A Benchmark Suite for Behavior-Based Security Mechanisms

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 products) by tackling zero-day attacks that have no signatures extracted yet, as well as polymorphous attacks that have no stable signatures.

In this work we present a benchmark suite of eight programs. All of them are legitimate applications and can be infected with a variety of malevolent activities. An evaluation framework is designed to infect, disinfect, build and run the benchmark programs. This benchmark suite aims to help evaluate the effectiveness of behavior-based defense mechanisms during multiple 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 analysis to protect the execution of the application running on a sever from being hijacked. Most of these applications are known or highly suspected to horde security vulnerabilities like buffer overflow and format string [21]. These systems use various methods to examine the actions taken by a program at the abstraction level of library API or system call. Actions that appear malicious, such as attempting a buffer overflow or opening a network connection in certain contexts, will trigger the alert of monitoring agents.

Over the past few years, spyware has become a pervasive problem [13, 16]. Many infections occur when spyware is piggybacked on 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 to address this problem. All of these companies have emphasized that they detect spyware by observing the system behavior and discerning the abnormal activities from the normal ones.

Signature-based intrusion detection and anti-virus solutions lag behind these types of exploitation and don not adapt well to even small changes of the exploitation. 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 no signature extracted yet and polymorphous attacks pose a great hardship 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 behaviors associated with malicious activities and distinct from the normal behavior of legitimate programs. Evading a behavior-based protection 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 pre-

vention systems. Different amount of details of various system components along a process' system call trace can be recorded and characterized as the typical behavior of the program. By establishing a profile of the normal behavior of a program, 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. The idea is 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 will be 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. Whether a resource request the program makes will be granted or rejected 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 common and the rules and analytics engine underlying these mechanisms become more sophisticated, we need a way to evaluate these security mechanisms. The evaluation could and ideally should be used both for testing purpose during development of these mechanisms and for validation and ranking purpose among competing offerings.

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 the level of security a whole systems achieved [20, 32].

Kajava et al [11] summarized a variety of criteria to qualify and quantify the sense of security of a target system into three major types: risk analysis, certification and measures of penetration process.

- Risk analysis is a process of estimating the possibility of individual exploitations, their consequence to the system and the organization, as well as the costs to mitigate the risk. Risk analysis is thought to be the basis of high-level security evaluation and a trade-off between cost and protection [3].

- Certification is the classification of 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 could be put on the security features offered by a product [41, 31], or the degree of conformance of a security process to the established guidelines such as ITIL [14], CMM [10] and COBIT [1].

- Penetration testing is statistics measurement of the outcomes of those efforts to make an intrusion against a target. For example, the WAVES project [42] standardizes the practice of penetration testing Web applications.

There have also been some efforts trying to employ multiple orthogonal criteria to quantify the value of the perceived security enhancement and the cost paid for these 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, as well as how to evaluate and justify the investment on the security measures mitigating the risk exposure. The criteria they used for security evaluation has covered all the above three types.

Yet there is no widely accepted way to measure and rank security properties. The difficulty of finding a common ground for evaluating various security mechanisms suggests that it is worthwhile to explore different evaluation methodology for different categories of security mechanisms.

This paper is to apply the benchmarking methodology to evaluate those behavior-based security mechanisms. We present a benchmark suite composed of eight applications that are typical on a workstation/desktop environment. These applications are infected with a variety of malicious code that in turn represent a wide spectrum of exploitations these

mechanisms try to protect against. We use the benchmark suite to test a simple example of 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 a 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. Conclusion and future work are given in section 5.

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 put together as this common ground. These applications reflect typical workloads running on a certain category of computer systems or being processed by a certain class of system software. The value of different designs and implementations is reflected in the performance obtained by the system when running the benchmark suite. All other conditions remaining the same, the one that runs the benchmark suite fastest is the one that performs best. Benchmarking promotes the practice of quantitative analysis [8]. There have also been some efforts to apply benchmarking methodology to evaluate properties other than performance, such as dependability [12].

One of the main characteristics of most security-related mechanisms is that they address a moving target. The activities and scenarios that may do harm to the system are unpredictable. It would seem that 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 difference between their various approaches, all the behavior-based mechanisms make a common claim that they 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 motivated by financial gains [17]. This indicates that the malicious behavior these mechanism are trying to single out is limited and 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 behavior-based security mechanisms an objective test, either for prototyping a new idea, or for giving an independent evaluation of a final product.

Our benchmarking methodology is different from either the penetration testing performed by third-party audit and certification service providers [41, 31] or a software package composed of a set of penetration cases [42], although it seems that all of them carry out attacks against legitimate applications.

- First, the main purpose of penetration testing is to find out 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.

- Second, penetration testing is very implementation specific. Whenever a exploitation against a new 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 exploitations. Unless the entire logic 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, the benchmarking methodology is complementary to the audit and certification services. Designers and developers can benefit from our benchmark suite because it is more cost-effective and convenient 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 (i.e., cutting-edge) exploits and see how many of them anti-virus products can detect. In a test performed by `Virus Bulletin`[40], all the tested anti-virus software detect 100% of the signatures in their benchmark suite. It is apparent that there is no much meaningful comparison here. The problem indicates that benchmarking may not be a good way to evaluate the detection accuracy (i.e., effectiveness) of anti-virus technology.

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 SecSpec-Behavior Benchmark Suite

3.1 Components of the benchmark suite

We have developed a benchmark suite called SecSpec-Behavior. The benchmark programs included in the suite and the malicious code are all 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 and another set of benchmark programs may need to selected.

We target a typical workstation/desktop computing environment when choosing component programs for the benchmark suit. We include four types of applications and pick two particular exampales in each type.

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

Editors: Jedit [33] and Jexit [25] are two full-blown editors. The rich features of these two applications pose a great challenge to behavior-based security mechanisms to keep false positives low: identify any malicious activity but not interfere with the "suspicious" but legitimate activity.

Instant Messengers: BIM [23] and SimpleAIM [27] are two simple AOL instant messenging clients. Among them SimpleAIM is console-based and BIM is GUI-based. IM has become a serious application in both enterprise and personal desktop environment 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 the games 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 make this categorization based on the behavior they present. Each category of malicious behavior includes one or more implementations. Table 1 lists these malicious behaviors we categorized and their implementations.

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, respectively.

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 malicious code in the context where the original benchmark presents a similar rather than distinctive profile of library API

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

and system call activities, a behavior-based mechanism will face a bigger challenge to do its job well. The authors of [18, 43, 6] 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 context of the benchmark program around the placeholders. However we have focused on the level of application behavior instead of the library API and system call because our purpose is not to defeat these security mechanisms but to evaluate their effectiveness. We want to measure the robustness of the analytics logics and rules sets underlying these mechanisms in face of potentially confusing information. We call this practice orthogonality-directed placement. The less orthogonal the malicious behavior and the surrounding context of benchmark are against each other, the greater difficulty 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 experimented with two placement schemes:

Random placement: Beyond the minimum requirement of not breaking benchmark programs, our random placement makes sure the malicious code will appear in at least two types of locations: (1) where it will surely appear on the execution path; (2) where it may or may not appear on the execution path, depending on some particular run time events. We place the placeholders in the startup or termination section to emulate the first scenario and in the UI event handling section to emulate the second scenario.

Orthogonality-directed placement: This requires us to figure out the degree of similarity of the program behavior and the malicious behavior. Our simple approach is to classify both all the benchmark programs and malicious code to four general categories of behavior: network oriented, file system oriented, mixed or neither. We then mix them together according to the extent of overlapping of their behavior among 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 behavior, we classify indirect information leakage, information tampering, and direct resource abuse as file system oriented, direct information leakage and indirect resource abuse as mixed.

A 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 the placeholders are inserted manually.

3.3 User interface of the benchmark suite

The user interface to the benchmark suite is via the Apache Ant build tool [19]. We provide four build

Benchmark programs		Malicious behavior				
		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

targets for each benchmark program:

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

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, displaying the command line for running the benchmark program to standard output.

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 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 permission [9] which could permit the 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:

1. The program's historical profile has already included a context that contains this permission,

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

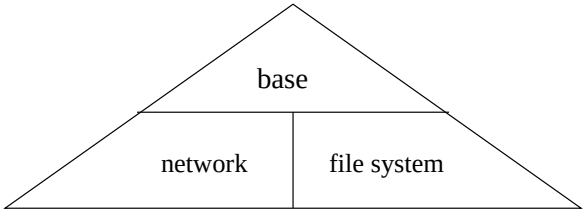

Figure 1: Contexts provided in history-based access control mechanism

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 mechanism succumbs to some combinations of a malicious code and an individual benchmark program.

The mechanism we implemented has three contexts: base, network and file system as shown in Figure 1. Base context grants the most restrictive permission, 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 meet the needs, two things will happen: base context will be added to this program's historical profile; and the search process stops, even permission in either the network or the file system context may also allow this access.

```
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 history-based access control mechanism

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

4.2 Evaluation

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

During the profiling stage, a clean version of each benchmark is run through once. We have modified the security manager to intercept all the 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 fine-tune the gathered permissions into the base context to the very fine-grained level in order to minimize the risk exposure. We make sure the clean version of each benchmark can run without having to be categorized into either the network or the file system context.

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

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

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

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

Before this experiment, we think that some information that are missing in the Java permission may cause trouble to our security mechanism, since it may not be able to distinguish the malicious behavior from the normal behavior with available information. Also, we suspect the permission gathered in the profiling stage is not fine-grained enough, i.e., it may be too permissive. The analysis of the result confirmed our thought and suspicion. In addition, it uncovered an instance of sloppy coding practice 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 accommodate all the file system access requests during the testing stage. Therefore, the program has to be categorized as file system context to continue 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 adding more fine-grained file system permission into the file system context. Another choice is 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: (1) Complete coverage during profiling is not always realistic; (2) 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 enough for all the attacks to happen.

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

2. The information provided by Java permission is not enough.

   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 portion of these two attacks that have access to the network was not stopped by the mechanism.

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

3. It may not be wise to count on others' programs

to fully appreciate and correctly utilize the security capabilities of a high-level system like Java.

Java provides a good interface to mediate the access to various resources: permission-based capabilities, as well as a security monitor mechanism that intercepts each request to a resource to check granted capabilitities. New security mechanism like the history-based access control mechanism can be readily implemented in this infrastructure. However, this mechanism is rendered powerless by those not well-formed applications.

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 permission and Java security monitor cannot offer any help. This is the real reason why our security mechanism cannot protect this program against any attack, even the phenomenon looks exactly the same as in the cases of the two browsers and the Jext.

This suggests that when we have no 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.

It is noted 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 the 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 over 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 near 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 format into an application directly.

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™ 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.