Ensuring the Integrity of Running Java Programs

Mark A. Thober, J. Aaron Pendergrass, and Andrew D. Jurik

The increased use of and reliance on computing devices elicits a desire to ensure the integrity of the software running on these devices. This desire is indeed well founded—malicious software has become a major problem for today's computer systems. Consequently, we have developed the Java Measurement Framework (JMF) for inspecting a running Java program and ensuring the program's integrity. JMF provides a mechanism for writing integrity policies about Java programs. Our implementation provides the capability to extract a measurement of the code and data of the running program and then evaluate this measurement against a policy. The result allows concerned parties to achieve a greater confidence in the integrity of the software running on their systems. We show how our system can be used on several real-world Java programs and with adequate performance overhead.

INTRODUCTION

Computing devices are increasingly relied on to store, manipulate, transmit, and visualize data. This reliance extends to nearly all aspects of modern society, from individuals who rely on their smartphones to always be at their fingertips to organizations with vast networks and racks of servers that must always be fully functional. Such reliance poses a great risk in the event that the software components of these devices are not operating as expected or required. Alterations to device software, which may occur either by accident or malice, bring into doubt the integrity of the software. A piece of software (e.g., a running process) is said to have integrity if it runs without improper system alterations.[1] Stakeholders would like to base their decisions on the operational integrity of the relevant running software. For example, a user may wish to know that there is no keyboard logger on a system before entering a password; a network access control point may wish to validate that only authorized computers may gain access to the network.

Previous work has shown that it is possible to measure software as it is loaded. These load-time systems (e.g., Refs. 2 and 3) are typically based on computing a cryptographic hash of the program image. However, these systems only show that the software was correct when it was started. They do not provide evidence to show that the software continues to operate as expected. This is an

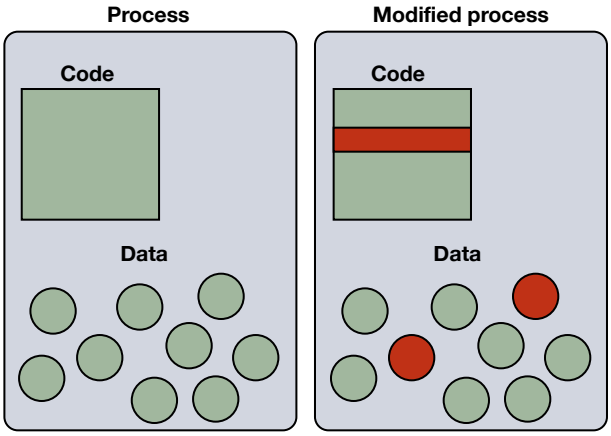

Figure 1. Notional depiction of runtime program modification.

especially relevant point because virtually all software is susceptible to either accidental or malicious alteration, which may bring into doubt the trustworthiness of the running software. To combat this problem, runtime measurement systems attempt to establish that the current state of a process is consistent with its expected execution.

Motivation

Figure 1 presents an abstract representation of a running process, which consists of code and data. A process that is modified in unexpected and possibly malicious ways may result in dire consequences. As an example of the red code modification illustrated in Fig. 1, consider the following piece of code, which authenticates a user.

```
1: public User authenticate(Auth auth) {
2: String user = auth.getUsername();
3: String password = auth.getPassword();
4: SendToURL("attacker.com",user+password);
5: ...
6: }
```

The code on line 4 has been inserted to forward the user's access credentials to a remote server. This integrity violation can clearly lead to a complete access control failure of a system because now an attacker could log in to a remote system that accepts these credentials.

Certainly, modifying the executable code of a process can cause great damage. Indeed, this is the purpose of many rootkits and trojan malware. However, an attacker can also cause great damage solely by modifying critical data within a process (as illustrated by the red circles in Fig. 1). As an example, we consider bluffin-muffin,[4] which is an open-source Texas hold 'em application written in Java consisting of a centralized server and multiple clients. If an attacker were able to modify the data within the poker server process, he could change the cards and thereby cheat at the game. Such a modification is depicted in Fig. 2, where user bob always gives himself the ace of hearts and ace of spades. Such a scenario is plausible and can have great impact given the prevalence of online gambling sites that use actual money.

Overview

The focus of most previous runtime integrity measurement systems[5–7] is primarily on measurement, which extracts the relevant data from the target; relatively little emphasis is placed on appraisal, which determines whether a measurement is accepted by some policy.

The main goals of our work are to establish a runtime measurement framework that focuses on the appraisal policies and explore programming language support for the definition and application of appraisal policies. Java is a suitable language for our purposes because the object-oriented nature of the language provides clear representation of data structures both in the source code and in the runtime environment. For those who are unfamiliar with the Java execution environment, we refer the reader to Fig. 3. Java processes run inside a Java virtual machine (JVM), which runs on top of an operating system (OS). The JVM runs alongside other user applications on the machine. For our purposes, it is the Java process (as highlighted in the figure) that is the target of measurement.

Figure 2. Screenshot of bluffin-muffin poker game, showing user bob's hand. Unwitting user alice is unaware that bob has maliciously modified the data representing the cards in the game.

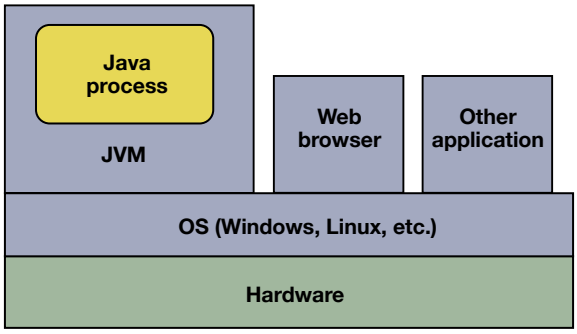

Figure 3. Execution environment. An OS runs on top of machine hardware. Applications run on top of the OS. One application is a JVM that runs a Java process, which is the target of measurement.

Adversary Model

To be explicit about the kinds of attacks we are trying to defend against, the following list describes our assumptions about the capabilities of the adversary.

- The attacker is able to modify the runtime bytecode of a Java application as well as any of its runtime data. (Java bytecode is the compiled version of the Java source program. The bytecode is therefore the sequence of instructions executed by the JVM.)
- We do not assume that a program always executes according to program code or the language semantics. In particular, we assume that an attacker has the capability to change arbitrary memory in the running Java process at any time.
- The runtime environment (i.e., the JVM and OS) is trusted when the measurement is taken. We anticipate our system running on a platform that uses other runtime measurement techniques to validate the integrity of the JVM and OS (e.g., Refs. 5 and 6).

Other approaches to runtime integrity monitoring (e.g., Refs. 8 and 9) rely on modifications to the actual bytecode of the target Java program. In these systems, the monitor is at the same privilege level as the target because monitoring is part of the Java process. In contrast, we are measuring the target from the JVM, which is outside the process. We do not trust the bytecode of the process and can therefore detect a more powerful adversary than these kinds of monitoring systems.

Java Measurement Framework

To combat the threats of code and data modifications, we have designed and implemented the Java Measurement Framework (JMF) for validating the integrity of running Java programs. Specifically, JMF can be used to ensure the following properties of a running Java application:

- Java bytecode running in the JVM is as expected.
- Application data structures are correct with respect to a policy.

In the remainder of this article, we provide a high-level description of the design and implementation of JMF and present an evaluation of the capabilities and performance overhead of the tool. Additional technical details of our work may be found in our previous article.[10]

DESIGN AND IMPLEMENTATION

We have implemented JMF using the Java language and runtime environment provided by OpenJDK 7.[11] Figure 4 depicts the key aspects of the implementation. The target Java process is running inside a JVM, as depicted on the left side of the figure. The memory of the Java process consists of three distinct elements: the PermGen space, which is the permanently generated portion of memory that contains the bytecode of the Java process and other immutable data; the heap, which contains all of the dynamically allocated data of the Java process; and the stack, which contains data relevant to the current execution context (e.g., local variables and the name of the functions currently being executed).

The implementation then provides for the taking of measurements of the process during runtime using a program called jmack. The jmack program runs on the target platform as a process managed by the OS (i.e., it is at the same level as the JVM). jmack interacts with the target JVM to extract structured representations of the target heap, stack, and PermGen space in the form of a heap dump, stack dump, and loaded classes, respectively. These three elements constitute the measurement.

To appraise the measurement, a programmer or other user is first required to write an integrity policy that describes the runtime constraints of the program; this is depicted as the integrity policy file in Fig. 4. The policy compiler part of the implementation transforms the policy directly into source code that integrates into the appraiser. This source code is merged with auxiliary code that parses Java application heaps and stacks; the code is then compiled to produce the policy appraiser. The class appraiser is a standard program that works for all Java processes. It takes as input a dump of loaded classes from jmack and a set of reference class files and compares them to ensure that the bytecode and other attributes of each Java class are identical.

In this section, we describe the components of JMF in detail, starting with the integrity policies and proceeding with details on the implementation of the measurement and appraisal capabilities.

Integrity Policies

Integrity policies represent the mechanism through which JMF is able to recognize unintended data modi-

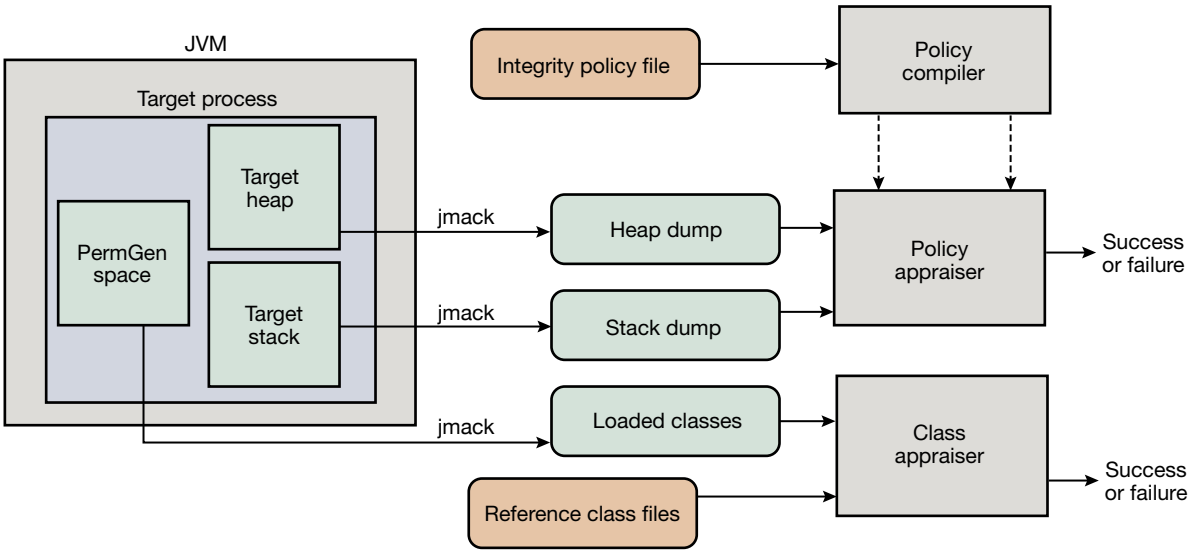

Figure 4. JMF system diagram. The policy is written and compiled into an appraiser program. The target Java process is periodically measured at runtime, and the appraiser is subsequently run on the measurement outputs and applicable policies.

fications. These policies are focused on the integrity of data only, not on the loaded code. Measuring the integrity of loaded code and the appraisal of these measurements are described in the *Measurement* and *Appraisal* sections. Verifying the integrity of loaded code is generic to all programs because all programs have code loaded in a similar manner, and the code can be extracted in the same way; moreover, loaded code is supposed to be immutable and can therefore be easily compared against known good code for a program. In contrast, program data are highly dynamic and vary greatly across different programs, even with separate executions of the same program. Hence, the integrity of data is specific to a particular program. To support this variability, we provide for the use of integrity policies for describing a set of constraints on the data within a program.

We define a syntax for writing integrity policies in Fig. 5. The syntax is similar to invariant specifications in Design By Contract[12] languages such as JML.[13] We chose to use a new syntax because it simplifies the presentation so that we can focus only on invariants and not on the other parts of these specification languages.

In our syntax, a policy consists of a number of specifications and qualifiers and an application of a boolean function. Note that the vertical bar | in Fig. 5 means "or," the overline means a list, and the turnstile ⊢ means that the function f returns true when applied to arguments based on the specification and qualifier. Specifications, *spec*, describe the types of objects on the heap to which the policy applies. Specifications are empty (ϵ), a universally quantified object type (read for all objects o of type τ that are contained in heap H), an existentially quantified object type (read there exists an object o of type τ that is contained in heap H), or a list of specifications. Qualifiers, *qual*, describe locations in the execution where the policy must hold. A qualifier is empty (ϵ), an object and method name *o.m*, a negation of a qualifier, a logical "or" of two qualifiers, or a logical "and" of two qualifiers. The definition of a boolean function, *fdef*, is consistent with the definition of methods in Java, only it must return a boolean. Thus, a function f has a list of formal parameters, each with its own type C and identifier *o*, and a body consisting of a sequence of statements {*s*}. The boolean function will be computed for objects matching the specification if all the qualifiers are satisfied.

$$
\begin{aligned}
spec &::= \epsilon \mid \forall o : \tau \in H \mid \exists o : \tau \in H \mid spec, spec \\
qual &::= \epsilon \mid o.m \mid \neg(qual) \mid qual \vee qual \mid qual \wedge qual \\
fdef &::= \textbf{boolean}\ f(\overline{C}\ \overline{o})\ \{\overline{s}\} \\
\rho &::= spec, qual \vdash f(\overline{e})
\end{aligned}
$$

specification
qualifier
function def.
policy

Figure 5. Integrity policy syntax.

Because appraisal of a measurement is purely functional (i.e., taking a measurement and producing a yes/no answer), the appraiser must not alter the measurement, and for this reason, our implementation does not allow method calls on objects in the measurement, because a method may mutate the state of the object. The statements in the function definitions cannot contain method invocations of the input arguments, and all object fields must be accessed directly.

One of the useful aspects of our framework is that the policy is independent of the source code of the application, so the application does not need to be recompiled. This supports the writing of policies for legacy applications and libraries. This also provides

$$\forall l : \texttt{LinkedList}, \neg(l.\texttt{add} \vee l.\texttt{remove}) \vdash \texttt{correctListSize}(l)$$

```
boolean correctListSize(LinkedList list) {
 int count = 0;
 Entry node;
 for (node = list.header.next; node != list.header; node = node.next) {
  count++;
 }
 return count == list.size;
}
```

Policy A.

flexibility in that any number of different policies may be written for the same application, based on the needs of the appraiser. Inline monitoring approaches such as Monitoring-Oriented Programming (MOP)[8] lack such flexibility and require recompilation of the source code to embed the monitor in the application.

Writing accurate integrity policies remains the responsibility of the policy writer, and JMF is not intended to determine whether a policy is the best policy for a given program. However, we do aim to provide a useful mechanism to help programmers write maintainable integrity policies for their programs.

Simple Example Policy

We now present a simple example policy to show how a useful policy is written. Realistic policies written on real Java applications are discussed in the *Evaluation* section.

The following policy (and corresponding function definition) states that every object of type `LinkedList` must have the same number of nodes on the list as stated by the size field in the object, except when the `add` or `remove` methods are in the process of manipulating the list. (The Java `LinkedList` class includes other methods for adding/removing elements that should be in the qualifier; these are omitted for brevity.)

This example illustrates the utility of qualifiers in our syntax. If a `LinkedList` object is in the process of having elements added to or removed from it, it is possible that traversing the list might not match with the size attribute on the `LinkedList` object. This would return a false-positive appraisal failure because the list update is allowed by the program. Therefore, our qualifier states that the execution is not currently in the `add` or `remove` method for the object in question (see Policy A).

Baseline Policies

Measurement systems may use a baseline measurement, taken while the process is known to be in a good state, as a parameter to the appraiser. Hence, the H specification is included in our syntax definition to allow support for separate measurement and baseline heaps. In the remainder of this article, we will refer to the measurement target heap as H_T and the baseline heap as H_B; when a baseline is not part of the policy, we assume the heap in question is the target heap and shall omit it from the policy and simply write H.

Baselining allows the appraisal policy to be more easily parameterized by values that either are unknown until runtime or may be tedious to enumerate explicitly. For example, one may wish to have a policy on a server application stating that the data structures holding the user access information are the same as a known-good program state, as shown in Policy B.

A useful feature of baselining is the ability to generate a baseline from a static configuration file. In the user access example, we may want to ensure that all User objects in the runtime have a corresponding entry in a configuration file. A separate baselining program could generate a baseline heap H_B from this static configuration file, so the server would not actually need to be booted to obtain a baseline.

Measurement

A runtime measurement of a Java process consists of three parts: a heap dump, thread stack dumps, and a dump of the hashes of loaded classes. We have implemented a stand-alone tool, jmack, that produces these measurements by attaching to a running JVM at any time after the process has been started. jmack is based on the JVM monitoring tools jmap and jstack. jmack is invoked with

$$\forall l1 : \texttt{User} \in H_T, \exists l2 : \texttt{User} \in H_B \vdash \texttt{HasSameAccess}(l1, l2)$$

```
boolean HasSameAccess(User l1, User l2) {
 return ((l1.name == l2.name) && (l1.access == l2.access));
}
```

Policy B.

the process identification (PID) of the target process and the desired location of the output measurements.

To obtain the measurements of the target Java process, the process must be run using our modified version of Open Java Development Kit (OpenJDK)[7] so that the JVM command is recognized when jmack invokes it on the target's JVM. Outside of running the necessary version of Java, jmack requires no further support from the compiler, class files, or target program to run on legacy software. Most annotation solutions (e.g., Ref. 14) require both access to and recompilation of source code. An unmodified JVM does possess the capability to dump the heap and stack traces to capture the same state, but the target process must suspend itself while the measurements occur. We later describe our modifications to the JDK that permit jmack measurements and also improve performance of the target during measurement.

The jmack functionality appears in the center of Fig. 4. At its core, jmack extracts a heap dump and thread stack traces (augmented with local variable information) from the JVM (note that these are the heap/stack for the Java application maintained by the JVM, not the heap/stack of the JVM maintained by the OS). It also outputs SHA-1 hashes of the loaded classes from the PermGen space of the target, so the appraiser may compare them with known good hashes that are generated from the Java class files.

We have combined the collection of all three measurements in one tool to provide an atomic snapshot, so the entire measurement represents a particular moment in time. To minimize the impact of measurement on the target process, we add a new virtual machine (VM) command to the JDK, `fork _ op`, that forks a target JVM process and calls several VM operations, namely to dump the target heap, stack, and bytecodes of the child process. The effects of the `fork _ op` command are depicted in Fig. 6. Forking the process uses the memory copy-on-write (COW) mechanism of Linux, which allows the target process to run while the measurement is performed on the forked process. COW has been shown elsewhere to be a useful mechanism to improve target performance while still guaranteeing atomicity of the measurement.[15] It is important to note that this is an OS-level fork of the JVM that is running the Java application; we do not need to implement any new COW functionality but instead make use of the existing COW implementation within Linux that occurs automatically with the fork.

The `fork _ op` JVM command invokes the fork system call. At that location in the call stack, the jmack has access to the target process state and can obtain the measurements from a target process clone while the target process runs in parallel. Once jmack initiates a fork of the target process, jmack begins capturing the state of the newly created process while the original target process proceeds.

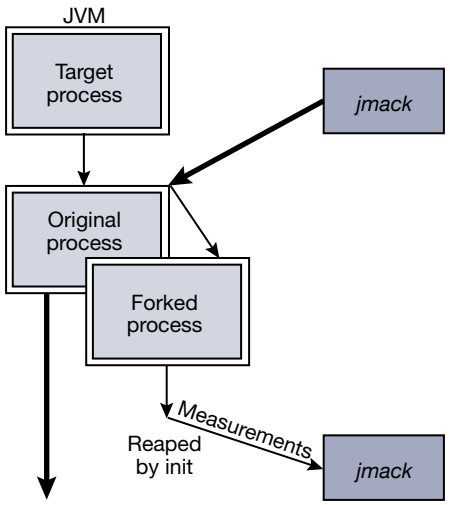

Figure 6. `fork _ op` on (modified) target JVM.

Appraisal

The appraiser consists of two separate components: the measurement appraiser generated by the policy compiler that verifies the integrity of the heap and stack dumps and a class appraiser that verifies the integrity of the loaded classes. The policy compiler takes a policy as input and produces Java source code. This Java code is added to a standard set of Java appraisal code to produce an appraiser that is specific to the policy. The standard appraisal code consists of code derived from the Java Heap Analysis Tool (jhat) that can parse heap dumps and stack traces and is augmented to support the relevant aspects of our policy language, including qualifiers and existential and universal quantifiers. The combined Java code for the appraiser is then compiled by a Java compiler to produce a program that serves as the appraiser. This program takes as input the snapshot from the measurement tool and outputs whether the policy has been satisfied.

Appraisal of a class measurement involves comparing the SHA-1 hashes contained in the measurement with SHA-1 hashes of the respective portions from class files. We use the class file parser from the Byte Code Engineering Library (BCEL)[16] to read in the class file information. We then compute a SHA-1 hash of the constant pool for each class file and SHA-1 hashes of the bytecodes of all methods contained in each class file. These hashes are then compared with the loaded class measurement, with any discrepancies causing a failure of the appraisal.

EVALUATION

We now provide some evaluation of our system, first by describing how JMF may be used to provide for the enforcement of meaningful integrity policies on real Java programs. We then provide a description of the runtime performance overhead of JMF.

bluffin-muffin

As discussed briefly in the *Introduction*, bluffin-muffin[4] is an open-source Texas hold 'em application consisting of a centralized server and multiple clients. In the example in Fig. 2, an attacker modified a player's cards without changes to the bytecode. The policy illustrated in Fig. 7 ensures that all cards in the players' hands and in the deck are unique; i.e., a player should not have the ace of spades if it also appears in the deck. This example involves multiple class instances (game, players, cards, etc.), showing that our system is suitable to describe and enforce sophisticated security policies. The policy is closely tied to the source code, as the field names (e.g., m_playing) are defined by the source program. Comments are included to aid in readability.

This example policy is for use on a server application because that is where the poker game data reside. This illustrates that JMF can be used to help clients verify that the server is acting in an acceptable manner. Similarly, a server may want to verify properties of a connected client to be sure that the client is acting properly as well. In the poker example, the server may wish to ensure that only valid client programs are running and no poker bots are being used. These kinds of client/server trust requirements are increasingly common, as in multiuser online gaming, financial software, and beyond. We believe our system, when paired with a proper attestation protocol, can have wide applicability to client/server trust relationships.

Apache FtpServer

Apache FtpServer[17] is an open-source FTP server written in Java. An important property of the FTP server is that all the user account information should be as expected and not be modified. The `PropertiesUserManager` object contains a `HashTable` that keeps track of all of the usernames and properties for those users (such as `enableflag` and `writepermission`, for each account). An example of a policy that uses a baseline measurement for comparison of account information properties is shown in Fig. 8. It ensures that every object of type `PropertiesUserManager` in the target has a corresponding object of the same type in the baseline

∀ g:poker.game.PokerGame ⊢ noCardRepeat(g)

```
boolean noCardRepeat(poker.game.PokerGame g) {
for (int i = 0; i < maxPlayers; i ++) { // Loop over all players
 JavaObject p1 = arrPlayers[i]; // Player 1 Object
 if ((p1 != null) && (p1.m_playing || p1.m_allIn)) { // Is Player 1 playing?
  JavaObjectArray playersHand = p1.m_cards.cards.elementData; // Player 1 hand Object
  int p1c1 = playersHand[0].m_ID; // Player 1 First Card
  int p1c2 = playersHand[1].m_ID; // Player 1 Second Card
  // Make sure no other player has one of Player 1's cards
  for (k = (i+1); k < maxPlayers; k++) { // Loop over all other players
   JavaObject p2 = arrPlayers[k]; // Player 2 Object
   if ((p2 != null) && (p2.m_playing || p2.m_allIn)) { // Is Player 2 playing?
    JavaObjectArray player2Hand = p2.m_cards.cards.elementData; // Player 2 hand Object
    int p2c1 = player2Hand[0].m_ID; // Player 2 First Card
    int p2c2 = player2Hand[1].m_ID; // Player 2 Second Card
    if ((p1c1==p1c2) || (p1c1==p2c1) || (p1c1==p2c2) || (p1c2==p2c1) || (p1c2==p2c2)) {
     // Two of the cards are the same.  Something bad happened
     return false;
    }
   }
  }
  // Make sure none of Player 1's cards is also in the deck
  for (j = 0; j < dealerDeck.size; j++) { // Trace through deck
   int dcard = dealerDeck.elementData[j].m_ID;
   if ((dcard == p1c1) || (dcard == p1c2)) {
    // One of Player 1's cards is in the deck.  Something bad happened
    return false;
   }
  }
 } // End loop over players
// No duplicate cards, so return true
return true;
}
```

Figure 7. bluffin-muffin card uniqueness policy.

$$\forall \texttt{pTgt} : \texttt{PropertiesUserManager} \in H_T, \exists \texttt{pBase} : \texttt{PropertiesUserManager} \in H_B \vdash \texttt{SameAccts}(\texttt{pTgt}, \texttt{pBase})$$

```
boolean SameAccts(PropertiesUserManager pTgt, PropertiesUserManager pBase) {
 for (int i=0; i < pTgt.userDataProp.table.length; i++) {
  String key = pTgt.userDataProp.table[i].key;
  String value = pTgt.userDataProp.table[i].value;
  if (!value.equals(HashTableGet(pBase.userDataProp.table,key))) {
   return false;
  }
 }
 return true;
}
```

Figure 8. FTP server policy.

and that the property values are the same. To improve the readability, the example policy in this figure does not use full classpaths; the helper function `HashTableGet` is omitted for brevity.

Performance

To demonstrate the practicality of our implementation, we have performed a series of benchmarks showing we can perform runtime measurement of a Java process with reasonable overhead. We now describe our experimental methodology and analyze the performance results.

We carried out our experiments on a desktop computer running 32-bit Ubuntu 10.04 with 2 GiB of memory and dual Intel Pentium D 3.20-GHz processors with a modification of the OpenJDK Client VM (1.7.0). We used a subset of the DaCapo-9.12-bach benchmark suite,[18] which includes several open-source Java programs. The subset we used included the Java programs avrora, eclipse, h2, jython, lusearch, pmd, sunflow, tradebeans, tradesoap, and xalan. We ran each benchmark five times.

Performance Results

We evaluate the performance of measuring a process using JMF's jmack tool, including the COW implementation using fork. The sequence of steps is as follows:

1. User starts the target process.
2. User invokes jmack with the process ID of the target process.
3. jmack executes the `fork_op` VM command within the target process JVM. The target process then continues unabated.
4. The forked process runs in parallel to the original process to capture the state snapshot and then terminates; the target JVM is momentarily paused while the measurement transfers to the forked JVM.

Figure 9 shows the execution times of each of the benchmarks for various delays between measurements; we report the sample mean values. The sample standard deviations are small relative to the sample mean values. For example, on average across the set of experiments, the sample standard deviation is 2.4% of the mean value with a range of 0.3–12.8%. "Delay = 0 s" means that once jmack completes a measurement, another measurement is immediately initiated. A positive delay means that the process is allowed to run for the specified number of seconds before another measurement is taken. We restrict test values up to "Delay = 10 s" because the shortest benchmark takes ~10 s to run on our experimental setup. Further increasing the delay reduces the overhead.

We calculate the runtime overhead by comparing the execution times of a particular execution scenario and the "No Measurement" (i.e., with no measurements being taken) execution scenario. The average overhead across all the benchmarks for "Delay = 0 s" is 38.0%, and for "Delay = 5 s," it is 6.8%. For "Delay = 10 s," the average overhead shrinks further to a more manageable 3.0%.

The question of the frequency of measurements can be reduced to a trade-off between performance and concerns about the power of the adversary. An adversary who uses measurement infrequency as an attack vector must compromise the target between measurements and exit the target when a measurement occurs (an adversary who makes changes that are not part of the integrity policy can obviously avoid detection). Because measurement is best suited for persistent changes to code and data structures, we believe the time between measurements need not be small. In many cases, it may best be used as part of an attestation scenario (e.g., where one party of a network connection wants to ascertain whether another party is operating as expected before connecting).[19]

DISCUSSION

The primary use of JMF is to ensure the integrity of applications as part of a "defense in depth" security approach, with other components ensuring the correct operation of the environment (JVM and OS). Our tool also has other uses. Once a policy is developed, JMF can be used as a debugging tool to ensure that

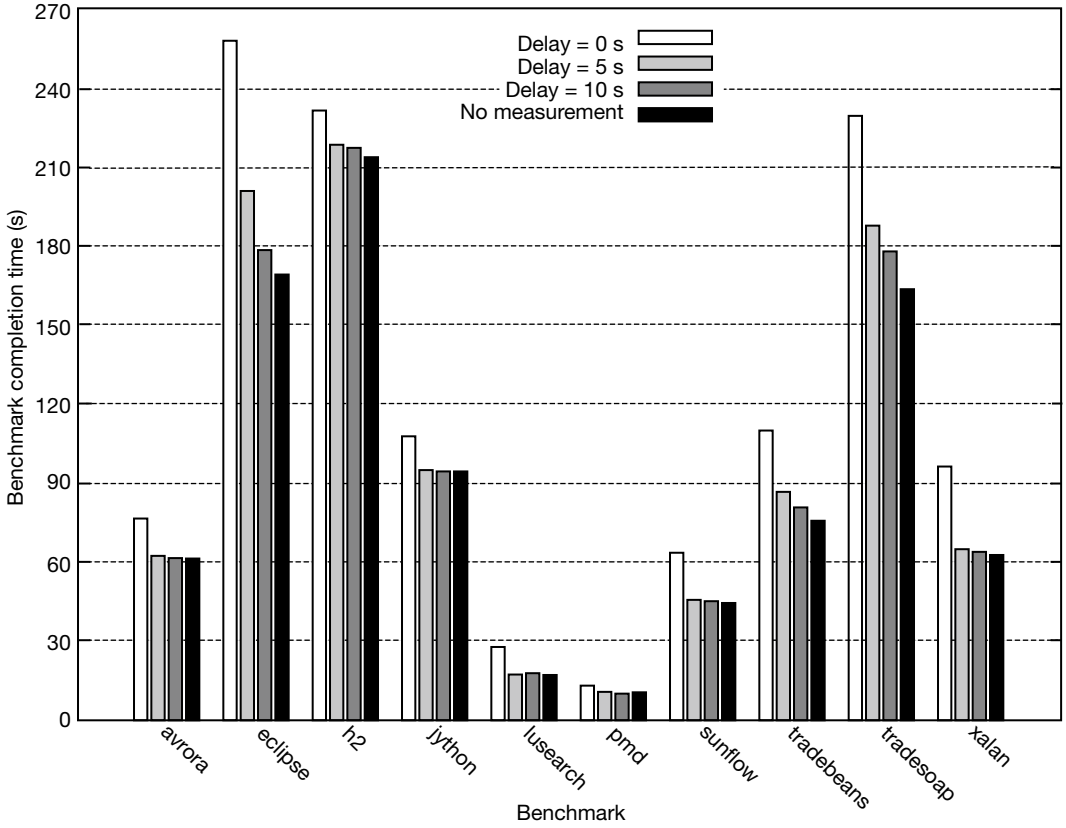

Figure 9. Execution times of the DaCapo benchmarks when the amount of delay between measurements varied. The execution times of the benchmarks vary from as short as 10 s to more than 3 min, and as the delay between jmack invocations increases, the execution times decrease.

desired properties actually hold at runtime. It is also possible to use JMF in concert with other systems. For example, if one wants to use a security monitor that is integrated into the Java bytecode of an application, JMF may be used to ensure that this bytecode has not been modified.

Related Work

We now provide a brief description of related work on integrity policies for kernel data and monitoring systems; additional discussion of related work may be found in our other article.[10] Several prior works developed systems to provide runtime integrity measurement[5–7] of a kernel; these works focused primarily on the measurement component, with little emphasis on appraisal. There has so far been little work on general techniques for specifying integrity policies. Petroni et al.[20] describe architecture for defining integrity specifications suited specifically for dynamic kernel data. Because they are concerned with low-level compiled C code, they must describe how the data are organized in memory and build up a model that abstracts the low-level data into a more understandable form. Because our system is object-oriented, the data structures are explicitly represented both in the code and in the runtime, thus avoiding the step of representing data structures in a model, and our policies are described directly over the program source code.

Integrity measurement as described is closely related to a significant body of work on invariant monitoring; Delgado et al.,[21] for example, present an overview of software-fault monitoring tools, and Parno et al.[22] categorize and explain extant approaches to bootstrapping trust. Projects such as MOP,[8] InvTS,[9] Tracematches,[23] Jahob,[24] and Java-MaC[25] provide inline invariant enforcement by instrumenting a program's source code based on a policy specification similar to that used by JMF. The TrustedVM implementation of Haldar et al.'s semantic remote attestation concept[26] provides a modification of the JVM to monitor protocols. JMF provides a more comprehensive implementation and focuses on application data structures. Furthermore, JMF can be run on an unmodified, already-running process and considers only the state captured in a snapshot when making policy appraisals.

Measurement versus Monitoring

JMF is a runtime measurement system. A measurement system attempts to validate the current state of a target at a particular moment in time, independent of any previous behavior. In contrast, a monitoring system

attempts to continuously validate some properties of a target while it executes. We now provide a general comparison of the two approaches.

The instrumented code in monitoring systems maintains an abstract model of the target program's policy-relevant state that may be consulted to validate the program's execution steps. This provides the advantage of ensuring that the target program never enters a state that is inconsistent with the policy. Although this may seem like the ideal case, for performance reasons, the policy must always reflect a relatively small subset of the program's actual state; validating the entire state of a program after every execution step is clearly infeasible. In addition, it is essential to the correctness of the monitor that the system is unable to make a policy-relevant execution step without consulting the monitor. If the system is allowed to make even a single unmonitored execution step, the monitor's model of the program state will no longer be valid, and thus all future decisions of the monitor may be suspect. This makes monitoring-based systems vulnerable to state changes outside of the expected execution model of the system, such as code injection attacks or direct memory access.

The ability of a measurement to recognize an invalid state is limited only by its ability to inspect the state of its target. This makes measurement systems immune to the vulnerabilities described for monitors but leaves open the potential for the system to pass through intermediate invalid states without detection. Moreover, because measurement is periodic, performance concerns are fewer for a measurement system, and the measurement can therefore include a much larger part of the target's state than a continuous monitoring system.

Future Work

In future work, we plan to improve our implementation in several ways. We aim to make a more efficient measurer that only collects the portions of the heap that are relevant to a policy and to improve the appraiser to more efficiently traverse the target heap. We also plan to make general improvements to the compiler to ensure support of all of Java's language features.

Although we largely assume that an acceptable policy is available within the context of the system, drafting an effective policy is a challenging task. Policies may require domain expertise and suffer from many of the same pitfalls that software encounters. For this reason, automatic policy generation may be helpful in identifying not only useful invariants but also starting places upon which to develop more sophisticated policies. Other systems have already been used to identify invariants automatically based on traces of execution.[27–30] We plan to explore how these techniques can be applied to JMF.

We also plan to modify our policy compiler to support integration of the original source code when producing the appraiser. In many cases, our policies replicate code that already exists in the original source, such as looking up entries in `HashTable` objects or obtaining the data fields of objects. We aim to improve the efficiency of writing policies by allowing the policy writer to more easily make use of functionality that is already implemented in the program's source code.

We will also investigate how the concepts of JMF may apply to other languages and determine which other languages may have features that are useful for measurement. Whether an integrity policy is sufficient for justifying process integrity remains an open problem. Hence, we plan to consider techniques to improve the confidence that a policy fully covers the integrity requirements of a program. We intend to explore programming language concepts that can help a programmer write programs that are more amenable to measurement and create metrics that may be used to evaluate the robustness of an integrity policy.

REFERENCES

[1] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C., "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur Comput.* **1**(1), 11–43 (2004).

[2] Jaeger, T., Sailer, R., and Shankar, U., "PRIMA: Policy-Reduced Integrity Measurement Architecture," in *ACM Symp. on Access Control Models and Technologies (SACMAT)*, Lake Tahoe, CA, pp. 7–9 (2006).

[3] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L., "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *Proc. USENIX Security Symp.*, San Diego, CA, pp. 223–238 (2004).

[4] bluffin muffin, *Open-Source Poker Game*, http://code.google.com/p/bluffin-muffin/.

[5] Loscocco, P. A., Wilson, P. W., Pendergrass, J. A., and McDonell, C. D., "Linux Kernel Integrity Measurement Using Contextual Inspection," in *Proc. 2007 ACM Workshop on Scalable Trusted Computing (STC)*, Alexandria, VA, pp. 21–29 (2007).

[6] Petroni, N. L. Jr., Fraser, T., Molina, J., and Arbaugh, W. A., "Copilot—A Coprocessor-Based Kernel Runtime Integrity Monitor," in *Proc. USENIX Security Symp.*, San Diego, CA, pp. 179–194 (2004).

[7] Petroni, N. L. Jr., and Hicks, M., "Automated Detection of Persistent Kernel Control-Flow Attacks," in *Proc. ACM Conf. on Computer and Communications Security (CCS)*, Alexandria, VA, pp. 103–115 (2007).

[8] Chen, F., and Roşu, G., "MOP: An Efficient and Generic Runtime Verification Framework," in *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Montreal, Canada, pp. 569–588 (2007).

[9] Gorbovitski, M., Rothamel, T., Liu, Y. A., and Stoller, S. D., "Efficient Runtime Invariant Checking: A Framework and Case Study," in *Proc. International Workshop on Dynamic Analysis (WODA)*, Seattle, WA, pp. 43–49 (2008).

[10] Thober, M., Pendergrass, J. A., and Jurik, A. D., "JMF: Java Measurement Framework: Language-Supported Runtime Integrity Measurement," in *Proc. 7th ACM Workshop on Scalable Trusted Computing (STC)*, New York, NY, pp. 21–32 (2012).

[11] Sun Microsystems, *OpenJDK*, http://openjdk.java.net/.

[12] Meyer, B., "Applying 'Design by Contract'," *Computer* **25**(10), 40–51 (1992).

[13] Leavens, G. T., Baker, A. L., and Ruby, C., "JML: A Notation for Detailed Design," Chap. 12, *Behavioral Specifications for Businesses and Systems*, H. Kilov, B. Rumpe, and W. Harvey (eds.), Kluwer Academic Publishers, Boston, MA, pp. 175–188 (1999).

[14] Chen, F., and Roşu, G., "Java-MOP: A Monitoring Oriented Programming Environment for Java," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Edinburgh, UK, pp. 546–550 (2005).

[15] Thober, M., Pendergrass, J. A., and McDonell, C. D., "Improving Coherency of Runtime Integrity Measurement," in *Proc. 3rd ACM Workshop on Scalable Trusted Computing (STC)*, Fairfax, VA, pp. 51–60 (2008).

[16]Apache Software Foundation, *The Byte Code Engineering Library*, http://jakarta.apache.org/bcel/.
[17]Apache Software Foundation, Apache FtpServer, http://mina.apache.org/ftpserver-project/.
[18]Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., et al., "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, pp. 169–190 (2006).
[19]Coker, G., Guttman, J., Loscocco, P., Sheehy, J., and Sniffen, B., "Attestation: Evidence and Trust," in *Proc. 10th International Conf. on Information and Communications Security*, Vol. 5308 of *Lecture Notes in Computer Science*, L. Chen, M. Ryan, and G. Wang (eds.), Springer-Verlag, Berlin/Heidelberg, pp. 1–18 (2008).
[20]Petroni, N. L. Jr., Fraser, T., Walters, A., and Arbaugh, W. A., "An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data," in *Proc. USENIX Security Symp.*, Vancouver, Canada, pp. 289–304 (2006).
[21]Delgado, N., Gates, A., and Roach, S., "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools," *IEEE Trans. Softw. Eng.* **30**(12), 859–872 (2004).
[22]Parno, B., McCune, J., and Perrig, A., "Bootstrapping Trust in Commodity Computers," in *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, pp. 414–429 (2010).
[23]Avgustinov, P., Bodden, E., Hajiyev, E., Hendren, L., Lhoták, O., et al., "Aspects for Trace Monitoring," in *Formal Approaches to Software Testing and Runtime Verification*, K. Havelund, M. Núnez, G. Rosu, and B. Wolff (eds.), Springer, Seattle, WA, pp. 20–39 (2006).
[24]Zee, K., Kuncak, V., Taylor, M., and Rinard, M., "Runtime Checking for Program Verification," in *Proc. International Conf. on Runtime Verification (RV)*, Vancouver, Canada, pp. 202–213 (2007).
[25]Kim, M., Kannan, S., Lee, L., Sokolsky, O., and Viswanathan, M., "Java-MaC: A Run-Time Assurance Tool for Java Programs," *Electron. Notes Theor. Comput. Sci.* **55**(2), 129–155 (2001).
[26]Haldar, V., Chandra, D., and Franz, M., "Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing," in *Proc. 3rd Conf. on Virtual Machine Research and Technology Symp.*, Vol. 3, Berkeley, CA, pp. 29–41 (2004).
[27]Baliga, A., Ganapathy, V., and Iftode, L., "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *Proc. 24th Annual Computer Security Applications Conf. (ACSAC)*, Anaheim, CA, pp. 77–86 (2008).
[28]Csallner, C., Tillmann, N., and Smaragdakis, Y., "DySy: Dynamic Symbolic Execution for Invariant Inference," in *International Conf. on Software Engineering (ICSE)*, Leipzig, Germany, pp. 281–290 (2008).
[29]Demsky, B., Ernst, M. D., Guo, P. J., McCamant, S., Perkins, J. H., and Rinard, M., "Inference and Enforcement of Data Structure Consistency Specifications," in *Proc. International Symp. on Software Testing and Analysis (ISSTA)*, Portland, ME, pp. 233–244 (2006).
[30]Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., et al., "Automatically Patching Errors in Deployed Software," in *Proc. ACM SIGOPS 22nd Symp. on Operating Systems Principles (SOSP)*, Big Sky, MT, pp. 87–102 (2009).

The Authors

Mark A. Thober was the project Principal Investigator (PI) for the JMF task, which was an Independent Research and Development project jointly funded out of a Laboratory Cross Enterprise Initiative and the Cyber Operations Mission Area. Dr. Thober is a computer scientist doing computer security research in APL's Asymmetric Operations Department. He worked on the basic research for JMF, wrote parts of the implementation, and provided guidance throughout the task. **J. Aaron Pendergrass** is also a computer scientist in APL's Asymmetric Operations Department. He worked closely with Dr. Thober on the initial research and design for JMF and wrote prototype code for the implementation, such as the policy compiler. **Andrew D. Jurik** is a software engineer in APL's Asymmetric Operations Department. He was the primary software developer on the JMF task and wrote a significant part of the software for the system prototype; he was responsible for many of the performance improvements to the system. For further information on the work reported here, contact Mark Thober. His e-mail address is mark.thober@jhuapl.edu.

The *Johns Hopkins APL Technical Digest* can be accessed electronically at **www.jhuapl.edu/techdigest**.