Anomaly Intrusion Detection in Dynamic Execution Environments

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
We describe an anomaly intrusion-detection system for platforms that incorporate dynamic compilation and profiling. We call this approach "dynamic sandboxing." By gathering information about applications' behavior usually unavailable to other anomaly intrusion-detection systems, dynamic sandboxing is able to detect anomalies at the application layer. We show our implementation in a Java Virtual Machine is both effective and efficient at stopping a backdoor and a virus, and has a low false positive rate.

Keywords
anomaly detection, Java, dynamic sandboxing

1. INTRODUCTION
Over the past several years, there has been a move towards dynamic compilation, profiling, and optimization technologies. We call these platforms Dynamic Execution Environments (DEE). Some well-known examples include Sun's Java platform [22], Transmeta's Crusoe [15], HP's Dynamo [32], and even the Common Language Infrastructure (CLI) component of Microsoft's .NET project [10]. These technologies will further decouple hardware and software, allowing legacy code to benefit from advances in both hardware and software design.

The popularity of Java and Microsoft's support of C# suggest that these technologies will soon become ubiquitous. Although they were adopted for performance reasons, the potential exists to leverage their infrastructure for anomaly intrusion detection with extremely low performance penalties. Because each virtual machine (VM) incorporating these technologies hosts only one application, the resulting intrusion-detection system (IDS) is customized to a specific application, an additional benefit.

Application intrusion-detection systems (AppIDS) are not a completely new idea. Other researchers have pointed out that IDSs customized to specific applications might use semantic information not available at lower levels of execution to improve intrusion detection[27, 36, 3]. Several such systems have been constructed, many with the idea of protecting common services, like HTTP [26]. However, incorporating application-level information into an IDS is a nontrivial exercise, and in the earlier projects, the application was modified and the IDS constructed by hand [3, 36]. Our approach automates the construction of an AppIDS without modifying the application. We do this by profiling information already in place for dynamic optimization. We call this "dynamic sandboxing."

To demonstrate this approach, we describe an implementation of Dynamic Sandboxing in a Java Virtual Machine (JVM). We chose Java because it is currently the most popular DEE. In the rest of this paper, we summarize the Java security model and its vulnerabilities; we then present a detailed description of dynamic sandboxing and give evidence that normally executing Java programs exhibit highly regular behavior. Next, we describe our implementation of dynamic sandboxing and report experimental results on its efficiency and its effectiveness at stopping attacks. Finally, we discuss the implications of our results and outline some areas for future research.

2. JAVA SECURITY
To see where Dynamic Sandboxing fits into Java, a review of the current state of Java security may be helpful. Java is noted both for the mobility of its code and its integrated security mechanisms. Recent versions of Java have an extremely rich security model, allowing precise control of multiple sandboxes in a single JVM [24, 34]. This contrasts greatly with the original "all-or-nothing" sandbox, in which users chose between a overly restrictive applet sandbox and the application sandbox that had no restrictions. This original security model was then extended to give signed code additional permissions permissions. Even more choices have been added to recent releases, allowing users to specify custom sandboxes and developers to add new types of permissions to the security model.

Java's security model rests upon its bytecode instruction set—a form of typed assembly language. Type checking allows the verifier to prevent unsafe behavior, such as bad casting or pointer manipulation, from occurring. Runtime checks for dynamic features, such as array bounds checking, complement static verification. The Security Manager sits on top of this infrastructure. All potentially dangerous actions, such as I/O calls to the network or filesystem, are guarded by a call to the Security Manager, which interprets the policy. A policy is a list permissions for each sandbox in the JVM. Interaction between different sandboxes is handled internally through stack introspection [34]. Java security thus resembles the matrix security model described in many texts, in which verification guarantees fidelity [25, 28]. One complication is the possibility of multiple interacting sandboxes, each called a "protection domain." A complete description of Java security mechanisms and the relevant APIs is available in Java Security [24].

Current research on Java security emphasizes verification mechanisms, either extending them or leveraging them in new ways. For example, code-signing, information-flow, and proof-carrying code all rely on verifying features of the bytecode. Code-signing has been available in the JDK for several releases. Information flow mechanisms are being investigated by Andrew Myer's group [23]. Proof-carrying code has long been a research topic in the OS community with language support an important component. Colby et al's work on Java is an example [9]. In all of these approaches, verification ties code permissions to identity. The JVM allows executing code to inherit permissions from the entity who signed the code. In short, the signing entity allows the JVM to "trust" the code. For proof-carrying code, verification ties proofs to properties of the code itself. Instead of trusting the code, the JVM deduces properties of the code with the help of the proof, namely that the code does not exceed the permissions allotted to it.

The richness of Java's verification and other security mechanisms means that policy can be very complex. This raises questions about how users will be able to maintain complex policies correctly. And, even if the policy is correct, developers must add Security Manager hooks at the appropriate places in their application and the JVM's verifier must be bug free in order for the application to be secure. Thus, there are many opportunities for security violations in the current Java security model, and there is some evidence that the standard mechanisms, verification with runtime security checks through the Security Manager, do not guarantee secure environments. As an example of this latter point, we describe briefly several applet exploits that have been reported since April, 1997. Many of these exploits are against the verification mechanism:

- **September 2002:** The verification mechanism used to supervise DLL loading in Java Database Connectivity (JDBC) classes of the Microsoft VM can be spoofed to allow the loading of any DLL. An unrelated bug in the same package results in allowing arbitrary data to be used as a pointer.
- **October 1999:** A bug in the verifier for Microsoft's VM allows casting of values to unrelated types. This can be used to violate the sandbox and gain complete access to the client's computer.
- **April 1999:** A bug in all popular JVMs allows downloaded code to execute without prior verification. Malicious bytecode can subvert the type safety mechanisms to gain access to the client.
- **July 1998:** A Classloader bug allows an applet to redefine vital classes and lead to uncaught violations of the type system, potentially allowing breaches of the sandbox in Netscape 4.0.x.
- **April 1997:** An Applet can change its own signee to one that is trusted, allowing subversion of the sandbox in JDK 1.0

Others exploit the Security Manager:

- **September 2002:** Microsoft's XML utility classes do not have the proper Security Manager hooks, allowing a remote attack to gain control.
- **October 2000:** URLConnection and URLInputStream do not implement Security Manager hooks. Applets can then access files using a string argument of the form `file://filename` to gain access to any file on the client, including directory listings on Netscape 4.5.x. The demonstration exploit implemented an HTTP server named Brown Orifice which gives an attacker unlimited access to a victim's files.
- **August 1999:** A race condition in the Security Manager library of Microsoft's VM allows violation of the security rules. Applets can gain total access to the machine using this bug.
- **July 1998:** A bug in the Security Manager allows applets to turn off the verifier in Netscape 4.0.x.

All of the above exploits, except for Brown Orifice and the September 2002 Microsoft vulnerabilities, were reported by the Princeton Secure Internet Programming team [4, 21]. Brown Orifice was discovered by Dan Brumleve [6]. The Microsoft vulnerabilities were disclosed by the company itself [2]. CERT has issued three advisories pertaining to Java exploits. Other individuals post exploits to their web pages [1]. Some of these show denial-of-service exploits, which do not breach the sandbox, and are annoying (removing them may require shutting down the browser), rather than dangerous. Although these numbers are nothing compared with intrusions at the network or operating system levels, we believe they will increase as Java becomes more prevalent in production systems and there are important applications worth compromising.

Currently, almost all exploits are applet exploits. Application exploits have not yet become a priority for attackers, possibly because there are few commonly used Java applications. However there is one published application exploit, a virus named Strange Brew [14]. We are not aware of any existing attacks that exploit policy bugs, but this type of security flaw is almost inevitable given the potential complexity of policies and the complicated nature of "protection domain" interaction. These problems, both real and projected, support our view that redundant mechanisms, in this case an IDS, will improve security in the Java domain.

3. DYNAMIC SANDBOXING

Dynamic sandboxing for a given program consists of two activities: sandbox generation and sandbox execution. In the first, a sandbox profile is constructed by running the program with an instrumented JVM. During this training session, profiling information is recorded to the sandbox profile. The sandbox is initially empty and grows during the training run by accumulating records for each unique behavior. Because nothing is added that isn't observed, each sandbox is customized to a given program and context in which it is executed. During sandbox execution, behavior that isn't in the profile is considered anomalous.

Dynamic sandboxing is meant to complement, not replace, the standard sandbox. Java's standard security model creates fixed boundaries within which a program must execute while dynamic sandboxing detects anomalies. Although we haven't addressed the question of response in our current implementation, a dynamic sandbox response is potentially very flexible. It could range from logging the anomaly to the application termination. In the standard Java security model, there is only one response, which is to disallow the attempted behavior.

Dynamic sandboxing's efficacy is directly related to the stability of the instrumented program's behavior. If the underlying program continually executes large amounts of novel code, it will generate a high number of false positives. As we discuss in the next section, however, our results suggest that many programs have highly regular behavior.

3.1 Program Behavior

An example of regular program behavior is the familiar 90/10 rule, the rule of thumb that 90% of a program's time is spent in 10% of its code. Computer architectures have long exploited program behavior regularities through caches and branch prediction. Most compilers can use profiling information to guide optimized code generation, and we have found that many programs executing with JVMs exhibit regular behavior.

Because JVMs profile many aspects of an executing program, there are many possibilities for anomaly detection. Some of these possibilities are listed below, in order of increasing complexity. For each feature, the utility, space requirements, and performance impact are discussed.

- **Methods:** Methods are the fundamental units in Java bytecode. JVMs are built around manipulating this structure and allow easy instrumentation of their dynamic behavior. Methods are our initial best guess of a viable unit out of which to construct profiles.
- **Memory behavior:** Some applications have distinctive heap allocation and garbage collection behavior. Garbage collection is becoming more important and integral to performance sensitive applications, especially in Java. Adaptive-garbage collection methods are being actively investigated, and most garbage collectors have hooks for instrumentation as well. Instrumenting memory management might reveal anomalies not seen with other techniques.
- **Basic blocks:** There are many more basic blocks than methods. Thus, basic blocks would increase space of possible anomalies enormously. In addition, basic blocks can be extensively instrumented because they are the base of many optimizations.
- **Method arguments:** Instrumenting method arguments would allow finer distinctions between normal and abnormal execution sequences, and might prove useful in the future. It would, however, result in a large method-argument space, and we have thus avoided it for our initial implementation.
- **Patterns of methods:** Rather than looking at individual methods, it might be useful to characterize the patterns in which they occur, using any of several data modeling methods. System-call sequences, for example, have been used successfully to characterize specific processes [35]. Method sequences might show similar behavior. As the space of patterns defined over a data set is always larger than the data itself, we decided to start with the simplest possible representation (presence or absence of individual methods). This corresponds to a sequence length of one.
- **Whole program paths:** Larus et al. showed that programs spend 90% of their time in a small number of "hot paths" [5, 17]. Paths reveal a large amount of information about the global behavior of an application, are beginning to be used for performance reasons in VMs, and thus, they are a candidate for future use [32].

For our initial implementation, we have focused on methods, which have very regular behavior. We are not looking at sequences, as in earlier work on system calls [11, 13, 35]. Consequently, we do not need to consider threading interactions, or other trace path complications. Several tens of thousands methods could potentially be executed by the typical application (compared with at most 200 potential system calls in a typical Unix system), so starting with the smallest possible sequence length of one was reasonable. Indeed, we find that programs exhibit similar behavior at the single method invocation level. We report results for two examples: the canonical "Hello World", and a much larger and more complicated peer-to-peer file sharing utility, LimeWire [19]. If one looks at the number of methods invoked by the two programs during runs, then plots the methods sorted by frequency using log-log scales, one finds power-law behavior. Figure 1 and Figure 2 plot this behavior. In Hello World, one finds power-law behavior with small deviations. On the larger LimeWire program, the simple power-law behavior soon decays with time into more complicated, multiple domain power-law behavior. Even with this complexity, however, the plot shows what code optimizers have long known, that a very few methods account for a very large proportion of behavior.

3.2 Implementation

Our current prototype monitors only method invocations. During the training session, each time a new method is invoked, its signature is written to a log. This log constitutes the sandbox profile. The profile for the "Hello World" program, for example, contains 268 methods. Every method,

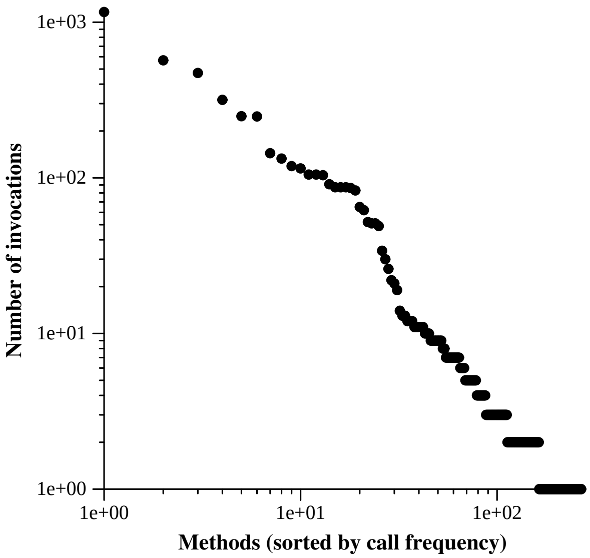

Figure 1: Behavior of Hello World

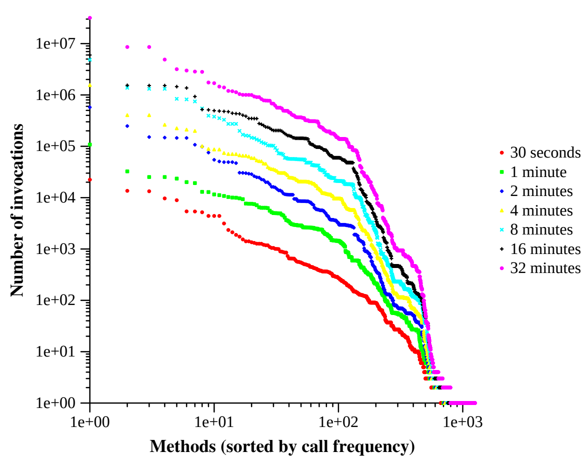

Figure 2: Behavior of "LimeWire" Gnutella Application

including those in libraries and natives, was included in the profile. During execution, the log is checked for the method signature before the method is compiled.

We modified Intel's Open Runtime Platform (ORP) to perform dynamic sandboxing [16]. From a user's perspective, we made two changes to the ORP interface. First, we added two flags `-profile <filename>` and `-sandbox<filename>`, which write a profile to a file and read in the profile to be used as a sandbox, respectively. The two flags may be used simultaneously.

The mechanisms of generating and using the sandbox profile rely on ORP's Just-in-time (JIT) compiler implementation. When ORP loads a class it doesn't compile the Java byte code to native code. Rather, it follows a lazy strategy, delaying native compilation of each method until it is invoked. In the place of the native code, ORP inserts small stubs called trampolines, which operate as follows:

- Call the JIT compiler for the specified method.
- Patch the jump table to call the newly compiled native code.
- Jump to the newly compiled code.

Dynamic sandboxing requires a slight modification of this strategy. We modified ORP to append the class and signature to the end of the sandbox profile. This takes place in the profiling phase and before the JIT compiler is called. When in sandboxing mode, our modified ORP checks the method against the sandbox profile. Only if the profile contains the method does ORP continue with JIT compilation. If the method does not appear in the profile, ORP informs the user of a security violation and exits.

The lookup of methods in the sandbox profile is currently implemented as a linear search through the file. The sandbox profile file format is simply a list of class and method signature pairs prefixed by their combined string length.

A summary of the algorithm is presented in pseudocode:

```
int result;
if ( jit_status == ON )
{
    compile_method(method);
}
else
{
    profile_method_load(sandbox, method);
}

if ( profiling == ON )
    add_method_to_profile(profile, method);
```

This code is executed only if the method has not been previously invoked. If the JIT is enabled, then we are in the training stage. If the JIT is not enabled, then we call profile_method_load with the current sandbox. After that, if we are in the profiling mode, we add the the JIT to the profile. Note that there are two profiles active here, the profile we are gathering, called *profile*, and the sandbox profile, called *sandbox*. Currently, profile_method_load checks to see if the method signature is in the profile and then loads and compiles the method.[1]

Dynamic sandboxing in ORP was designed to be efficient. First, we are able to use a JIT compiler, so programs execute efficiently. Second, instead of performing a check on every method invocation, we only perform the check the first time a method is invoked. There is one remaining implementation inefficiency—the linear search of the sandbox profile. Although its worst case is linear in the number of unique methods invoked (corresponding to a linear search through the file), in practice it is much closer to constant because the order of methods in the profile reflects the execution path from a previous run. The execution paths of later runs are usually similar, so that the profile file pointer is usually pointing to the method description queried by ORP.

Dynamic sandboxing in ORP is only one of several implementations we have developed. An earlier implementation relied on Kaffe's interpreter [31], which ran very slowly. Recently we have also developed implementations by instrumenting libraries and adding a wrapper around the Sun JIT compiler interface. This allow us to run with the standard Sun libraries. However, there is a large performance cost to this last approach, because there is overhead on every method invocation instead of only on the first.

3.3 Experimental Results

In order to be useful, the implementation should have the following properties: It should be effective at stopping attacks, be efficient, and experience few false positives. As we discussed above, there is a paucity of documented attacks in the wild on production Java programs, which makes it difficult to perform large-scale systematic testing. We did, however, run our sandboxing method against two exploits, one taken from the literature and one that we devised ourselves. To assess the performance impact, we used two benchmarks.

3.3.1 Effectiveness

We tested dynamic sandboxing against a Java virus and a simple HTTP server with a backdoor. The Java virus, named Strange Brew, is the first reported virus targeting Java programs [14]. When invoked, the virus searches the current directory for uninfected class files. For each uninfected class, it adds a copy of itself to the class and modifies the constructor to call itself. It then pads the file to a multiple of 101 so it can determine a class's infection status without opening it.

The Strange Brew virus readily infects any application, including the canonical simple program "Hello World." In Java, "Hello World" consists of a one-line main routine which calls System.out.println(). When the infected program is run with a profile gathered against the uninfected class, no security violations are found. That is because main() never calls the infected constructor—no "Hello World" instance is created. It isn't until the anomalous code attempts to execute that our sandbox can detect a problem. Depending on one's point of view, this is either a "feature" or a "bug." In our view, the foreign code isn't dangerous until it executes, and in this way intrusion detection can devote its resources to code that is about to cause damage. Thus, dynamic sandboxing focuses on *behavior*, not structure. With this in mind, a second line was added to Hello World which calls the constructor. The new class proved infectious. With sandboxing enabled, however, the first call into the virus violated the profile, causing ORP to exit, and the attack was prevented. This result would be seen for any program. not just "Hello World". Because it is the foreign virus code that is identified, the dynamic sandbox is able to prevent the virus from infecting any dynamically sandboxed program.

The second exploit we tested is a backdoor to a simple HTTP server written by the first author. The backdoor was implemented as a special command which allows a remote client to execute arbitrary commands on the server. A sandbox profile was constructed by running the HTTP server and exercising it by downloading pages. We then activated dynamic sandboxing using the custom profile. When we attempted to exercise the backdoor, the sandbox trapped the first call in the backdoor, a call to System.exec, without disrupting other legal uses of the server. Again, dynamic sandboxing is effective here because our sandbox is a reflection of behavior, not structure. The exploit does not insert foreign code into the application, although we believe dynamic sandboxing would protect against that as well (based on the virus example described earlier). In this case, the malicious code is part of the application itself. Because the backdoor is a form of "dead code," the sandbox effectively eliminates it.

3.3.2 Efficiency

Dynamic sandboxing should be efficient as well as effective, if it is ever going to be useful. We predict that our implementation will perform efficiently for applications in which methods are executed repeatedly. For many interesting programs, notably server applications, this is true. Indeed, this is almost universally true, because the number of total invocations is much larger than the number of methods available in most applications.

To confirm that our implementation is efficient on realistic programs, we tested dynamic sandboxing against a Java version of the Olden benchmarks [8, 7]. The Olden benchmarks are a series of 10 programs that are representative of typical applications. We ran the benchmarks 15 times each with no flags, the -profile flag, and the-sandbox flag, for a total of 45 runs.[2]

The results appear in Table 1. Profiling and sandboxing each incurred an overhead of less than 2%, confirming our

[1] Ideally, profile_method_load would load the native code directly from the profile. To accomplish this, however, we would have had to write the equivalent of a linker. We are currently investigating how this might be accomplished.

[2] We used the standard benchmark parameters with three exceptions: 2048 instead of 4096 on Barnes-Hut, 512 instead of 1024 on Minimum Spanning Tree, and 10 instead of 20 on the TreeAdd (tree traversal) benchmarks. The original parameters exceeded the unconfigurable ORP heap size.

ORP Parameters	Mean user + system time in seconds (Std. Dev.)
Default (no sandboxing)	304.59 (0.26)
Logging the sandbox profile (-profile)	308.93 (0.35)
Dynamic sandboxing enabled (-sandbox)	308.03 (0.19)

Table 1: Efficiency of sandbox generation and protection on the Olden benchmark.

expectation that the implementation would be efficient.

The Olden benchmarks show that the average case is efficient. How efficient is the worst case? We wrote a synthetic benchmark to test the efficiency of our implementation under conditions in which the overhead of lookup cannot be amortized over multiple invocations. Because overhead is isolated to the first invocation of each method, our benchmark consists of a class with 1000 empty static methods, each invoked once. After gathering the sandbox profile, we modified the benchmark to invoke the methods in the exact reverse order of the profile – the pathological case. The benchmark was run 100 times under no flags, `-profile`, and twice under `-sandbox`. See Table 2 for the resulting data.

Sandbox generation and the synthetic benchmark had modest performance decreases of 6%. The pathological case had an enormous slowdown of 2216%. The slowdowns shown for profiling and dynamic sandboxing exaggerate the true effect they have on applications and are included to show worst case behavior. First, overhead is only incurred on the first invocation of a method. True overhead is small when amortized even over a modest number of invocations, as seen in the Olden benchmarks. Second, the numbers don't reflect the cost of invoking a JIT for non-empty methods. In real applications, any overhead in profiling or sandboxing is quickly overwhelmed by the cost of JIT compilation. Finally, the pathological case shows that the lookup scheme isn't efficient in all cases. It is optimized for method invocations to occur in the same order encountered during profile generation—the pathological case is ordered exactly opposite. This is unlikely in most programs, so moving to a scheme like hashing would make the system slower in the normal case. The typical case would be much more expensive, although including hashing in addition to the current scheme might improve performance at the expense of space.

3.4 False Positives

The exploits and efficiency tests give evidence that dynamic sandboxing can be effective and efficient at stopping exploits. The last requirement for an IDS is a low false positive rate. For the experiments described above, we found zero false positives. This is certainly encouraging, but not conclusive given the limited nature of our tests. For real applications, we would expect to see at least some false positives.

The false-positive rate is related to the problem of "perpetual novelty" in interesting applications. A anomaly-detection system can never be sure that it has observed the entire range of normal behavior. One approach to this is that of generalizing over the space of observed patterns, with the hope that the generalization will include most new legitimate behavior as in [18, 12, 20].

A second approach is to look at the distribution of novel patterns in the data and use that to make predictions about the distribution of novel patterns in the data. A rank/frequency plot is often used to study such distributions, such as the figures shown in Figures 1 and 2 in Section 3.1. When plotted on a log-log scale, the slope of the rank/frequency curve reveals the proportion of frequently seen behavior to potential false positives.

For example, consider the LimeWire trace recorded for 32 minutes discussed in Section 3.1. We can see that the distribution falls off more quickly than a power-law. This tells us that the frequency of rare events decreases faster than a polynomial and slower that an exponential. Only 162 methods are required to obtain .01 false positive rate per method invocation and 444, 35% of total number of methods executed during the trace, to push that rate to .0001.[3] As the program is run longer, the relative frequency of these rare events will decrease (this can be seen by examining the different trajectories plotted in Figure 2.

Finally, the size of the entire profile is small enough that we can record the long tail of the power-law in our profile. We emphasize again that we are not looking at paths or sequences, which blow up the space enormously. LimeWire, a "real" application, invokes less than 1300 methods. In fact, the entire possible space is only on the order of tens of thousands of methods, depending on the specific Java distribution used. As Somayaji has noted, smaller spaces have advantages in the areas of space requirements, generalization, and profile generation time [29].

4. DISCUSSION

Dynamic sandboxing, like other anomaly IDS, infers policy from behavior. This is predicated on the idea that normal execution can reveal and document complex policies more reliably and efficiently than users or developers can. As we know from familiar environments like UNIX, user devised policies are often flawed. As with any empirically derived model of normal behavior, dynamic sandboxing comes with the risk of imperfect detection, that is, false positives and false negatives. In practice, we expect the number of false positives to be low, but to date we have only limited experimental evidence to justify this prediction.

This differs from specification based approaches, like that of Wagner and Dean [33], with no possibility of false positives. Although they looked at system call sequences, an

[3]We would have liked to have used LimeWire as a test, but it currently does not run under ORP. We instrumented the LimeWire bytecode in order to get the data, and as a result the program was too sluggish for human interaction. The number of invocations would be much larger in uninstrumented code for the same time period.

ORP parameters	Mean user + system time in seconds (Std. Dev.)
Default (no sandboxing)	0.26 (0.01)
Logging the sandbox profile (-profile)	0.28 (0.01)
Dynamic sandboxing enabled (-sandbox)	0.28 (0.01)
Dynamic sandboxing on pathological benchmark	5.83 (.21)

Table 2: Efficiency of sandbox generation and protection on the synthetic benchmark.

analogous system could be devised for Java. Although they prefer static linking, a realistic Java IDS would build the specification from the bytecode class files at load time. The IDS would then limit execution to a larger sandbox, the entire possible call-graph of Java methods determined from the bytecode. This provides little protection over the traditional sandbox, and is simply a more complex verification mechanism. Once an attack has circumvented the traditional security mechanisms, it can execute anything, since those code paths would have been examined statically at class load time. Considering our own experiments, both the virus and trojan HTTP server exploit would run successfully. This is because the code, which is the basis for the specification in the hypothesized Wagner and Dean type IDS, carries the exploit with it. Dynamic sandboxing prevents the security fault because it focuses on behavior, not structure.

Specification based approaches often have significant performance penalties. The Wagner and Dean system experienced slowdowns on the order of seconds per transaction [33]. We believe, however, that IDSs will not be used unless they impose minimal performance penalties. Also, we believe IDSs should prevent intrusions from gaining control. Our systems must therefore be computationally efficient and online. Dynamic sandboxing meets these goals. The system is essentially free, and can run as part of the just-in-time compiler system.

Given that we have studied only two attack classes, it is interesting to consider how dynamic sandboxing would fare against a wider range of attack types. Because dynamic sandboxing stops novel code from executing, the sandbox would prevent it from using any methods outside the traditional Java sandbox. A successful exploit would need both to disable the traditional security mechanisms and use only previously invoked methods (perhaps with different arguments). Against all three classes of exploits (verification bugs, security manager bugs, and policy bugs) dynamic sandboxing would likely perform well against naïve attackers. In each case, the initial intrusion might succeed, but new code could not execute if used any methods not already in the profile. One possibility is that the exploit could jump to native code, which dynamic sandboxing could not stop, because it acts only within the Java domain. An intelligent adversary would need to embed the payload within methods already in the profile, in what is essentially a mimicry attack.

The existence of false positives makes response tricky. Because we have not yet experienced false positives in this setting, our current response is simplistic, exiting the JVM when an anomaly occurs. If false positives are rare enough, and we believe they will be, additional analysis could be performed when an anomaly is raised. The JVM could completely deconstruct the stack, giving the call path to the novel method and its arguments, including types (information unavailable at lower levels of execution). It might examine the novel method's code, looking for suspicious calls. Such analysis could give a priority or confidence level to the alert. Beyond reporting, the system could use that information to guide a range of responses, from ignoring it, adding it to the profile, throwing an exception, or halting the thread. It would be straightforward to modify the VM so that a thrown exception could not be caught by the exploit itself. This would be accomplished using the same stack introspection mechanisms used by the traditional sandbox.

Another approach would be to borrow the response strategy from Somayaji's pH system [30]. pH uses exponentially increasing delays of subsequent system calls in the executing process. Small numbers of anomalies have small delays, and go unnoticed. Large numbers of temporally clustered anomalies, produced when programs execute novel code paths, have such large delays that the program essentially freezes. A small utility is available for users to "unfreeze" programs.

The appropriate response strategy depends on the exploits and the monitored programs. The dynamic sandbox described here seems best suited to server or middleware applications, where security is most needed and behavior usually limited. Interactive applications, which are closer to the user, might have functionality that is invoked at infrequent intervals, creating larger numbers of false positives. The first response strategy, providing more information for specific anomalies, might work well in server applications while the second is more appropriate for interactive applications.

5. FUTURE DIRECTIONS

In the future, we hope to concentrate on: optimizations to the current implementation, using more more diverse sensors, adding a response mechanism, and better characterizing the anomalies.

- Obvious optimizations to the current prototype are a more efficient log structure and to possibly store the native code as the profile itself.
- Beyond method invocations, we are currently investigating the behavior of the memory allocator and garbage collector. Profiling useful for adaptive garbage collection may also be useful for anomaly detection. We are able to predict many object lifetimes precisely and will see if this can be leveraged for our dynamic sandbox.
- A robust response mechanism, discussed in detail in Section 4, needs to be investigated and implemented.

- We are also interested in characterizing the anomalies, including the false positives. Do anomalies have any commonalities? Do anomalies generated by exploits look different from the false positives? Examining the anomalies may tell us something fundamental about program behavior.

Except for the performance optimizations, this future work relies on collecting more data, both on exploits and from real applications. Until more applications run on ORP, our analysis will rely on instrumented applications rather than full-fledged implementations of dynamic sandboxing.

6. CONCLUSIONS

We described a strategy, called dynamic sandboxing, which is an anomaly intrusion-detection system for applications running in Java-like environments. To show its efficacy, we implemented a prototype system using a limited amount of profiling information and tested it against two exploits. We presented additional arguments that our system satisfies three criteria of a successful IDS: high true-positive rate, low false-positive rate (zero for the tests reported here), and low performance penalties.

7. ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the National Science Foundation (grant ANIR-9986555), the Office of Naval Research (grant N00014-99-1-0417), Defense Advanced Projects Agency (grant AGR F30602-00-2-0584), the Intel Corporation, and the Santa Fe Institute.

We thank the Adaptive Computation group at the University of New Mexico for their help in developing these ideas, and Darko Stefanovic for discussions on benchmarking and the implementation of ORP.

8. REFERENCES

[1] Hostile applet home page. http://www.cigital.com/hostile-applets/index.html.

[2] Flaw in Microsoft VM JDBC classes could allow code execution (Q329077). http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS02-052.asp, 2002.

[3] M. Almgren and U. Lindqvist. Application-integrated data collection for security monitoring. In *Recent Advances in Intrusion Detection: 4th International Symposium, RAID 2001*, 2001.

[4] A. Appel and E. Felton. Princeton secure internet programming. http://www.cs.princeton.edu/sip/history.

[5] T. Ball and J. R. Larus. Using paths to measure, explain, and enhance program behavior. *Computer*, 33:57–66, 2000.

[6] D. Brumleve. Brown orifice. http://www.brumleve.com/BrownOrifice/.

[7] B. Cahoon and K. S. McKinley. Tolerating latency by prefetching Java objects. In *Workshop on Hardware Support for Objects and Microarchitectures for Java*, Austin, TX, October 1999.

[8] M. C. Carlisle and A. Rogers. Sofware caching and computation migration. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbra, CA, July 1995.

[9] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[10] M. Corp. Common language infrastructure. http://www.ecma.ch/ecma1/STAND/ecma-335.htm.

[11] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.

[12] S. Hofmeyr. *An Immunological Model of Distributed Detection and it application to Computer Security.* PhD thesis, University of New Mexico, 1999.

[13] S. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[14] L. C. Intl. Codebreakers-4. http://www.codebreakers.org, 1998.

[15] A. Klaiber. The technology behind crusoe processors. http://www.transmeta.com/crusoe/dnload/pdf/crusoetechwp.pdf, 2000.

[16] I. M. R. Labs. Open runtime platform. http://www.intel.com/research/mrl/orp/, 2000.

[17] J. R. Larus. Whole program paths. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[18] W. Lee, S. Stolfo, and P. Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI97 Workshop on AI Methods in Fraud and Risk Management*, 1997.

[19] L. LLC. Limewire. http://www.limewire.com.

[20] C. Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *New Security Paradigms Workshop 2000*, Cork, Ireland, 2000.

[21] G. McGraw and E. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, Inc, 1999.

[22] S. Microsystems. The Java HotSpot performance engine architecture. http://www.java.sun.com/products/hotspot/whitepaper.html, 1999.

[23] A. C. Myers, N. Nystrom, and L. Z. S. Zdancewic. Java + information flow. http://www.cs.cornell.edu/jif/.

[24] S. Oaks. *Java Security*. O'Reilly and Associates, 1998.

[25] C. Pfleeger. *Security in Computing*. Prentice Hall, 2 edition, 1997.

[26] I. Sanctum. Appsheild. http://www.sanctum.inc.com.

[27] R. S. Sielken. Application intrusion detection, 1999.

[28] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw Hill Inc, New York, 1994.

[29] A. Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, University of New Mexico, 2001.

[30] A. Somayaji. *Operating System Stability and Security through Process Homeostatis*. PhD thesis, University of New Mexico, May 2002.

[31] T. Technologies. Kaffe 1.0.6. http://www.kaffe.org, 2000.

[32] B. Vasanth, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming, Language Design and Implementation*, 2000.

[33] D. Wagner and D. Dean. Intrusion detection via static analysis. In *2001 IEEE Symposium on Security and Privacy*, 2001.

[34] 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.

[35] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Los Alamitos, CA, 1999. IEEE Computer Society.

[36] M. Welz and A. Hutchison. Interfacing trusted applications with intrusion detection systems. In *Recent Advances in Intrusion Detection: 4th International Symposium, RAID 2001*, 2001.