Inferring Java Security Policies through Dynamic Sandboxing

Hajime Inoue
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
hinoue@cs.unm.edu

Stephanie Forrest
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
forrest@cs.unm.edu

***Abstract*— Complex enterprise and server-level applications are often written in Java because of its reputation for security. The Java policy language allows users to specify very fine-grained and complex security policies. However, this expressiveness makes it difficult to determine the correct policy with respect to the principle of least privilege. We describe a method for automatically learning the minimum security policy called *dynamic sandboxing*. A minimal sandbox (security policy) is inferred by observing program execution and expressed in the standard Java policy language. The minimum policy stops Java exploits and learning the policy does not cripple performance, allowing applications to run normally during training.**

Keywords

Computer security, security policy, policy inference, Java, policy language, principle of least privilege.

I. Introduction

Java has become the platform of choice for enterprise and server level applications, in part because of its security features. Java provides robust security mechanisms for cryptography, verification, and access control. In early Java versions, security policies were specified by implementing the Java security classes themselves. Since Java 1.2, however, Java has supported a special-purpose language for specifying security policies. This paper describes a technique for automatically inferring the policy for an application, simply by observing its behavior during execution. The minimal sandbox (most restrictive security policy that allows the application to run) is learned by our method and expressed in the standard human readable Java policy language. We call the general approach of inferring sandbox policy through learning *dynamic sandboxing*. The paper describes dynamic sandboxing, discusses its advantages, and compares its strengths and weaknesses with other related approaches.

The Principle of Least Privilege (or Least Authority) was first described by Saltzer and Schroeder in “The Protection of Information in Computer Systems” [21] and continues to be a prominent principle of computer security. It states that “programs should operate using the least set of privileges necessary to complete the job.” This is useful because it limits the amount of damage that can occur due to malicious subversion or through simple bugs.

There is a tradeoff, however, between the ease of stating a policy and its granularity. A binary policy (all-or-nothing access) is easy to state but is not a good representation of the security requirements of most applications. A more precise description would necessarily be more complex, following the evolution of the Java standard. Its original security model divided programs into applets and applications. Applets had a highly restrictive policy, while applications had no restrictions at all. Later versions of Java introduced finer grained security models to allow a closer fit between requirements and privileges. Predictably, the expressiveness of the current policy language makes it difficult to understand the exact privileges an application requires.

Indeed, Java’s current security mechanism supports highly precise policies [12]. The sandbox supports almost any imaginable policy if one reimplements the security classes, while the policy language is very expressive in terms of the granularity of resources it can represent, although it does not support arbitrary algorithms. The policy language is essentially a mapping of resources to code. A sandbox is configured by granting specific Permissions to code. The ability to execute a protected operation depends on the set of granted Permissions and other details such as the origin of the code and the identity, if any, of its digital signers.[1] A collection of classes, signed or not, constitutes a “protection domain,” the basic unit of Java security.

[1]“Permission” is capitalized when we refer to subclasses of the Java class java.security.Permission.

The Protection Domain itself can be viewed as a sandbox, since each Domain has an attached set of Permissions, which constitutes its policy. Domains can interact, but protected actions are allowed if all Domains include the relevant Permission. A common case of this occurs when application code calls standard library code, for example, when manipulating files. In this case, there are two domains: the system domain, which includes all the standard libraries, and the application domain. The application domain can open a file if it is given that permission—the system domain is given all permissions. Thus Java platform security can be viewed as a set of interacting sandboxes.

There are several different subclasses of Permission. Each can be constructed with a number of names denoting a specific operation. For some the name indicates the permission, others are annotated with a set of allowed actions. For example. The permission specified by

```
permission java.io.FilePermission
      "/etc/passwd", "read";
```

denotes the ability to read the password file. Clearly, a Protection Domain's policy can be fine-grained. The default Java security policy file is 48 lines long including comments.

Java's security rests on its ability to verify bytecode. Java bytecode is a typed assembly language that allows the Virtual Machine (VM) to determine the types of all operands. That, combined with runtime checks, allows the VM to prevent bad casts and buffer overflows (see ref. [12] for a detailed description of Java's security mechanisms.). However, even with this infrastructure, many vulnerabilities have been reported, including those found by the Princeton Secure Internet Programming Team [5] and by individual vendors [2], [1]. Java's security faults can be classified into three main categories:

- **VM Bugs** Early Java VMs had several verifier errors, which allowed bad casts, leading to potential subversion of the VM. This class of bugs has declined in recent VMs, because Java bytecode verification is now well understood and the implementations are mature.
- **Errors in the Standard Libraries** Sensitive operations need to be protected by checking Permissions. If these checks are omitted, then applications can potentially access resources that are denied by their policy. These are a growing problem because the size of the Java libraries is increasing quickly (See Figure 1).
- **Policy Misconfiguration** Reports of incorrect policies are infrequent due to their site specific nature. Security policies are necessarily tailored to the individual application and host. However, there have been some reports of sandbox misconfiguration for applets [3]. Policy misconfiguration is likely to be a continuing problem, as long as they are configured manually. Like configuring network firewalls, Java security policy specification requires the ability to integrate knowledge of the host, network, and application.

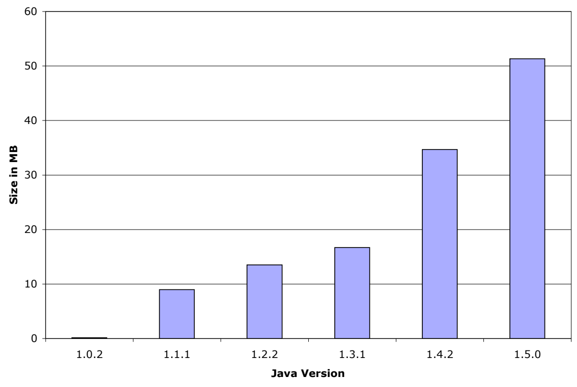

Fig. 1. The Java standard libraries are growing with each release. Each resource within the library must be properly protected or a malicious program may exploit it.

We believe that specifying a policy in accordance with the principle of least privilege is difficult for even moderately complicated Java programs and will become more difficult as additional Permissions are added. Today, even specifying a reasonably common policy seems too complicated. For example, Sun's VMs by default do not initialize the Security Manager [12].

Inferring the minimum security policy is useful. In earlier work, we showed that a minimal sandbox was effective when Java method signatures were treated as the resource to be controlled [16]. The system learned which methods were needed for an application to run under normal conditions. It then restricted the application program to using only those methods and was able to prevent security violations, even when the faults were present but unexercised during training. This paper extends that work beyond method signatures by treating Domains and Permissions as features.

In the remainder of the paper, we first describe our method for learning minimal policies and evaluate its efficacy and performance. We then discuss the results, comparing them with our previous results and other related work.

II. Policy Inference

Our goal is to automatically infer (learn) the minimum required permissions for each required domain in a given

```
public void checkPermission(Permission p)
{
    if (recursive()) return;
    if (training)
        foreach(ProtectionDomain d)
            if (!Policy.getPolicy().implies(d, p))
            {
                writePolicy(d, p);
                Policy.getPolicy.refresh();
            }
    super.checkPermission(perm);
}
```

Fig. 2. Pseudocode for the `checkPermission()` method of `SecurityManager`. If the check call is initiated within this method, return. Otherwise, if training is activated, then add the policy to each protection domain, rewrite the policy file, and refresh the policy. Then check the Permission.

application program. We do this empirically during a training phase in which we run the program and record all Permissions that are not implied by the current policy. We start with a policy that grants no permissions. We then add those Permissions to the current policy in the policy language provided by Java.[2] The policy file at the end of the run is the record of all Permissions required for that run. In subsequent runs, the policy file is used to enforce the inferred sandbox.

We implemented Dynamic Sandboxing using a custom Security Manager to log all calls to `checkPermission()`. Within `checkPermission()`, we wrote a private method to determine if the required Permission is implied by the current policy. The individual `check` methods that do not take Permission arguments construct Permissions internally with the appropriate arguments and then call `checkPermission()` directly. Recursive calls, required for determining whether a Permission is implied by a domain, are recognized by walking the execution stack and then suppressed. Calls to `checkPermission()` use the appropriate execution context if it is provided.If it is not provided, we add the Permission to each Protection Domain in the current execution context, rewrite the policy file, and refresh the policy. Figure 2 presents the pseudocode for `checkPermission()`.

The implementation required two new classes for training: the custom Security Manager and an application launcher to install the Security Manager. Subsequent runs require no special code. Training runs are initiated with the command:

```
java
-Djava.security.policy=<application>.policy
DSLauncher <application>
```

and runs with the inferred policy are invoked the usual way:

```
java -Djava.security.manager
-Djava.security.policy=<application>.policy
<application>
```

where *`<application>`* is the name of the class to be executed. Arguments to the class are appended to the end in the usual way. The policy file need not exist because all necessary Permissions are inferred automatically and added to the policy file. If the policy file exists, the system policy is initialized with that policy. This is helpful in tuning existing policies.

A. Experiments

To be practical, the implementation needs to be effective at stopping attacks, experience few false positives, and be efficient to run. First, we explore its effectiveness at stopping attacks, describing experiments against four exploits. Next we discuss false positives, and finally we report timing runs against the SPEC Java benchmark suite to assess efficiency.

1) Exploits: We tested the Dynamic Sandbox against four exploits: a modified form of **StrangeBrew** [18], [25], **BeanHive** [26], **Port25** [29], and **HttpTrojan**, which we developed ourselves. The original exploit for StrangeBrew does not function for Java versions later than 1.0, and we do not have source copies of BeanHive and Port25. We implemented our own versions of these three exploits based on their published descriptions.

StrangeBrew was the first virus targetting Java applications and is still the only effective one released into the wild. When invoked, StrangeBrew searches its current directory for uninfected class files. For each uninfected class, it adds a copy of itself to the class and modifies the parameter-less constructor to call itself. It then pads the length of the file by inserting null operations until it is a multiple of 101 so it can identify a class's infection status without opening it.

BeanHive was the second virus found for Java. It is interesting because most of its code is not stored in the infected `.class` files. Instead, a small amount of virus code downloads helper classes that enable the virus to search through the current directory, infecting any uninfected class files. It adds the infection stub to the end of all constructors. We wrote a reliable version of BeanHive using the Apache BCEL libraries [4].

Port25 and HttpTrojan are not viruses. Port25 is a short class file that tests whether it can connect to an outside server. In effect, it probes the current policy to determine if it is constrained by the applet sandbox. Its

[2]We ignore the differences between signed and unsigned code for these experiments. For signed code, we could simply recognize the signatories and output them with the policy.

most likely role would be as part of a trojan because it performs no function other than the probe. The HttpTrojan is an exploit we developed. It is a simple HTTP server that has an undocumented backdoor, which allows the user to execute arbitrary commands on the remote machine.

We tested the first three exploits with a small host program that either reads or writes to a specified file. A sample workload was generated that reads and writes several files in the current directory, reads /etc/passwd, and reads and writes some files to /tmp. We generated a policy file for the workload and confirmed that no policy violations occur when running identical workloads. Then an infected version of the host program was run using the same inputs used during training. All experiments used Sun's Java 1.4.2.

The Dynamic Sandbox detected and stopped policy violations of three out of the four exploits. StrangeBrew was the only failure, and it failed because StrangeBrew's behavior is so unambitious—it runs and modifies code only in the current directory. If the virus were more comprehensive (e.g., by ensuring that its code is called in other functions or by recursively searching the directory structure for Java code) then the dynamic sandbox would detect a policy violation.

In BeanHive, the Dynamic Sandbox identified the creation of the URLClassLoader as a policy violation. Because it could not download its infection code, BeanHive failed. Similarly, Port25 failed because the inferred policy does not include the permission to resolve the www.netscape.com address.

HttpTrojan is perhaps the most interesting of the exploits, because it is a fully functional server application that includes a large amount of network and file manipulation. We trained it by browsing a series of pages from the authors' web sites, making sure that we hit all usual error states (pages not found, etc.). When tested, the backdoor code failed to execute because the inferred policy denied all files permission to execute.

2) False Positives and Generalization: False positives are an important issue for any scheme that is trained on finite samples of behavior. A false positive in this context would correspond to a legitimate behavior of the program that violates the minimal security policy. In particular, we are concerned about false positives that arise from insufficient training examples or from configuration changes that necessitate revising the policy. The host application used for the first three test exploits required no tuning (zero false positives) because it only manipulated files in a small number of directories.

The more realistic exploit, HttpTrojan, did require tuning for false positives. In its current implementation, the Dynamic Sandbox grants file permissions by filename and action. Thus, the training runs must browse all files that will be available during testing. For websites containing many files this could become cumbersome, so we manually edited the policy file to permit access to any file or directory within a specified base directory. Because the sandbox is expressed in the Java policy language, it is straightforward to make these changes. Similarly, the sandbox assigns network permissions by IP address and port. We relaxed the permissions to include all ports above 1024 and all IP addresses within our university. We foresee that similar tuning would likely be required for many large applications.

Tuning the policy to remove false positives is often a matter of generalization. In the HttpTrojan, we noticed that Permissions to most of the files used the same base directory so we generalized the permission to allow reads to all files in that directory. We similarly generalized the ports and IP addresses. Although we did this manually, it would be easy to automate the process. In the future, simple heuristics for each type could be defined to determine if sets of Permissions should be coalesced into more general ones. To date, we have implemented generalization heuristics for Socket and File permissions and verified that they work. In addition to reducing training times, generalization could also be used to minimize the size of policy files. A network application that explicitly listed every port and incoming IP address would incur significant performance penalties. Thus, generalization can be used to prevent the size of the policy file from exploding during training.

Unfortunately, it is difficult to generalize the generalization. Heuristics must be tailored for each Permission type and even then there is no guarantee that the heuristic is sound. The policy languages we discuss in Section III were designed so that the policy could be mechanically analyzed, but Java's policy language does not allow that. The advantage of writing a human-readable policy comes from the necessity of occasionally manually editing policy code.

False positives arise in part from the Principle of Least Privilege and are not specific to our system (overly restrictive firewalls are a common example). Anytime a policy creates restrictions, especially one that is learned empirically, there is a risk of false positives. They will be rare when the policy is much less restrictive than the needs of the application. We believe that the convenience and protection of Dynamic Sandboxing is a reasonable tradeoff for false positives that can be repaired through generalization or other heuristics.

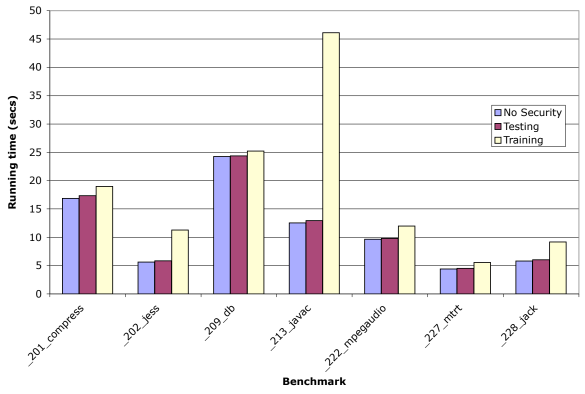

Fig. 3. The performance of the SPEC JVM98 benchmarks under no SecurityManager, while generating a policy (training), and running with that policy (testing).

3) Performance: We ran three sets of experiments using the SPEC JVM98 benchmark suite [24]. We report three sets of numbers for reach benchmark in the suite: *no security*, *training*, and *testing*. No security times the application running without a Security Manager (the Java default). Training measures the total application time when runs are inferring a policy (without generalization heuristics). Testing measures the total application time during runs in which the policy is enforced. Experiments were conducted on a PC running Linux 2.4.21 with a 1.7Ghz Xeon processor and 1 GB of memory.

Figure 3 shows the running times of each benchmark under the thee different conditions. The performance penalty for running with our the sandbox in place is modest, averaging about 2%. This is notable because our inferred policies are typically much longer than the standard Java policy. The penalty for training is greater, although this is a one-time cost. It ranges from 4% for *db* to 267% for *javac* and averages 34% (this shrinks to 24% if *javac* is excluded.) The training time arises from computations to keep the profile parsimonious and by writing and then reading the file whenever a Permission is added. More efficient training regimes may be possible, although writing out the policy as it changes could benefit the administrator; he or she could decide what tuning or generalization is necessary as the application is running.

We conclude that performance is not a major issue. Training security policies for applications can be viewed as part of the installation and configuration process. Running with security enabled produces few performance penalties at runtime. The training cost, while significant, would likely be reasonable for most of the applications we envision, which are often I/O bound. In future work, we plan to investigate optimizations to the training procedure.

B. Comparison to the Original Dynamic Sandbox

In previous work we used Dynamic Sandboxing to study other features of execution [16], [17]. For example, we studied methods, method sequences, object types, and object lifetimes. These features are collected by most modern VM's, for example, to control the level of optimization or the type of garbage collection algorithm. In the context of security, Dynamic Sandboxing by methods performs similarly to using Permissions. However, if methods are included as a sandbox resource, then the StrangeBrew virus is blocked as well as the other three exploits.

There are other differences in the two systems' behavior, however. First, our earlier system only sandboxed one resource, methods, while our current work includes all the resources that are explicitly Permission checked. Second, our previous system required VM modifications; it could not be implemented in pure Java without significant performance penalties. Those modifications provided benefits, however. Our current system is vulnerable to VM bugs and some library bugs. Dynamic Sandboxing by methods is orthogonal to the standard security apparatus and prevents many of those faults.[3] Finally, our current system produces short, human readable profiles of normal behavior. The older system used large files of method signatures placement in the profile had a large impact on performance.

We believe that the two systems are complimentary. Each is able to prevent security faults the other would tolerate. Together, they cover the entire space of security faults: VM bugs, library bugs, and policy bugs.

III. Related Work

Much of the previous research on Java security has matured and been incorporated into the Java platform. The integration of type safety research with the stack introspection work of Wallach [30] developed into the current scheme of interacting protection domains [13]. These improvements make Java highly resistant to the types of security vulnerabilities, like buffer overflows, that are seen in other software. However, Java still suffers from vulnerabilities outlined in Section I.

Two active areas of computer security are directly relevant to this paper: languages and reasoning for policies and anomaly detection.

[3]Dynamic Sandboxing by methods ensures that methods not listed in the normal profile are never compiled and thus can never be executed. This is "policy as mechanism" [6]. A set of mechanisms that allows a flexible policy, like Java provides, is more likely to be subverted.

There are several general policy languages that have been proposed and several that are in use. Some examples are KAoS [7], Ponder [11], Rei [19], and Condell's SPSL [10]. In a paper comparing some of these languages, Tonti and his coauthors describe five features of policy languages: expressiveness, simplicity, enforceability, scalability, and analyzability [27]. Using these criteria, the Java policy language is not a general language, and thus succeeds brilliantly in expressiveness, simplicity, enforceability and scalability, because it is tailored directly to the Java platform.[4] Java's language fails in analyzability because Permissions are not distinct sets, making it difficult to reason about; some Permissions imply others and this information is available only within the bytecode itself.

There are other tools that learn policies. Configuring firewalls is in many ways analogous to configuring Java security policy and there are several tools available. A good example is a system by Burns [8]. There is at least one other tool, SPECTRE, for dynamically inferring security policies and specifying that policy in a standard language [22]. Its inference policies are specific to web services, however. Lam and Chiueh describe a system called *Paid* which limits application behavior to a statically calculated system call sequences [9]. They refer to this as "Automatic Extraction" of sandboxing policy. Its approach is more similar to other host-based anomaly detection systems than to our approach. Their system protects a new resource, system call-order, and then infers a policy. Our system outputs a standard policy file for resources that are already protected. More similar to our system is Naumovich's work on consistency in J2EE security policies [20]. They statically analyze the methods invoked by different roles (groups of users) to suggest inconsistencies in policy. J2EE policies are in a different policy language (XML) and are converted to Java policies automatically. Their approach addresses a different level of security and is based on source analysis rather than on observed behavior.

Although it is not usually associated with security policy research, anomaly detection is in fact a way to learn or infer policies. In anomaly detection, a set of specified features of execution are examined during a program's run. The values of these features are stored as a profile of normal behavior. This is called training. If these values are different during subsequent runs, the run is deemed anomalous. The profile of normal behavior embodies the inferred security policy. Anomaly detection is rarely perfect, and false positives can be an issue. These are actions that should be allowed but were not learned during training. Thus, it is usually the case that inferred policies are too restrictive, although in a security setting this is generally preferable to policies that are too loose.

[4]The performance of Java's security is debatable. Applications pay a measurable performance cost for security. Improving the performance of the security mechanisms has been an active research area but is outside the scope of this paper [13], [28].

There are several anomaly detection systems written in Java, but few are aimed at detecting anomalies within Java itself. The most interesting one is DIDUCE, a debugging aid [14]. It is useful strictly for development since performance is very poor. Our group has studied anomaly detection systems in a variety of setting, including operating systems[23], TCP networks [15], and the Java VM [16]. As we described above, our first Dynamic Sandboxing system inferred a security policy for a single resource: methods. The work presented here can be viewed as a way for our algorithm to view several different resources.

IV. Conclusions and Future Work

In this paper we described the first practical system to infer minimum security policies for Java applications. We showed that it is both effective and efficient at deriving and enforcing policies. These policies, while not perfect, form a useful basis for hand-tuning. Editing policies is familar to administrators because the policy format is the Java standard. Our implementation of Dynamic Sandboxing uses Permissions as the feature from which to construct the sandbox. The work is part of a larger, ongoing exploration of execution features available within virtual machines. We believe that more research will reveal some of these features, like methods were, to be useful in novel applications for optimization or security.

Our implementation is still at the prototype stage, and we think the performance of the custom security manager can be improved significantly. Generalizations in our system could also be improved and better integrated into the policy inference mechanism.

Beyond these incremental improvements, we see two interesting additions. First, we are interested in adding the ability to explicitly state Permissions that cannot be granted. This could be stated in an *anti*-policy that uses the usual policy syntax. The Dynamic Sandbox Security Manager would refuse to add these Permissions during training and inform the operator of a fault. Sun has contemplated adding such a mechanism but has thus far refused due to its greater complexity [12]. Second, we would like to extend this work beyond Java. Microsoft's .NET virtual machine includes a security infrastructure similar to Java's. Should .NET become ubiquitous, as it should with the release of Longhorn, a similar system to the one developed here would be useful.

Acknowledgements

The authors gratefully acknowledge support of the National Science Foundation (grant ANIR-9986555, CCR-0219587, CCR-0085792, and EIA-0218262), the Defense Advanced Research Projects Agency (grant AGR F30602-00-2-0584), the Intel Corporation, and the Santa Fe Institute.

We thank the Adaptive Computation and OAL groups at the University of New Mexico for help in developing and refining these ideas.

References

[1] Chronology of security-related bugs and issues. http://java.sun.com/sfaq/chronology.html.

[2] Microsoft security home. http://www.microsoft.com/security/.

[3] Ms02-069: Flaw in microsoft vm may compromise windows. http://support.microsoft.com/kb/810030/EN-US/, 2002.

[4] Apache. Bcel - byte code engineering library. http://jakarta.apache.org/bcel/manual.html.

[5] Andrew Appel and Edward Felton. Princeton secure internet programming. http://www.cs.princeton.edu/sip/history.

[6] Gabriela Barrantes. *Automated Methods for Creating Diversity in Computer Systems*. PhD thesis, University of New Mexico, 2005.

[7] J.M. Bradshaw, S. Dutfield, P. Benoit, and J.D. Woolley. *Software Agents*, chapter KAoS: Towards an Industrial Strength Generic Agent Architecture, pages 375–418. AAAI Press/MIT Press, 1997.

[8] J. Burns, A. Cheng, P. Gurung, S. Rajagopalan, P. Rao, D. Rosenbluth, A. V. Surendran, and D. M. Martin Jr. Automatic management of network security policy. In *DARPA Information Survivability Conference and Exposition (DISCEX II '01)*, volume 2, June 2001.

[9] Lap chung Lam and Tzi cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[10] M. Condell, C. Lynn, and J. K. Zhao. Security policy specification language (spsl). Internet Draft.

[11] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proceedings of the Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, Bristol, UK, 2001. Springer-Verlag.

[12] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 2 edition, 1993.

[13] Li Gong and Roland Schemers. Implementing protection domains in java development kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998. Internet Society.

[14] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

[15] Steve Hofmeyr. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.

[16] Hajime Inoue and Stephanie Forrest. Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the New Security Paradigms Workshop 2002*. ACM Press, 2002.

[17] Hajime Inoue, Darko Stefanovic, and Stephanie Forrest. On the prediction of java object lifetimes. Submitted to IEEE Transactions on Computers, 2004.

[18] Landing Camel Intl. Codebreakers-4. http://www.codebreakers.org, 1998.

[19] L. Kagal. Rei: A policy language for the me-centric project (hpl-2002-070). Technical report, HP Labs, 2002.

[20] Gleb Naumovich and Paolina Centonze. Static analysis of role-based access control in j2ee applications. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[21] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63(9), pages 1278–1308, September 1975.

[22] D. Scott and R. Sharp. Spectre: A tool for inferring, specifying, and enforcing web-security. Technical report, Cambridge University, 2002.

[23] Anil Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, University of New Mexico, 2002.

[24] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[25] Symantec. Security response: Javaapp.strangebrew. http://securityresponse.symantec.com/avcenter/venc/data/javaapp.strangebrew.html.

[26] Symantec. Security response: Java.beanhive. http://securityresponse.symantec.com/avcenter/venc/data/java.beanhive.html.

[27] G. Tonti, J. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A Uszok. Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In *Proceedings of the International Semantic Web Conference (ISWC 03)*, Sanibel Island, Florida, 2003.

[28] Dana Triplett. Spotlight on java performance: Distinguished engineer robert berry highlights key areas of research and development at ibm. http://www-106.ibm.com/developerworks/java/library/j-berry/, 2001.

[29] VirusList.com. not-a-virus: Javaclass.port25. http://www.viruslist.com/en/viruses/encyclopedia?virusid=62347.

[30] D. Wallach, D. Bafanz, D. Dean, and E. Felten. Extensible Security Architecture for Java. In *Proc. 16th ACM SIGOPS Symp. on Operating Systems Principles*, volume 31:5, pages 116–128, Saint Malo, France, 1997.