A Benchmark Suite for Behavior-Based Security Mechanisms

Dong Ye, Micha Moffie and David Kaeli
Computer Architecture Research Laboratory
Northeastern University, Boston, MA 02115
{dye,mmoffie,kaeli}@ece.neu.edu

Abstract

This paper presents a benchmark suite for evaluating behavior-based security mechanisms. Behavior-based mechanisms are used to protect computer systems from intrusion and detect malicious code embedded in legitimate applications. They complement signature-based mechanisms (e.g., anti-virus software) by tackling zero-day attacks whose signatures have not been added yet to the signature database, as well as polymorphous attacks that have no stable signatures.

In this work we present a benchmark suite of eight programs. All of these programs are legitimate applications, but we have designed them to be infected by malicious software. An evaluation framework is designed to infect, disinfect, build, and run the benchmark programs. This benchmark suite aims to help evaluate the effectiveness of various behavior-based defense mechanisms during different development stages, including prototyping, testing, and normal operation. We use this benchmark suite to evaluate a simple behavior-based security mechanism and report our findings.

1 Introduction

1.1 Behavior-based security mechanisms

Many host-based intrusion prevention systems [29, 34, 38] employ behavior-based analysis to protect an application running on a server from being hijacked. Most of these applications are known or highly suspected to horde security vulnerabilities, such as buffer overflows and format strings [21]. These systems use various methods to examine the actions taken by a program by inspecting library API activity and system calls. Actions that appear malicious, such as attempting a buffer overflow or opening a network connection in certain contexts, will trigger an alarm by the monitoring agents.

Over the past few years, spyware has become a pervasive problem [13, 16]. Many infections occur when spyware is piggybacked on top of popular software packages. Saroiu et al. [16] found that spyware is packaged with four of the ten most popular shareware and freeware software titles from C|Net's `http://download.com/`. Commercial security software vendors [28, 35, 30, 37] have developed a number of security products addressing this problem. All of these companies have emphasized that they detect spyware by observing system behavior and detecting abnormal activity from the norm.

Signature-based intrusion detection and anti-virus solutions fail to expose this class of exploitation and do not adapt well to even small changes in an exploit. A signature is a regular expression known a priori that matches the instruction sequence of the exploitation or the network packets presented in a specific attack [39]. Therefore, zero-days attacks that have not had a signature extracted yet, as well as polymorphous attacks, pose a great danger to these signature-based mechanisms. Behavior-based mechanisms aim to overcome these shortcomings and complement signature-based mechanisms with more adaptive and proactive protection. Instead of looking for fixed signatures in instruction sequences and network packet payloads, behavior-based approaches focus on detecting patterns at a higher level of abstraction. Ideally, the patterns are the inherent behavior associated with malicious activities and distinct from the normal behavior of legitimate programs. Evading a behavior-based protection mechanism normally requires a change in the logic of the malicious activity itself.

Gao et al. [6] investigated the design space of system-call-based program tracking, which is the technology behind many host-based anomaly detection and prevention systems. A detailed system call trace can be recorded and characterized to better understand the typical behavior of the program. By establishing a profile of normal behavior, an intrusion into the process will be detected when the system-call behavior deviates from this normal profile.

Edjlali et al. [4] presented a history-based access-control mechanism to mediate accesses to system resources from mobile code. Their idea was to maintain a selective history of the access requests made by individual programs and to use this history to differentiate between safe and potentially dangerous requests. Each program is categorized into one of several groups, whereas each of these groups contains a different profile of resource requests. The behavior of each program during the entire execution is also constantly monitored. The decision of whether to grant a resource request that the program makes depends on both its preassigned identity and its historical behavior during this execution, as well as additional criteria, such as the location where the program was loaded or the identity of its author/provider.

1.2 Security metrics and measurement

As behavior-based mechanisms become more commonly used, and the rules and analytics engine underlying these mechanisms become more sophisticated, we need a methodology to evaluate these security mechanisms. The evaluation could be (and, ideally should be) used both for testing these mechanisms during code development, and for the validation and product rankings.

Developing metrics to define security properties remains an ongoing research topic [5]. A number of approaches have been proposed to measure the value of a security product or technology, and to assess the level of security attained by the whole system [20, 32].

Kajava et al. [11] considered a range of criteria to qualify and quantify the degree of security attained. They summarized three major classes:

- *Risk analysis* is the process of estimating the possibility of individual exploitations, their impact on the system, and as well as the cost to mitigate the risk. Risk analysis considers the trade-off between cost and the level of protection, and is thought to be a good basis for any security evaluation [3].
- *Certification* involves decomposing the system into different classes based on design characteristics and security mechanisms. Standards organizations and commercial companies provide certification services to measure the level of confidence that can be assigned to the security features offered by a product [41, 31], or the degree of conformance of a security process to the established guidelines (e.g., ITIL [14], CMM [10] and COBIT [1]).
- *Penetration testing* provides statistics about the probability that an intrusion attack will be successful. For example, the WAVES project [42] standardizes the practice of penetration testing for Web applications.

There have also been efforts to employ multiple orthogonal criteria to quantify the value of the perceived security enhancement, and the cost associated with the enhancement. Gordon et al. [7] proposed a framework to use the concept of *insurance* to manage the risk of doing business in the Internet era. They also described how to evaluate and justify security-related investments. The criteria they used for their security evaluation includes the three elements just discussed.

There still remains no widely accepted way to measure and rank security properties. The difficulty of finding a common ground for evaluating various security mechanisms suggests that further work is needed before we can adopt an unified evaluation methodology for different categories of security mechanisms.

The goal of this paper is to describe a new benchmarking methodology to evaluate behavior-based security mechanisms. We present a benchmark suite composed of eight applications that are typically found in workstation/desktop environments. These applications are infected with a variety of malicious codes, that in turn, represent a broad spectrum of exploits. We demonstrate the utility of our benchmark suite by applying it to a simple behavior-based security mechanism. The rest of paper is organized as follows. We discuss the rationale of our benchmarking methodology for evaluating behavior-based security mechanisms in section 2. We then describe the suite of benchmarks we have created in section 3. In section 4, we use this benchmark suite to evaluate a simple behavior-based security mechanism and analyze the results. In section 5, we summarize the paper and discuss future directions for our work.

2 A Case for Benchmarking Behavior-Based Security Mechanisms

Benchmarking has been used widely in the field of computer architecture and system software development to evaluate the performance of a particular design or implementation. The basic idea behind benchmarking is to create a common ground of comparison for a certain category of targets. Normally a suite of applications is constructed to serve as this common ground. These applications reflect typical workloads running on a selected category of computer systems (e.g., servers) or a selected category of application software (e.g., database). The value of different design mechanisms is measured by obtaining performance metrics while running the suite. Benchmarking promotes the practice of quantitative analysis [8]. There have also been efforts to use benchmarking to evaluate properties other than performance, such as dependability [12].

One of the key challenges addressed by most security-related mechanisms is that they need to address a moving target. The activities and scenarios that may do harm to the system are unpredictable, and tend to change their form. It would seem that a benchmarking methodology might not be a good choice for evaluating security mechanisms, since there is no stable workload that can be used.

In spite of the differences between their various approaches, all the behavior-based mechanisms make a common claim that they can differentiate the behavior of the malicious code from the normal behavior of the program. Malicious behaviors are limited to several general categories, such as resource abuse, information tampering, and information leakage [16]. More and more of these attacks are being motivated to obtain financial gains [17]. This indicates that the malicious behavior that these mechanisms are trying to single out is limited, and is relatively stable. For these cases, benchmarking can be very useful. A benchmark suite that consists of representative workloads infected with representative malicious activities can provide a good test of behavior-based security mechanisms.

Our benchmarking approach diverges from the penetration testing either performed by third-party auditors and certification service providers [41, 31], or embodied in software packages which are composed of a set of penetration cases [42]. These differences include:

- The main purpose of penetration testing is to find security vulnerabilities in the targeted programs, while the goal of our benchmarking technology is to find out whether the analytics and rules behind behavior-based mechanisms are sufficient.

- Penetration testing can be very implementation specific. Whenever a exploit of a newly discovered vulnerability appears, this new penetration scenario must be added to the set of test cases. On the contrary, the collection of malicious behavior included in our benchmark suite is much less dependent upon individual exploits. Unless the entire strategy behind an exploit is different from those included in the benchmark suite, there is no need to update the benchmark suite with every newly discovered exploit.

- Last, our benchmarking methodology is complementary to commonly used audit and certification services. Designers and developers can benefit from our benchmark suite because it is more cost-effective and convenient to use to test new ideas and prototype products during the entire development cycle.

The anti-virus community has already tested the idea of benchmarking. Basically they combine the signatures of all the known (and some not widely known) exploits and see how many of them different anti-virus products can find. In a test performed by `Virus Bulletin` [40], 100% of their signatures were detected by all the tested anti-virus software. It should be apparent that it would be difficult to produce a meaningful comparison here. A 100% detection rate suggests that benchmarking may not be a good way to evaluate detection accuracy (i.e., effectiveness) of anti-virus technology.

Using our approach, we emphasize that it is behavior-based mechanisms that we propose to evaluate using benchmarking. Different types of security mechanisms may need different methods to be properly evaluated.

3 The SecSpec Behavioral Benchmark Suite

3.1 Components of the benchmark suite

We have developed a benchmark suite called *SecSpec*. The benchmark programs included in the suite, as well as the malicious code, are written in Java. The choice of language should not limit the scope of applying the benchmarking methodology, though the implementations of malicious behavior may need to be ported to another language and a new set of benchmark programs may need to selected.

We target a typical workstation/desktop computing environment when choosing the component programs for the benchmark suite. We include four types of applications and consider two particular programs from type.

Browsers: Jbrowser [24] and JXWB [26] are two simple and functional web browsers. They are simple because they do not possess elaborate features such as client-side plug-ins.

Editors: Jedit [33] and Jexit [25] are two full-blown editors. The feature richness of these two applications pose a great challenge to behavior-based security mechanisms.

Instant Messengers: BIM [23] and SimpleAIM [27] are two simple AOL instant messaging clients. SimpleAIM is console-based and BIM is GUI-based. Instant Messaging (IM) has become a serious application in both enterprise and personal desktop environments, and is also a favorite medium for spyware distribution [15].

Games: Computer games are a major channel for viruses to infect both enterprise and home desktops. Even games developed for mobile phones can be be infected with viruses [2]. We include two simple games, Tetris [36] and AntiChess [22], to cover this category of applications.

In our suite, we cover five categories of malicious code. We arrive at this categorization based on the behaviors they present. Each category of malicious behavior includes one or more implementations. Table 1 lists our categorization of these malicious behaviors.

We have placed the implementations of the malicious behavior inside a single source file for easy maintenance. Different types of malicious behavior are implemented in separate functions. The execution of a particular malicious behavior is simply a call to the corresponding function(s).

Specially-formatted comments are placed in the source code of the benchmark programs. These special comments are placeholders for the invocation of malicious behavior. To infect (or disinfect) the benchmark programs, we simply uncomment (or comment) these placeholders.

Malicious behavior type	**Implementation(s)**
1. Direct information leakage	Read local file and email out.
2. Indirect information leakage	Copy local file to user's webpage directory.
	Copy local file to /tmp.
	Change file permission bits.
3. Information tampering	Update .hosts file in home directory.
4. Direct resource abuse	Write a huge file to current directory.
	Crash a process.
5. Indirect resource abuse	Download remote code, put in the system startup folder or update system startup script.

Table 1: Categorization and implementation of malicious behavior

3.2 Placement of malicious code inside benchmark programs

The location of malicious behavior inside a benchmark impacts the accuracy of behavior-based security mechanisms. When invoking malicious code at different locations, the malicious behavior will appear in different contexts. If we place the invocation of the malicious code such that it presents a similar library API call or system call profile as in the original application, the behavior-based mechanism will face a bigger challenge to do its job well. Previous studies [18, 43, 6] have demonstrated the viability of the mimicry attacks against host-based intrusion prevention systems. They engineered the attack code to confuse the detection agent by limiting the usage of library APIs and system calls to those that are also used by the application.

This could lead to a practice of choosing the location of the placeholders inside the benchmark program according to the similarity between the malicious code and the context of the benchmark program around the placeholders. However, we have focused on capturing more general application behavior instead of worrying about mimicing a specific low-level library API and system call profile. Our goal is not to defeat these security mechanisms, but instead, to evaluate their effectiveness. We want to measure the robustness of the logic and rules sets underlying these mechanisms when encountering potentially confusing information. We call this practice orthogonality-directed placement. The less orthogonal the malicious behavior and the surrounding context of benchmark are relative to one another, the larger the challenge that this benchmark suite poses to behavior-based mechanisms.

Different placement schemes demand different levels of understanding of benchmark programs. The minimum level of understanding is to make sure the insertion of placeholders does not break the original code. We have experimented with two placement schemes:

Random placement: Beyond the minimum requirement of not breaking benchmark programs, our random placement makes sure that the malicious code will appear in at least two types of locations: at a location where it will definitely appear on the execution path; and at a location where it may or may not appear on the execution path, depending on some particular run time events. We position the placeholders in the startup or termination section to emulate the first scenario and in the user interface event handling section to emulate the second scenario.

Orthogonality-directed placement: This requires us to compute the degree of similarity of the program behavior and the malicious behavior. Our approach is to classify both the benchmark programs and malicious code to obtain four general categories of behavior: network oriented, file system oriented, mixed or neither. We then mix them together according to the extent of overlap between behaviors in these four categories.

Among the four types of benchmark programs, we classify IM clients as network-oriented, editors as file system-oriented, browsers as mixed, and games as neither. Among the five types of malicious behaviors, we classify indirect information leakage, information tampering, and direct resource abuse as file system oriented, direct information leakage and indirect resource abuse as mixed.

		Malicious behavior				
Benchmark programs		1	2	3	4	5
Browsers	Jbrowser [24]	Δ				
	JXWB [26]	Δ				
Editors	Jedit [33]		Δ	Δ	Δ	
	Jext [25]		Δ	Δ	Δ	
IMs	BIM [23]					Δ
	SimpleAIM [27]					Δ
Games	AntiChess [22]	Δ	Δ	Δ	Δ	Δ
	Tetris [36]	Δ	Δ	Δ	Δ	Δ

Table 2: Placement of malicious code in applications

An example of an orthogonality-directed placement would look like Table 2. Note that the numbering of the malicious behavior corresponds to the numbering given in Table 1. All of the placeholders are inserted manually.

3.3 User interface of the benchmark suite

The user interface to the benchmark suite is provided via the Apache Ant build tool [19]. We provide four build targets for each benchmark program:

1. Infect: Insert malicious code into a benchmark program by uncommenting the placeholders in the source code.

2. Disinfect: Restore a benchmark program to the clean version by commenting out these placeholders.

3. Jar: Build a single jar file of a benchmark program, including all the class files, supporting files, as well as the library package that implements the malicious behavior.

4. Run: Run a benchmark program, generating the command line and running the benchmark program.

4 Experimentation

4.1 A History-Based Access Control

To test our benchmark suite, we have implemented a history-based access control mechanism based on the work done in [4]. This is an example of a behavior-based security mechanism.

The basic idea of this mechanism is that a running program is constantly categorized into a series of contexts according to the resource requests it makes during execution. Each context includes a number of Java permissions [9] which could permit access to the guarded resource. This series of contexts is the historical profile of the program and determines whether the future resource request should be granted or rejected.

The relationship between different contexts are either cooperative or non-cooperative. A policy file explicitly specifies the cooperative relationship. Permission to a new resource request can be granted only under one of the following two scenarios:

- The program's historical profile already includes a context that contains this permission,

- The context that needs to be added to grant this permission must be held in a cooperative relationship with the program's historical profile.

We have implemented a simple version of the history-based access control. More sophisticated mechanisms can be implemented in a similar way. However, this simple mechanism helps us to locate where the problem is when this it succumbs to an exploit.

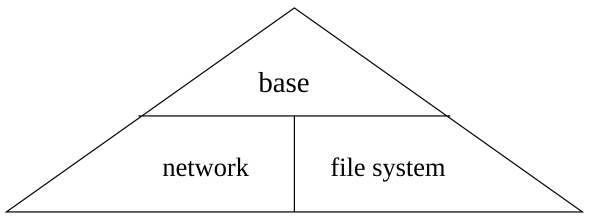

Figure 1: Contexts provided in a history-based access control mechanism.

The mechanism we implemented has three contexts: base, network, and file system, as shown in Figure 1. The base context includes the most restrictive permissions, network and file system grant all network-related and all file system-related permissions, respectively, which are thought of as resources susceptible to attack.

When a resource access request is made, the base context is searched first for permissions that could imply allowing this access. Whenever a permission in the base context can service the need, two things will happen: the base context will be added to this program's historical profile; and the search process stops, even if permission in either the network or the file system context may also allow this access.

Figure 2 shows an outline of the policy file for this simple history-based access control mechanism. Note the priority of the `base` context over the `network` and `file_system` contexts is indicated by the fact that the specification of the `base` context precedes the other two in the policy file.

4.2 Evaluation

We carried out our experiment in two stages: (1) first profiling clean benchmarks; (2) testing the security mechanism against infected benchmarks.

During the profiling stage, a clean version of each benchmark is run once. We have modified the security manager to intercept all resource requests. Permissions that are required to run a clean benchmark are granted and recorded. We then create the policy file for the history-based access control mechanism. We organize the gathered permissions into the base context, and try to make some too permissive permission more fine-grained, in order to minimize the risk exposure of the base context. We make sure the clean version of each benchmark can run without having to be categorized into either a network or file system context.

During the testing stage, we run the infected version of each benchmark. The security manager is loaded upon the startup of the JVM and uses the policy file established from the profiling stage to apply history-based access control.

```
context base
{
    permission java.net.SocketPermission "vanders.ece.neu.edu", "resolve";
    permission java.net.SocketPermission "localhost:*", "connect,listen, resolve";
    permission java.net.NetPermission "specifyStreamHandler", "";
    permission java.io.FilePermission "/home/student/dye/.jedit/-", "read,write,delete";
    permission java.lang.reflect.ReflectPermission "*", "";
    permission java.awt.AWTPermission "*", "";
    permission java.lang.RuntimePermission "*", "";
    permission java.util.PropertyPermission "*", "read,write";
    permission java.util.logging.LoggingPermission "control", "";
    ....
};
context file_system
{
    permission java.io.FilePermission "<<ALL FILES>>", "read,write,execute,delete";
};
context network
{
    permission java.net.SocketPermission "*", "connect,listen,accept,resolve";
};
CooperatingContexts
{
    file_system
    base
};
CooperatingContexts
{
    network
    base
};
```

Figure 2: Skeleton of the policy file for the history-based access control mechanism.

Table 3 shows our experimental results. In this experiment, we randomly placed the five types of malicious behavior inside each benchmark program.

Before running this experiment, we anticipated that holes in Java permission could cause trouble for our security mechanism. Also, we suspected that the permissions gathered in the profiling stage are not fine-grained enough (i.e., we may be too permissive). The analysis of our testing results confirmed our suspicions. In addition, we uncovered an instance of sloppy coding practices in terms of security.

1. The permissions inside the contexts of this history-based mechanism are not sufficiently fine-grained.

   In the two games, the security mechanism stopped all network-based attacks, yet failed to detect any file system-based attacks. The problem is that the base context cannot identify all of the file system access requests during the testing stage. Therefore, the program has to be categorized as file system context to continue

Attack stopped$\surd$/missed$\times$		Malicious behavior				
Benchmark programs		1	2	3	4	5
Browsers	Jbrowser [24]	$\times$	$\times$	$\times$	$\times$	$\times$
	JXWB [26]	$\times$	$\times$	$\times$	$\times$	$\times$
Editors	Jedit [33]	$\times$	$\times$	$\times$	$\times$	$\times$
	Jext [25]	$\times$	$\times$	$\times$	$\times$	$\times$
IMs	BIM [23]	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
	SimpleAIM [27]	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Games	AntiChess [22]	$\surd$	$\times$	$\times$	$\times$	$\surd$
	Tetris [36]	$\surd$	$\times$	$\times$	$\times$	$\surd$

Table 3: Malicious behaviors inside the benchmark suite stopped or missed by the history-based access control. A $\surd$ indicates the failure of this instance of attack (being stopped); A $\times$ indicates the success of this attack (being missed).

running. Once the file system context is added into the historical profile of the program, any file system-based attack can succeed in this program.

One possible remedy would be to add a fine-grained file system permission into the file system context. Another choice would be to profile the program more extensively so that every possible file system access permission required by the clean version of the program could be added into the base context. However, this second approach has two shortcomings: Complete coverage during profiling is not always realistic; and we may not be able to to profile every program before deployment.

The two browsers are wide open to any attack. The network-related and file system-related permissions included in the base context are sufficient for all the attacks to succeed.

Although we characterized editors as file system oriented, the Jext program needs network access to provide the functionality of viewing a URL and editing the file denoted by the URL. The execution of this functionality during the profiling stage has already granted some network access permissions to the base context. As such, all network-based attacks in our benchmark suite can also succeed.

2. The information provided by Java is insufficient. It appears that the history-based access control mechanism did a perfect job in protecting the two IM clients. However the interpretation of the logging messages indicates these two mixed attacks (i.e., direct information leakage and indirect resource abuse) were stopped only because of the portion that needs file system access. The portions of these two attacks that have access to the network were not stopped by the mechanism.

   This time we do not believe the problem lies in the coarseness of the network access permissions. After all, it is impossible to specify every possible instance of a network connection. This suggests other information, such as the producer of the destination address of a network connection (binary or console input)) should be collected and analyzed to detect potential malicious behavior.

3. It may not be wise to count on other programs to fully appreciate and correctly utilize the security capabilities of a high-level system like Java. Java provides a good interface to mediate access to various resources: permission-based capabilities, as well as a security manager mechanism that intercepts each request to a resource to check granted capabilities. New security mechanisms such as this history-based access control mechanism can be readily implemented in this infrastructure. However, this mechanism can be rendered powerless if the application is not well-formed. For instance, a library function call inside Jedit simply requests `java.security.AllPermission` upon program startup. Once this permission is granted, our security mechanism (based on Java permissions and Java security manager) cannot offer any help. This is the real reason why our security mechanism cannot protect this program against any attack, even though the case looks exactly the same as in the cases of the two browsers and the Jext.

   This suggests that when we have little confidence in the code quality of an application, behavior-based security mechanisms may have to gather lower-level information to discern the behavior, even though a more convenient higher-level infrastructure is available.

We should note that these problems all apply to a wider range of security mechanisms. We expect to expose more design problems if similar benchmarking processes are applied to more sophisticated mechanisms.

5 Conclusion and Future Work

In this paper, we have presented a benchmarking methodology to evaluate the effectiveness of behavior-based security mechanisms. We have developed a benchmark suite and designed an evaluation framework. We exercised our suite by applying it to a simple history-based access control mechanism. We discussed the findings of our experiment. The experience and the results suggest that benchmarking is a viable approach to evaluate the effectiveness of behavior-based security mechanisms.

In the future, we plan to implement a set of benchmarks using other mainstream languages such as C and C++. This will allow us to evaluate some commercial behavior-based security mechanisms. In the long term, we plan to explore more sophisticated algorithms for malicious code placement. We also plan to look into whether we can use binary instrumentation to insert malicious code in binary form directly into an application.

References

[1] Information Systems Audit and Control Association. Control Objectives for Information and Related Technology (COBIT).

[2] BBC. Game Virus Bites Mobile Phones. `http://news.bbc.co.uk/1/hi/technology/3554514.stm`.

[3] Jeff Crume. *Inside Internet Security: What Hackers Don't Want You to Know*, chapter 4, pages 38–50. Addison-Wesley, 2000.

[4] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based Access Control for Mobile Code. In *Proceedings of the 5th Conference on Computer & Communications Security*, pages 103–118, 1998.

[5] Marshall D. Abrams et al. Position Papers. In *Proceedings of the 1st Workshop on Information-Security-System Rating and Ranking*, pages 35–40, 2001.

[6] Debin Gao, Michael K. Reiter, and Dawn Song. On Gray-Box Program Tracking for Anomaly Detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 103–118, 2004.

[7] Lawrence A. Gordon, Martin P. Loeb, and Tashfeen Sahail. A Framework for Using Insurance for Cyber-Risk Management. *Communications of the ACM*, 46(3), March 2003.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[9] Permissins in the Java^{TM} 2 SDK. http://java.sun.com/j2se/1.4.2/docs/guide/security/.

[10] Information Technology—Systems Security Engineering—Capability Maturity Model (SSE-CMM). ISO/IEC 21827.

[11] Jorma Kajava and Reijo Savola. Towards Better Information Security Management by Understanding Security Metrics and Measuring Processes. In *Proceedings of the European University Information Systems (EUNIS) Conference*, Manchester, U.K., 2005.

[12] Philip Koopman and Henrique Madeira. Papers. In *Proceedings of Workshop on Dependability Benchmarking*, 2002.

[13] David Moll. Testimony on Spyware in Congress. http://commerce.senate.gov/hearings/testimony.cfm?id=1496&wit_id=4255.

[14] U.K. Office of Government Commerce. IT Infrastructure Library (ITIL).

[15] Paul F. Roberts. Instant Messaging: A New Front in the Malware War. http://www.eweek.com/article2/0,1759,1818611,00.asp.

[16] Stefan Saroiu, Steven D. Gibble, and Henry M. Levey. Measurement and Analysis of Spyware in a University Environment. In *Proceedings of the 1st ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–31, San Francisco, CA, USA, 2004.

[17] Bruce Schneier. Attack Trends: 2004 and 2005. *ACM Queue, Special Issue on Security: A War Without End*, 3(5), June 2005.

[18] Kymie M. C. Tan, John McHugh, and Kevin S. Killourhy. Hiding Intrusions: From the Abnormal to the Normal and Beyond. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, pages 1–17, London, UK, 2003. Springer-Verlag.

[19] Apache Ant. http://ant.apache.org/.

[20] Common Criteria Evaluation & Validation Scheme (CCEVS). http://niap.nist.gov/cc-scheme. National Institute of Standards and Technology.

[21] National Vulnerability Database. http://nvd.nist.gov/.

[22] AntiChess. http://sourceforge.net/projects/antichess/.

[23] BIM. http://sourceforge.net/projects/bim-im/.

[24] Jbrowser. http://sourceforge.net/projects/jbrowser/.

[25] Jext. http://sourceforge.net/projects/jext/.

[26] JXWB. http://sourceforge.net/projects/jxwb/.

[27] SimpleAIM. http://sourceforge.net/projects/simpleaim/.

[28] WebSense. http://ww2.websense.com/.

[29] Cisco Security Agent 4.5. http://www.cisco.com/.

[30] NOD32. http://www.eset.com/.

[31] ICSA Labs. http://www.icsalabs.com/.

[32] Information Technology Security Evaluation Criteria (ITSEC). http://www.itsec.gov.uk/. Commission for the European Communities.

[33] Jedit. http://www.jedit.org/.

[34] McAfee Entercept 5.1. http://www.networkassociates.com/.

[35] PC Tools. http://www.pctools.com/.

[36] Tetris. http://www.percederberg.net/home/java/tetris/tetris.html.

[37] QRadar. http://www.q1labs.com/.

[38] Sana Security Primary Response 3.0. http://www.sanasecurity.com/.

[39] Snort. http://www.snort.com/.

[40] Virus Bulletin. http://www.virusbtn.com/.

[41] Checkmark. http://www.westcoastlabs.org/.

[42] WAVES (Web Application Vulnerability and Error Scanner). http://www.openwaves.net/.

[43] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, New York, NY, USA, 2002. ACM Press.