beneficial when satisfaction of the policy depends on runtime values. For example, consider the above policy except programs can now write to files after a file read if and only if the name of the file being written to is `log.txt`. Assuming the file name is only available at runtime, the policy can be enforced at a fine-grained level because a mechanism only needs to examine the value of the object instance invoking the `write` method (i.e., the name of the file bound to the `FileWriter` object).

Mechanisms operating at this level cannot, in general, precisely enforce hyperproperties [13], which are policies that may require relationships to hold between different executions of a program. This limitation of fine-grained mechanisms exists because such mechanisms have no knowledge of other possible program executions when only monitoring program events as they are attempted during the current execution.

C. Medium-Grained Policy Enforcement

Medium-grained policies exclusively capture all well-defined, modular language constructs in between fine-grained program statements and whole programs, and can be enforced by analyzing the code body of such constructs at runtime. For example, consider an application-specific policy that states "if a resource is acquired within an application-defined method then the resource must be released before the end of the method's execution" [14]. This policy can be enforced statically by analyzing every method's code body, but it can also be enforced at runtime by: 1) intercepting the invocation of every application-defined method, 2) fetching its associated class file, in the case of Java, and 3) performing code analysis on the method's body. The benefit of enforcing this policy at runtime over enforcing the policy statically is to more accurately determine the potential execution paths of the program. For example, consider the following method:

```
public void writeSubString(int index,String s){
    FileWriter writer = null;
    try{
        writer = new FileWriter(
            new File("substrings.txt"));
        if(index < 10){
            String newS = s.substring(0,index);
            writer.write(newS);
            writer.close();
        } else{
            String newS = s.substring(index);
            writer.write(newS);
        }
    } catch(IOException e){
        if(writer != null){writer.close();}
    }
}
```

In this method, the `FileWriter` object is not closed in the `else` branch. A sound static analyzer will take into account all possible executions of this method and report that the `FileWriter` object may not be closed. Assuming that the source code cannot be simply edited, because it's not available, the policy may be enforced more precisely at runtime if the mechanism can determine which branch will be taken.

Suppose `writeSubString(5,"Hello, World!")` is attempted by the program. Knowledge of the runtime values allows a medium-grained mechanism to determine that the `if` branch may be taken and, if so, the `FileWriter` object will be closed; therefore, the method can proceed to execute. Now assuming `writeSubString(10, "Hello, World!")` is attempted, the mechanism can then determine that the `else` branch may be taken and that the `FileWriter` object may not be closed. The mechanism can still allow the method to proceed but, if needed, insert an event to close the `FileWriter` object just before the method returns.

Runtime code analysis allows for more precise policy enforcement—as opposed to static code analysis—by security mechanisms when they can determine probable execution paths based on runtime values. However, to enforce policies at this level of granularity, policy writers must be familiar with low-level code when source code is not available.

D. Coarse-Grained Policy Enforcement

Coarse-grained policies can be enforced by performing code analysis on a whole program's source, intermediate, or binary code at runtime. Static code analyzers also analyze whole programs but cannot take into account runtime information.

Static code analyzers can be encoded as runtime mechanisms by implementing them to intercept program execution at a program's entry point. For example, in many high-level languages (e.g., Java, C/C++, and Rust) the entry point is at the `main` method. If a program takes command line arguments, these arguments may be used to predict the execution paths of the program. If the arguments can be used to rule out a number of execution paths, then coarse-grained mechanisms can be more precise than statically operating mechanisms.

E. Hybrid Policy Enforcement

Hybrid policies take into account code at two or more levels of granularity. Traditional hybrid mechanisms combine static code analysis and runtime enforcement. These types of mechanisms can be encoded as pure runtime mechanisms by monitoring coarse- and fine-grained program events. However, our approach to hybrid policy enforcement is more general because arbitrarily many levels can be combined.

Mechanisms that enforce policies exclusively at coarse- or medium-grained levels may not always be able to precisely predict control flow when medium-grained events are deeply nested within coarse- and other medium-grained events (e.g., nested method calls). However, hybrid runtime mechanisms can overcome this limitation by refining policy enforcement during a program's execution. For example, program rewriters can be encoded as hybrid runtime mechanisms by intervening at a program's entry point for coarse-grained code analysis. If code modification or inlined checks are necessary, the mechanism can output a new version of the program with the necessary modifications. A mechanism operating in this way is similar to runtime mechanisms that replace unsafe library function calls with safe versions (e.g., [15]), except the hybrid runtime mechanism outputs a coarser-grained program event, namely a safe program, with inlined checks for medium- or fine-grained events. If the mechanism cannot decide how to respond when a medium-grained event is reached during execution, perhaps due to nested method calls depending on runtime information, the mechanism can further refine policy enforcement by outputting a new medium-grained event with inlined checks to collect runtime information when the nested method calls are reached.

IV. Implementation

This section presents details about our implementation and explains how granular policies are enforced at runtime.

TABLE I: Possible policy granularities enforceable at each AspectJ join point type.

AspectJ join point type	Coarse	Medium	Fine
Main method execution	✓	✓	✓
Non-main method execution		✓	✓
Method call		✓	✓
Constructor call		✓	✓
Static-initializer execution		✓	✓
Object pre-initialization		✓	✓
Object initialization		✓	✓
Handler execution		✓	✓
Advice execution		✓	✓
Field reference			✓
Field assignment			✓

A. Architecture

The implementation is composed of 3 core components: AspectJ, a code analyzer, and JaBRO. AspectJ is an aspect-oriented Java language extension that allows policy writers to define *aspects* [16], similar to Java classes. Aspects contain *pointcuts*, which capture one or more *join points*, and *advice*. "Join points are well-defined points in the execution of the program" [16, p. 329]. Table I lists the join point types allowed by AspectJ and the possible policy granularities enforceable at each join point. The `main` method execution is the only join point that can be used to enforce a coarse-grained policy because it is the entry point of the program. Any join point that has a body of code provides the capability for enforcing medium-grained policies, and every join point provides the capability for enforcing fine-grained policies, because fine-grained policies do not require code analysis. Advice is a block of code to be executed before, after, or instead of a pointcut.

The code analyzer is code written by the policy writer to analyze Java class files (e.g., data or control flow analysis). This analysis ensures that a body of code adheres to the security policy in question.

JaBRO (Java Bytecode Rewriter and Optimizer) is a library that we have developed to extend the functionality of AspectJ to enable code analysis on optimized bytecode at runtime. It is composed of Javassist [17] and Soot [18], which are tools for altering and optimizing Java bytecode, respectively. We have made JaBRO available online [19].

B. Policy Enforcement

Coarse- and medium-grained policies are composed of an aspect, JaBRO, and a code analyzer. Fine-grained policies only consist of an aspect because they do not require code analysis. To monitor the target program, the AspectJ compiler is used to weave the policy into the target program, producing a self-monitoring program as shown in Figure 2.

Figure 3a illustrates how coarse- and medium-grained policies are enforced during a monitored program's execution:

1) A security-relevant event is intercepted by a pointcut.
2) Event context information, which includes the event name, its enclosing class name, and runtime arguments, and the program's dependencies are input to JaBRO, which is invoked from inside of the advice.

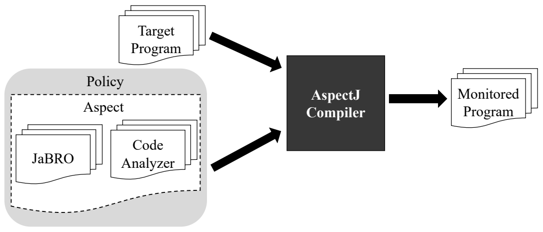


Fig. 2: Weaving a policy into a target program.

- AspectJ provides the capability to obtain the event context information through the join point construct.
- The program's dependencies are needed to resolve type information during the optimization process.

JaBRO uses the event's context information to obtain the class file that the event is declared in from the system search path. JaBRO then rewrites the event to include its runtime arguments within the event's code body. If the policy is coarse grained, JaBRO optimizes all of the program's class files. If the policy is medium grained, JaBRO only optimizes the class file that the event is declared in.

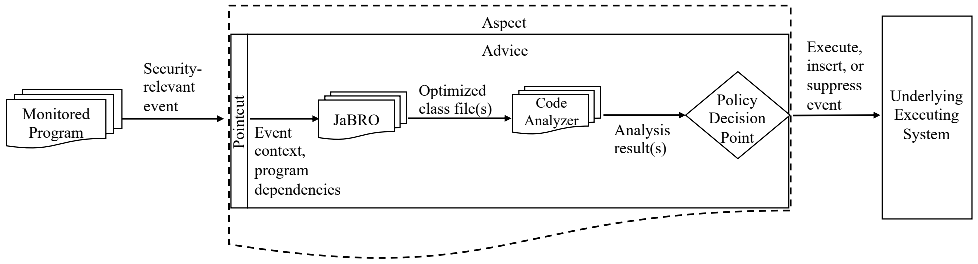
3) Code analysis is performed on the optimized class file(s), and an analysis result is output.
4) The policy decision point decides to execute, suppress, or insert an event based on the analysis result.

To illustrate the importance of JaBRO, consider the `writeSubString(5,"Hello, World!")` method call presented in Section III. After obtaining `writeSubString`'s enclosing class file, JaBRO first rewrites the method in the following way:

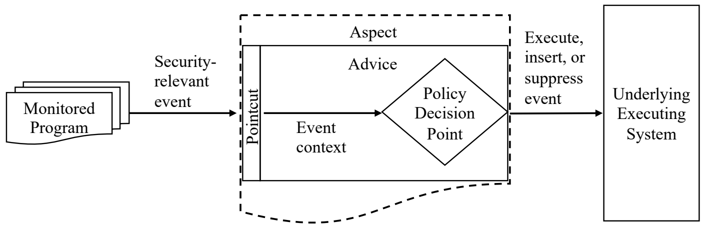
```
public void writeSubString(int index, String s){
    int index = 5;
    String s = "Hello, World!";
    FileWriter writer = null;
    try{
        writer = new FileWriter(
            new File("substrings.txt"));
        if(index < 10){
            String newS = s.substring(0,index);
            writer.write(newS);
            writer.close();
        } else{
            String newS = s.substring(index);
            writer.write(newS);
        }
    } catch(IOException e){
        if(writer != null){writer.close();}
    }
}
```

After rewriting, JaBRO propagates the arguments throughout the method and optimizes it in the following way:

```
public void writeSubString(int index, String s){
    FileWriter writer = null;
    try{
        writer = new FileWriter(
            new File("substrings.txt"));
        String newS = "Hello, World!"
            .substring(0,5);
        writer.write(newS);
```



(a) Coarse- or medium-grained policy



(b) Fine-grained policy

Fig. 3: Runtime policy enforcement

```
            writer.close();
        } catch(IOException e){
            if(writer != null){writer.close();}
        }
}
```

Given the runtime values, JaBRO is able to eliminate the `else` branch. A policy stipulating that file resources acquired in a method be released before the end of the method's execution can be enforced more precisely by performing code analysis on the optimized method rather than on the original.

Figure 3b illustrates how policies can be enforced at a fine-grained level. Similar to coarse- and medium-grained policies, security-relevant events are intercepted by pointcuts defined in the aspect. The policy decision point can then make an enforcement decision based on event context information.

V. Empirical Evaluation

To evaluate the effectiveness of the implementation, we enforced three security policies on two popular, open-source Java applications. The first application was the US National Archives and Resource Administration's (NARA) file analyzer and metadata harvester [20], which analyzes and collects metadata on files. The second application was JPlag [21], which is a software plagiarism detection tool. Both applications conduct operations on files and have the ability to access potentially sensitive information on a system, which permitted us to express practical policies.

We implemented policies to cover each level of granularity (i.e, coarse, medium, and fine). The coarse-grained policy disallowed network connections by prohibiting the use of `java.net.Socket` to ensure that neither application could exfiltrate sensitive information over the network. The medium-grained policy required file resources acquired within a method to be released before the end of the method's execution [14]. This policy was enforced to ensure that the applications could not deplete system resources. The fine-grained policy required files containing sensitive information to be hidden from the applications. This policy was enforced by checking whether the values of file operations (i.e., file names) were contained in a sensitive file list.

To evaluate the overhead introduced by the implementation, we measured the average time to weave and enforce each policy. Results are shown in Table II. The AspectJ compiler was used to weave each policy into each target application as shown in Figure 2. The average execution time of each program event was measured without enforcing the policy, indicated by the columns labeled unmonitored, and with enforcing the policy, indicated by the columns labeled monitored. The NARA file analyzer and metadata harvester application required user interaction; therefore, instead of measuring the total execution time of the application, we measured the time to enforce the coarse-grained policy, for the monitored column, and the time to start-up the application, for the unmonitored column. Each policy was enforced separately on each application. Each experiment was conducted 100 times on a MacBook Pro laptop with a 2.9 GHz Quad-Core Intel Core i7 processor and 16 GB of RAM.

Summary of Results

Each policy was successfully enforced on both applications. The results in Table II indicate that the applications' execution times were significantly impacted by the enforcement of the coarse- and medium-grained policies, which is due to the execution time of JaBRO and the code analyzer. At a coarse-grained level, the entire application must be traversed twice: first, JaBRO must rewrite the `main` method and optimize the entire application, and second, code analysis must be conducted on the entire optimized application. The overhead introduced by coarse-grained policy enforcement is likely unacceptable for time-sensitive applications. However, for

TABLE II: Average experimental performance results of 100 runs.

Application	Policy								
	Coarse			Medium			Fine		
	AspectJ weaving (s)	Unmonitored (ms)	Monitored (s)	AspectJ weaving (s)	Unmonitored (ms)	Monitored (s)	AspectJ weaving (s)	Unmonitored (ms)	Monitored (ms)
NARA File Analyzer and Metadata Harvester	2.29	11.35	33.54	2.91	0.26	1.01	2.91	0.10	0.11
JPlag (v2.11.8)	2.59	392.01	32.55	3.58	0.39	1.69	2.52	0.02	0.03

security-critical applications where runtime performance is not a concern and static analysis is too conservative, the approach may be advantageous. The overhead introduced by medium-grained policy enforcment may be more acceptable due to the smaller code fragment traversed by JaBRO and the code analyzer. The fine-grained policy only added 0.01 ms to the execution time of the fine-grained event for both applications.

We expected coarse-grained policies to be weaved significantly faster than medium- and fine-grained policies because the policy only needs to be weaved at a single point (i.e., the `main` method); however, the AspectJ compiler cannot differentiate between policy granularities and thus searched through the entire application looking for all possible matches to the `main` join point. The weaving time for coarse-grained policies may be improved by implementing a custom bytecode rewriter tailored for weaving coarse-grained policies.

VI. Conclusions and Future Work

We have presented a unified approach to security policy enforcement by encoding existing classes of security mechanisms as runtime mechanisms that operate at one or more levels of code granularity. A new taxonomy of security policies was presented based on the granularity of code that mechanisms analyze to enforce policies. The practicality of the unified approach was demonstrated through a prototype implementation that extends the AspectJ language extension with our JaBRO library, which enables runtime code analysis on optimized bytecode at various levels of code granularity.

Possible directions for future work include: 1) an in-depth theoretical analysis of the class of policies enforceable by runtime mechanisms operating at one or more levels of code granularity and 2) domain-specific policy-specification languages for granular security policies based on various language constructs, which may enable the synthesis of new kinds of runtime security mechanisms.

Acknowledgments

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1746051 and the Alfred P. Sloan Foundation University Center of Exemplary Mentoring 2013-5-13MPHDT.

References

[1] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[2] F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. G. de Aledo, "Skink: Static analysis of programs in llvm intermediate representation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 380–384.

[3] J. Kinder and H. Veith, "Precise static analysis of untrusted driver binaries," in *Formal Methods in Computer Aided Design*. IEEE, 2010, pp. 43–50.

[4] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[5] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[6] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of non-safety policies," *ACM Transactions on Information and System Security*, vol. 12, no. 3, p. 19, 2009.

[7] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 1, pp. 175–205, 2006.

[8] B. P. Rocha, M. Conti, S. Etalle, and B. Crispo, "Hybrid static-runtime information flow and declassification enforcement," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 8, pp. 1294–1305, 2013.

[9] H. Chabot, R. Khoury, and N. Tawbi, "Extending the enforcement power of truncation monitors using static analysis," *Computers & Security*, vol. 30, no. 4, pp. 194–207, 2011.

[10] Y. Mallios, L. Bauer, D. Kaynar, and J. Ligatti, "Enforcing more with less: Formalizing target-aware run-time monitors," in *International Workshop on Security and Trust Management*. Springer, 2012, pp. 17–32.

[11] S. Pinisetty, V. Preoteasa, S. Tripakis, T. Jéron, Y. Falcone, and H. Marchand, "Predictive runtime enforcement," *Formal Methods in System Design*, vol. 51, no. 1, pp. 154–199, 2017.

[12] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: Enforcement mechanisms for run-time security policies," *International Journal of Information Security*, vol. 4, no. 1-2, pp. 2–16, 2005.

[13] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[14] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," in *Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, 2005, pp. 365–383.

[15] A. Baratloo, N. Singh, T. K. Tsai *et al.*, "Transparent run-time defense against stack-smashing attacks." in *Annual Technical Conference*. USENIX, 2000, pp. 251–262.

[16] 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*. Springer, 2001, pp. 327–354.

[17] S. Chiba, "Javassist: Java bytecode engineering made simple," *Java Developer's Journal*, vol. 9, no. 1, 2004.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[19] S. Engram, "JaBRO," https://github.com/shamaria/JaBRO, 2020.

[20] US National Archives and Resource Administration, "NARA file analyzer and metadata harvester," https://github.com/usnationalarchives/File-Analyzer, 2016.

[21] "JPlag - detecting software plagiarism," https://github.com/jplag/jplag, 2015.