Linux Kernel Integrity Measurement Using Contextual Inspection

Peter A. Loscocco
National Security Agency
loscocco@tycho.nsa.gov

Perry W. Wilson
The Johns Hopkins University
Applied Physics Laboratory
perry.wilson@jhuapl.edu

J. Aaron Pendergrass
The Johns Hopkins University
Applied Physics Laboratory
james.pendergrass@jhuapl.edu

C. Durward McDonell
The Johns Hopkins University
Applied Physics Laboratory
durward.mcdonell@jhuapl.edu

ABSTRACT

This paper introduces the Linux Kernel Integrity Monitor (LKIM) as an improvement over conventional methods of software integrity measurement. LKIM employs *contextual inspection* as a means to more completely characterize the operational integrity of a running kernel. In addition to cryptographically hashing static code and data in the kernel, dynamic data structures are examined to provide improved integrity measurement. The base approach examines structures that control the execution flow of the kernel through the use of function pointers as well as other data that affect the operation of the kernel. Such structures provide an efficient means of extending the kernel operations, but they are also a means of inserting malicious code without modifying the static parts. The LKIM implementation is discussed and initial performance data is presented to show that contextual inspection is practical.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Secutiry and Protection; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Measurement, Security

Keywords

Integrity Measurement, LKIM, System Monitoring, Attestation Systems

1. INTRODUCTION

The computer industry is making a considerable investment to provide a greater level of confidence in the integrity of computing platforms. This response to a general desire for a safer computing environment is producing hardware support for reporting what software is running on the platform. This notion of integrity is focused on identifying the specific pieces of software loaded on the platform and assumes that the appropriateness of the software has been determined by some other means. Thus, if the configuration of the platform has changed beyond that approved for use by the owner, then the platform integrity is degraded. Using this notion of integrity, one can focus on the challenge of reliably measuring and reporting the software configuration of the platform.

To better understand this challenge, it is helpful to consider the system aspects of integrity measurement. The process of verifying a platform's integrity can be decomposed into three basic steps: identifying the software through measurement, reporting the measurement, and comparing the measurement against a standard of integrity. Therefore, the logical architecture for an integrity measurement system (IMS) has three components: the target of measurement, a measurement agent, and a decision maker. This notion is useful in discussing the properties of an IMS and how it might be improved to provide greater confidence in the platform integrity. A set of desired properties are given in section 2 that have been found useful in the design of improved techniques for measurement.

One prominent framework of integrity measurement is that promoted by the Trusted Computing Group (TCG), an industry consortium creating specifications for hardware and software components that make up an IMS. The standard of measurement used by the TCG is the SHA-1 cryptographic hash of critical components as they are loaded into the system. The TCG guidance for measurement includes the Trusted Platform Module (TPM) [20] as a hardware device to securely store and report measurement values in the form of a SHA-1 hash. This architecture seems to have gained acceptance and the TPM is becoming standard on personal computing platforms.

The TCG architecture provides a good framework for determining the integrity during software initialization; however, it does not address concerns about loss of platform integrity from runtime attacks against the system. A true statement of integrity is dependent upon the state of the software as it exists in memory. Unless it has been proven

STC'07, November 2, 2007, Alexandria, Virginia, USA.

that the system software is not vulnerable during its operation either through it's interfaces or the hardware and software environment, integrity measurement needs to be repeated periodically. This assertion presents a new set of challenges. First, the benign execution environment that existed at boot time and provided a safe place to perform measurement can't easily be reproduced without actually rebooting the system. Second, transformations on the software image after it is loaded will cause its hash value to change. In addition to subtle changes in the executable code that can occur, there are dynamic changes in the data that cannot easily be characterized with a hash value.

The Linux Kernel Integrity Monitor (LKIM) is designed to address these challenges and provide other benefits that make for a significant step forward in IMS technology. It leverages virtualization technology to provide separation between the measurement agent and the target of measurement. LKIM employs *contextual inspection* to measure components of the running kernel that cannot be easily represented with a hash. The kernel data structures that may be critical to correct operation have been identified through analysis, are examined during measurement, and represented as a graph that can presented to a decision maker at different levels of detail and completeness. This approach, as described in section 4.1, improves the security benefits of measurement and supports privacy concerns by enabling greater control over the release of measurement data. LKIM can detect a class of rootkits that can't be detected by conventional means, section 5 discusses the adore-ng rootkit. The remainder of this paper reviews measurement properties, discusses related work, and describes the features of LKIM.

2. MEASUREMENT PROPERTIES

LKIM serves as a measurement agent for a general purpose integrity measurement system intended to support many different measurement scenarios [17]. This system is capable of selecting from an extensible set of attestation scenarios governed by a system policy, taking into account requirements for security and privacy. In performing particular attestations, this IMS is capable of selecting appropriate measurement techniques by initiating appropriate measurement agents. Complete details of the entire IMS and the supported attestation scenarios are beyond the intended scope of this paper. The measurement properties disclosed in [10] provide a framework for discussion of LKIM. The generic architecture includes a target system who's integrity is in question, a measurement agent (MA) that inspects the target and a decision maker (DM) that interprets the output to determine the integrity of the target. The measurement component of an IMS should:

1. Produce *Complete* results. A MA should be capable of producing measurement data that is sufficient for the DM to determine if the target is the expected target as required for all of the measurement scenarios supported by the IMS.

2. Produce *Fresh* results. A MA should be capable of producing measurement data that reflects the target's state recently enough for the DM to be satisfied that the measured state is sufficiently close to the current state as required for all of the measurement scenarios supported by the IMS.

3. Produce *Flexible* results. A MA should be capable of producing measurement data with enough variability to satisfy potentially differing requirements of the DM for the different measurement scenarios supported by the IMS.

4. Produce *Usable* results. A MA should be capable of producing measurement data in a format that enables the DM to easily evaluate the expectedness of the target as required for all of the measurement scenarios supported by the IMS.

5. Be *Protected* from the target. A MA should be protected from the target of measurement to prevent the target from corrupting the measurement process or data in any way that the DM cannot detect.

6. *Minimize* impact on the target. A MA should not require modifications to that target nor should its execution negatively impact the target's performance.

3. RELATED WORK

Integrity measurement for system security is still an active area of research. Initial efforts have developed products that enable IT departments to control and monitor software configuration for deployed systems. This type of IMS inspects the filesystem of the target platform to provide assurance that the system files are up-to-date and have not been tampered with. More recent efforts have focused on providing stronger evidence that the system is running the software approved for use. Hardware enhancements are being leveraged to provide greater confidence in the authenticity of measurement reporting. Several of these existing measurement systems are discussed relative to the desired properties for general purpose measurement.

Tripwire [7] was an early integrity monitoring tool. It allowed administrators to statically measure systems against a baseline. Using Tripwire enables complete integrity measurement of filesystem objects such as executable images or configuration files. These measurements, however, cannot be considered complete for the runtime image of processes. Tripwire provides no indication that a particular file is associated with an executing process, nor can it detect the subversion of a process.

Tripwire performs well with respect to freshness of measurement data, and the impact on the target of measurement. Remeasurement is possible on demand, enabling the window for attack between measurement collection and decision making to be quite small. Since Tripwire is an application, installation is simple and its execution has little impact on the system. But because it is an application, the only protection available is that provided by the target system, making Tripwire's runtime process and results vulnerable to corruption or spoofing.

Tripwire is also limited with respect to flexibility and usability. Decision makers may only base decisions on whether or not a file has changed, not on the way in which that file has changed. Tripwire cannot generate usable results for files which may take on a wide variety of values. These limitations are generally characteristic of measurement systems that rely on hashes, making them most effective on targets not expected to change.

Systems like the CASS Security Kernels [12], TrustedBox [4], IMA [15] and systems like Prima [6] that build upon

IMA's concepts compare similarly to Tripwire when considered with respect to the described properties, but they do offer significant improvements. Their biggest advance is the protection of the measurement system and its data. These systems rely on modified kernels rather than user-land processes, making them immune to many purely user-space attacks that might subvert the Tripwire process. However, they are still vulnerable to many kernel-level attacks. IMA uses the TPM to detect subversion of the measurement results by comparing a hash value stored in the TPM with the expected value generated from the measurement system's audit log.

Like Tripwire, none of these systems makes truly complete measurements of running processes because the results only reflect static portions of processes before execution begins. But the measurements are more complete since running processes can be associated with the recorded hash values. These systems make no attempt to capture the current state of running processes or to revalidate the hash values of executable images. This means that fresh measurements cannot be provided to any decision process requiring updated measurements of running processes.

PRIMA extends the IMA concept to better minimize the performance impact on the system. By coupling IMA to SELinux policy [9], the number of measurement targets can be reduced to those that have information flows to trusted objects. This may also aid completeness in that measurement targets can be determined by policy analysis. The requirement for trusted applications to be PRIMA aware and required modifications to the operating system are development impacts on the target.

CoPilot [13] pushes the bar with respect to completeness, freshness and protection. Cryptographic hashes are still used to detect changes in measured objects, but unlike other systems, CoPilot's target of measurement is not the static image of a program and configuration files but the memory image of a running system. It also attempts to verify the possible execution paths of the measured kernel. The ability to inspect the runtime memory of the target is an improvement over filesystem hashes because it enables decisions about runtime state. Protection from the target is achieved by using a physically separate processing environment in the form of a PCI expansion card with a dedicated processor.

Although a considerable advance, CoPilot fails as a complete runtime IMS in two key ways. It cannot convincingly associate hashed memory regions with those actually in use by the target. It can only measure static data in predefined locations; dynamic state of the target is not reflected. The requirement of additional hardware in the target environment also impacts the target.

NFORCE [3], like CoPilot, was an attempt to develop a measurement approach that strives for completeness, freshness, and protection from the target. It measures the in-core image of the Linux operating system and its executing processes. The advantage of NFORCE is that it leverages the SMM [5] feature of the Intel CPU architecture to provide a protected execution environment from which to measure without the need for a separate coprocessor. From this environment, NFORCE can inspect the state of the CPU and determine which memory regions are actually being used by the target, providing a tight association between measurements and the target.

Like the other systems, NFORCE's measurements are limited to static memory regions in predefined locations. Freshness is achieved by periodic remeasurement triggered by a timer that generates SMM interrupts. Running in the limited SMM environment inhibits the usability and flexibility of NFORCE's results since it does not have a convenient way to report measurements to an external entity. Although NFORCE achieves comparable results to CoPilot without introducing new hardware, performance impact on the target is greater since CPU resources must be shared. However, this impact is reduced by performing measurement incrementally, spreading the impact over time.

Other measurement systems have been developed. Unlike those discussed so far, some use computations on or about the target system rather than employ a more traditional notion of measurement such as hashing. One such system is Pioneer[16]. It attempts to establish a dynamic root of trust for measurement without the need of a TPM or other hardware enhancements. The measurement agent is carefully designed to have a predictable run time and an ability to detect preemption. The measurement results can be fresh but are far from a complete characterization of the systems. Although in theory, this approach could support more complete measurement as long as the property of preemption detection is preserved.

Pioneer was designed to detect attempts of the target to interfere with the measurement agent, but it requires a difficult condition that the verifier be able to predict the amount of time elapsed during measurement. The impact on the target system can also be great because in order to achieve the preemption detection property, all other processing on the target has to be suspended for the entire measurement period.

Semantic integrity is a measurement approach targeting the dynamic state of the software during execution therefore providing fresh measurement results. Similar to the use of language-based virtual machines for remote attestation of dynamic program properties [2], this approach can provide increased flexibility for the challenger. If the software is well understood, then semantic specifications can be written to allow the integrity monitor to examine the current state and detect semantic integrity violations. This technique alone will not produce complete results as it does not attempt to characterize the entire system, but it does offer a way in which integrity evidence about portions of the target not suitable for measurement by hashing can be produced.

Such an approach has been shown effective in detecting both hidden processes and SELinux access vector cache inconsistencies in Linux [14]. A very flexible system was produced that can be run at any time to produce fresh results and that is easily extended to add new specifications. Better completeness than is possible from just hashing is achieved since kernel dynamic data is measured, but no attempt was made to completely measure the kernel. Completeness can only come with many additional specifications. Like CoPilot, a separate hardware environment was used to protect the measurement system from the target and to minimize the impact on the target at the cost of having extra hardware installed. However, it is subject to the same limitations as CoPilot.

4. THE LINUX KERNEL INTEGRITY MONITOR

LKIM uses *contextual inspection* to more completely characterize the Linux kernel. It produces detailed records of the state of security relevant structures within the Linux kernel, and can reasonably be extended to include additional structures without a dramatic leap in technology. LKIM can not only produce measurements at system boot time but also in response to system events or demand from the IMS. Measurement data is stored in a useful format that allows an IMS to retrieve any or all of the raw data. The IMS can use the data, or any transformation on any portion of it, according to the requirements of particular measurement scenarios and under the control of system policy.

Although LKIM's implementation is specific to measuring the Linux kernel, the techniques it employs are general and should apply equally well to other operating systems and complex software systems needing measurement. This technique is currently being applied to the Xen [1] hypervisor.

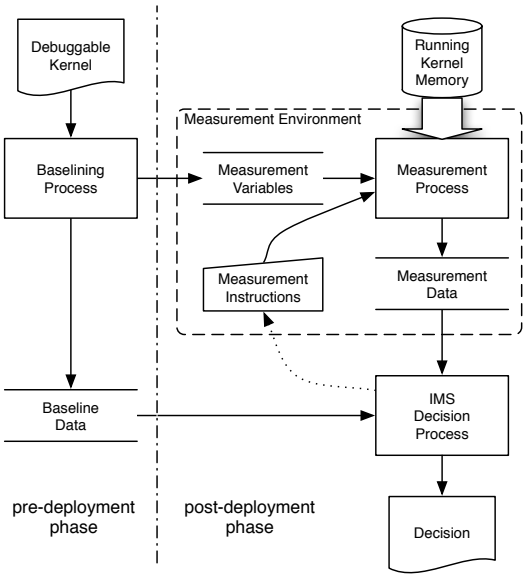

Figure 1: LKIM architecture and data flow

A simplified view of LKIM's architecture is depicted in Figure 1. LKIM has been developed to support two deployment scenarios: Native and Xen. There are significant differences in the measurement inspection mechanisms for each case. In the native scenario, LKIM is a user process within the target space and access to kernel memory is through `/dev/kmem`. On Xen, LKIM runs in a Xen domain distinct from the target's domain. The Xen hypervisor maps the target kernel's memory into the address space of LKIM's domain.[1]

4.1 Contextual Inspection

LKIM measures the Linux kernel using contextual inspection, striving for maximal completeness. This technique attempts to overcome many of the limitations of hash-based measurements, specifically the inability of hash-based measurements to useably identify systems with a large number of expected states and the inflexibility of the results generated by a hash-based system. Inspection uses detailed information about the layout of key data structures to traverse portions of a running process' object graph. This traversal is used to produce a detailed report which describes the structure of the explored subsystems and the current state of identifying variables within those systems.

Contextual inspection is a powerful technique that enables measurement systems to achieve better completeness than would be possible with hashing alone. It produces rich results that can reflect unpredictable structures. However, this richness of detail typically leads to a substantial increase in the size of the results produced, which may be far less usable than a hash-based measurement of a system that could have been effectively measured by either technique. A combination of hashing and contextual inspection allows measurement systems to locate and succinctly identify attributes of targets. The results can represent the structural information gathered by the contextual inspection portion of the system and the concise fingerprints generated by hashing. This combination requires more processing than a single hash of a system with only a few possible states, but results can be analyzed by a challenger in a reasonable period of time.

LKIM combines traditional hash-based measurement with contextual inspection. It uses contextual inspection to provide identifying measurements of the execution path of a running Linux kernel. It not only hashes static regions of the kernel such as its text section, system call table, interrupt descriptor table (IDT), and global descriptor table (GDT), but also traverses relevant portions of the target's object graph and the layout of various kernel structures. Unlike CoPilot, which uses a coprocessor, LKIM can inspect the CPU state and verify the kernel objects inspected are actually being referenced. This traversal produces evidence that indicates the location referenced by function pointers stored in dynamic data structures and the context in which they were detected. This allows a challenger to verify not only that the execution path is entirely within the hashed text section but also to perform sanity checking based on expected groupings of function pointers.

4.2 Instructions and Variables

LKIM breaks up the measurement process into a series of discrete measurements according to a set of *measurement variables*. These variables identify those portions of the target that LKIM can individually inspect. They are arranged hierarchically to enable LKIM to perform increasingly complete measurements of each piece of the kernel that LKIM is able to measure.

LKIM is governed by a set of *measurement instructions* indicating which measurement variables are of interest during a given run. A local configuration file defines the measurement instructions, giving the address and type information of top-level measurement variables. Alternatively, LKIM can receive its measurement instructions directly from an IMS. This greatly enhances the flexibility of the IMS by enabling it to selectively vary the measurement data produced according to the requirements of a particular attestation scenario.

Measurement variables are grouped into *measurement classes*,

[1]Xen was modified to give LKIM's Linux DomU domain access to inspection and control mechanisms normally reserved for Dom0.

each a vertical slice of the measurement variable hierarchy with successive levels providing LKIM with additional contextual information for measuring a particular part of the kernel. Top-level variables are just starting points from which many kernel variables will be examined. To measure a portion of the kernel, LKIM uses the corresponding top-level variable to find the appropriate location in its target's address space. According to the specific technique associated with the variable, LKIM then performs the measurement, recording any relevant properties detected. As prescribed by the measurement instructions, measurement proceeds recursively with increasingly lower levels of the class being inspected until the indicated degree of completeness is attained.

For example, a measurement class for the Linux Virtual File System (VFS) has been defined to include the following top-level measurement variables: *inode_in_use*, *inode_unused*, and *super_blocks*. Each of these variables reference linked lists in the kernel containing the state of inodes dynamically created by the kernel. LKIM is capable of measuring the state kept in each list, including tracing the pointers to VFS operations associated with each inode. LKIM's configuration file might include instructions to measure the VFS class, with the three measurement variables in it being used to select the exact portions of the VFS subsystem to be measured.

LKIM supports other measurement classes to selectively measure Linux. Included are classes for static portions like the kernel text and system call table, as well as dynamic portions like the executable file format handlers, the Linux Security Module (LSM) hooks, and parts of the block IO and networking subsystems. Parts of the kernel can be precisely measured with techniques such as hashing. In others, imprecise heuristics are the best known technique. Because LKIM uses measurement variables to control its operation, different measurement techniques can be assigned to different measurement variables. This enables each portion of the kernel to be measured using the most appropriate technique, yielding the best potential for completeness

Although the total set of measurement variables that LKIM understands does not provide complete coverage of the Linux kernel, LKIM can easily be extended to measure additional portions of the kernel. Where existing measurement techniques are appropriate, new measurement classes and/or variables simply need to be defined and included in measurement instructions. As new or improved techniques are developed and incorporated into LKIM, measurement variables can be redefined to enhance measurement data quality or new variables can be defined to augment the data already collected.

4.3 Baselining

Baselining capabilities were introduced into LKIM to supplement contextual inspection. Baselines are generated to create the structure definitions that indicate how LKIM handles the measurement process for particular measurement variables. Baselines can also be used by an IMS decision process to help validate measurements provided in an attestation. Baselining for LKIM, as shown in Figure 1, exists in two forms: static and extensible.

Static baselining enables LKIM to generate baseline measurements using the on-disk ELF image of the target kernel. LKIM parses standard DWARF debugging information that can be generated at compile time [19, 18], yielding the necessary data to associate regions of memory with particular structure types. LKIM can then decode and measure variables initialized at compile time. Although not all relevant structures can be baselined in this way, many common subversions infect structures such as file or inode operations tables [8] which are typically initialized at compile time.

Static baselining addresses a major problem of runtime measurement systems; performing baseline measurements of a running image may not yield a representation of the true expected configuration. The image may already have been subverted when the baseline is performed. This problem is specifically identified in [13] as a major shortcoming. Because LKIM uses a static baseline that is generated off-line in a safe environment, a system owner can be confident that integrity decisions using the baseline will be made relying on an accurate notion of the expected configuration.

The dynamic nature of target systems makes static baselining insufficient. Extensible baselines solve this problem. When a change in the target is detected, the system can be re-baselined, changing the measurement instructions used by LKIM as necessary. The updated baseline can be propagated to any relevant decision process, optionally allowing it to update its behavior.

4.4 Measuring Kernel Modules

Linux Kernel modules are difficult to accurately measure because they are *relocated* at load time. Hashing is unsuitable for modules because hash values will only be predictable for the original module image and not the relocated version that will execute. Addresses of key data structures cannot be known until relocation. For example, modules are commonly used to support additional filesystem types. Such modules include tables containing pointers to functions that provide filesystem-specific implementations of standard operations like *read* and *write*. Addresses of these functions are unpredictable because they depend on the relocation.

Linux has been modified to notify LKIM whenever modules are loaded or unloaded, making the module's name and address of each section available. On module load events, LKIM uses this information to simulate the loading process on a copy of the module. LKIM extends the current baseline file with data acquired by inspecting the module and adds directives to the measurement instructions to cause the module's data to be remeasured when handling subsequent measurement requests. On module unload events, LKIM reverses the changes.

It is not possible for LKIM's module handling capabilities to achieve complete measurements because there is no mechanism by which LKIM is able to generate a complete and reliable characterization of all modules which are or have been loaded into the kernel. This is not an issue for persistent components of the module such as its primary text section and global data because these sections are located by LKIM and added to the measurement instructions for future measurement requests. However, loadable modules may specify an initialization section which is executed at load time and then released. Such ephemeral module sections may introduce changes to the kernel which would not be connected to the module's main text body or the rest of the kernel's object graph. If measurement is not synchronized to module loading, the initialization section will go unmeasured.

Unfortunately, it is difficult to ascertain exactly which module is being loaded because the cooperation of the measured kernel would be required. Clearly, the kernel's notification could be instrumented to additionally provide a hash of the on-disk image of the module. Careful reasoning must be applied to verify either that the measured kernel cannot be lying and thus the hash must really correspond to the module being loaded, or that the measured kernel can only lie in a way that will be detected by later measurements. An alternate scheme may be to force the kernel to consult a trusted module server or validator before it is able to load a new module. This approach would require a similar argument to be made which ensures that the kernel is unable to surreptitiously bypass the new component when loading modules.

4.5 Remeasurement using LKIM

Repeated measurement helps an IMS achieve freshness of measurement data. Because LKIM supports measurement of a running Linux kernel on demand, remeasurement is simply achieved by running LKIM again. Remeasurement might be necessary as a response to requests from an IMS trying to satisfy the freshness requirements of some attestation scenario. As an example where this might be useful, consider a requirement that measurement data be produced within a certain time period prior to attestation. The IMS can satisfy that scenario by requesting that LKIM produce fresh measurements prior to responding to the attestation request.

LKIM's design also has provisions to attempt to identify conditions which will cause the most recent measurement data collected to no longer reflect the current state of the system, and hence limit the effectiveness of future integrity decisions based on that data. By recognizing such conditions LKIM would be able to anticipate that a remeasurement is necessary prior to being asked by the IMS. LKIM has been designed to respond to external events such as timers indicating that the measurement data is stale and a remeasurement needs to be scheduled. The design also allows for the possibility that the target system be instrumented with *triggers* that will allow a cooperating operating system to notify LKIM that some known event has occurred that will invalidate some or all of the most recent measurement data. Although triggers are useful to reduce response times to requests for measurement data, they are not necessary for correct operation, and LKIM still works when it is not possible to modify the system. To date, the only triggers that have been implemented in LKIM are those that indicate a change in the set of loaded kernel modules. However, the triggering mechanism is present, making it straightforward to add additional triggers as needed.

4.6 Measurement data in LKIM

LKIM was designed for flexibility and usability in the way that data is collected and subsequently reported. It achieves this through its *Measurement Data Template* (MDT). Whenever LKIM runs, collected raw measurement data is stored in the MDT. The MDT has been custom-designed for the target system to enable LKIM to store enough data to meet the maximum possible requirements for completeness. The MDT is formatted to add meaningful structure to measurement data. LKIM stores measurements for different parts of the systems in whatever way is appropriate for the measurement technique being used for that part of the system. If a hash is suitable for one section, the MDT would contain the hash value at the appropriate location. If some section warrants a more complex measurement strategy, the corresponding section of the MDT would contain whatever data was produced. As new measurement strategies are developed making more complete measurements possible, it is a simple matter to extend the definition of the MDT to allow the new form of measurement data to be reflected in the results.

- **Static Memory Regions**
  - **Result: stext [c0102000]:**
    sha1: 34fb210a340248c235c65778f94922620556464c
  + **Result: cpu_gdt_table [c02f0280]:**
  + **Result: sys_call_table [c02bc48c]:**
+ **System Call Table inspection**
- **Virtual File System Inspection**
  + **Result: super_blocks [c02f72c4]:**
  + **Result: inode_in_use [c02f79e4]:**
+ **Block IO Inspection**
- **Binary File Formats**
  - **Result: formats [c0373808]:**
    - **Result: formats[0] [c02f8288]:**
      - **Result: elf_format [c02f8288]:**
        load_elf_binary [c018218c]
        load_elf_library [c0181e1a]
        elf_core_dump [c018398a]
      + **Result: formats[1] [c02f8270]:**
      + **Result: formats[2] [c02f8020]:**
+ **Networking**
+ **SELinux**

Figure 2: Example Measurement Data Template

Figure 2 shows a partial MDT customized for Linux and rendered in HTML. The data are hierarchically arranged by measurement class as prescribed in the measurement instructions, forming a tree from a specified top level variable to the leaf object of concern (e.g., a function pointer). The MDT is stored in XML. It contains hashes of the static regions and detailed information regarding which collections of function pointers are active in the kernel, how many objects reference those collections, and the target address of each function pointer.

LKIM's use of the MDT supports flexibility and usability in the way measurement data can be reported. Since the different portions of the MDT characterize only pieces of the entire system, LKIM is able to support varying degrees of completeness requirements by selectively reporting those portions of the MDT as required by the IMS. It is possible to customize even further by reporting functions of all or part of the MDT. This could be useful in situations where, for example, only a hash of the MDT was deemed necessary by the IMS. The degree of flexibility made possible by the MDT would be very difficult to achieve using a system that only captures a single measurement result for the entire system.

The use of the MDT also supports freshness. When remeasurement is necessary, only those portions of the MDT for which the IMS needs fresher measurements need to be recalculated. This should reduce the impact of remeasurement by not performing wasteful remeasurements on portions of the system.

LKIM's use of an MDT enhances an IMS's ability to meet privacy requirements. When measurement produces data that should not be released in all attestation scenarios, an IMS can dynamically filter the MDT depending on the concerns of the current scenario. Sensitive portions of the measurement data may be sent to trusted third parties so that they may perform the attestations to entities that are not entitled to see the private data. This has the secondary benefit of relieving the burden of integrity decisions at the systems that initiated attestations by allowing specialized systems to be used.

The hierarchical structure of the MDT allows selective reporting of measurement data on any or all of the kernel subsystems. The MDT includes freshness information in the form of time stamps. Depending on the completeness requirements for the current situation, LKIM can select different portions of the data, pruning the tree as required. Remeasurement can be selectively performed on sections as needed. For example, a simple scenario may require only the hash of the kernel text, but from the same MDT, more complex scenarios can also be supported. Along with the kernel text's hash, a report on function pointers might be required so that it can be verified that all function pointers refer to locations in the text section and that are represented in the baseline. As an even more complex scenario, the report might require additional information about function pointer groupings (i.e. pointers stored in the same structure) so that it can be determined that they are similarly represented in the baseline. Using the MDT, LKIM is able to support each of these scenarios without modification.

4.7 Protecting LKIM from Linux

To investigate the feasibility of contextual inspection, LKIM was initially implemented as a Linux kernel module. It executed out of the same address space as the target Linux system, using the kernel to report the measurement results. Although this initial system produced encouraging results with respect to completeness and freshness, there was a noticeable impact to the target kernel. To address this, LKIM was moved into a user-space process, accessing kernel data through the `/dev/kmem` interface. Moving to the richer user-space environment had the additional benefit of enabling LKIM's data reporting to be to enhanced.

Although LKIM could be deployed like this today, it is not recommended. There is no way to protect LKIM from Linux. In fact, Linux must cooperate with LKIM if any measurement data is going to be produced at all, as the LKIM process is totally dependent on Linux for all resources that it requires. The quality of the data collected will always be questionable since Linux would be free to misrepresent true kernel state to LKIM.

To address the protection concerns, LKIM was ported to the Xen Hypervisor [1] and will run on Intel Corporation's Trusted Execution Technology (TXT). The Xen architecture allows functionality to be isolated in virtual machines (VM). LKIM executes in a separate VM from the target Linux system[2] and uses Xen memory mapping functionality to gain the necessary access for measurement. By separating LKIM in this way, LKIM's operation can be protected from Linux, allowing it to perform its measurements and store results without fear of interference from Linux. Using the features of TXT can strengthen the assurance argument and enables a better software monitor to be designed and launched.

This approach succeeds in removing the measurement system from the direct control of the target operating system. However, more is required. With LKIM running in a separate Xen VM, an ability to produce measurements about LKIM and the Xen Hypervisor might be necessary to satisfy completeness requirements. Linking all measurements to a hardware root of trust using a TPM could also be required. An IMS designed to use LKIM running in a VM should address these issues, but specifics about how it might are beyond the scope of this paper.

5. LKIM PERFORMANCE

The contextual inspection approach used by LKIM comes at a cost in terms of impact on the target and complexity for the decision process. However, the gains in flexibility and completeness can justify this expense. Especially if the target is vulnerable to compromises that cannot be detected by hashing. This is the value proposition for LKIM.

The ability to detect rootkits that only infect dynamic data has been demonstrated by LKIM. Detecting modifications to the kernel text area and static data areas can be accomplished with a hash. However, the adore-ng rootkit targets the Linux VFS data structures that are dynamically created in kernel data [8]. It redirects the reference to file system operations on the `/proc` file system to new operations introduced by the module. By traversing the list of active inodes, LKIM reports the existence of the reference to the adore-ng code. A verification check with the baseline of allowable operations then detects its existence. This allows a challenger to detect many redirection attacks by comparing the measurement of a running system to a baseline generated from the static image of an approved kernel.

Performance considerations of an integrity measurement system design include the impact on target performance, the response of the measurement agent to requests for measurement data, and the time it takes the decision maker to process measurement data. The initial analysis of LKIM's performance focuses only on the first two concerns. Since the measurement agent and the target operating system share computing resources, reducing the impact on the target may come at the cost of a longer response to measurement requests and vice versa. This assumes that the number of data structures inspected is the same, however the workload of the target operating system determines the number of data structures LKIM inspects.

Testing was performed using a standard desktop platform and two simulated workloads. The same hardware platform[3] was used for both Xen and Native configurations.[4] Resource contention between LKIM and the target workload is managed by a scheduler and consequently the measurement duration is determined by priority given to the measurement agent; default scheduling algorithms were used in both configurations. Target kernel workloads were simulated by the Webstone benchmark and a build of the Linux kernel. Webstone [11] performs mostly I/O processing and the number of

[2]Xen's planned security and light-weight VM capabilities will allow LKIM to be truly isolated in its own VM without Dom0 privileges.

[3]Hardware: Dell Optiplex GX400 w/1GB of RAM.

[4]Xen config: two 2.6.16.13-xenU kernels each w/256M RAM. Native config: a standard Linux 2.6.16.13 kernel w/256M RAM.

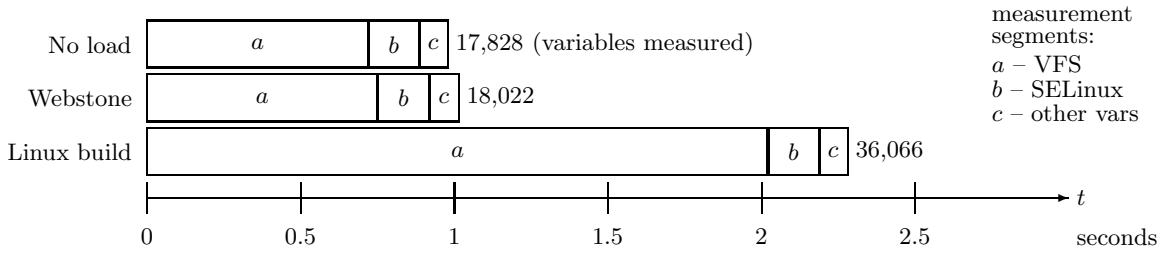

Figure 3: LKIM processing timeline

measurement variables is a function of the number of clients hitting the web server. The kernel build workload provides a combination of I/O and CPU utilization and creates a large number of variables for LKIM to measure. In all cases, the set of measurement instructions included the full set of classes described in section 4.2.

Figure 3 shows the processing timeline for LKIM under each workload configuration. Using the Xen control mechanisms, LKIM is able to suspend the target during measurement; the timeline shown represents LKIM processing during peak activity for each workload. Without any workload on the target kernel, LKIM takes just under 1 second to inspect nearly 18,000 kernel variables. Inspection of VFS accounts for the majority of this time, with SELinux and other variable inspections taking approximately 260ms. Under Webstone, the number of variables increases only slightly, but with Linux build the number increases to just over 36,000 variables. In each case, the increase in variables is due to an increase in dynamically created data structures within the VFS.

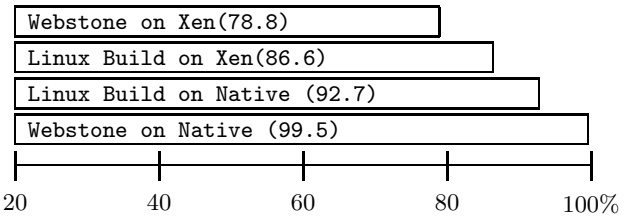

Figure 4: Target performance relative to Native platform without LKIM. Measurements are repeated every 2 minutes.

The impact of measurement on the target kernel can be regulated by adjusting the measurement frequency. Figure 4 shows how target performance is affected by LKIM processing with the measurement interval fixed at 2 minutes. For each workload, the relative performance is shown for both Native and Xen configurations.

The performance results show where improvements in efficiency would make the best gains for the Xen architecture. The biggest improvement would be to reduce the number of variables measured. Currently, LKIM assumes all objects need to be inspected for each measurement run. A better approach would be to recognized which objects have been modified and only measure those. Xen provides a way to detect which pages have been dirtied by the target but the largest set of objects, the VFS nodes, are in a linked list. A more sophisticated algorithm would be needed to locate only the entries that have changed.

6. FUTURE DIRECTIONS

LKIM was specifically designed to expand the bounds of the completeness, freshness and flexibility properties. Ongoing work is continuing to push contextual inspection of Linux to enable measurements that most accurately reflect the state of Linux at the time of measurement. This work focuses on analyzing unmeasured portions of the dynamic data segment for places where unauthorized modifications could affect the system's execution flow and then experimenting with different ways to characterize that data in ways useful for an IMS. For example, the kernel data structures representing SELinux policy and its access vector cache [9] are critical pieces that must be reflected in any measurement data. Although it is unlikely that an OS may ever be completely measured, the goal is to make it increasingly difficult to have modified systems go undetected. Ways in which the complimentary techniques of contextual inspection and semantic measurement could be combined to further this goal should also be investigated.

Work to push the bounds of measurement data freshness is underway. Linux is being instrumented with better triggering mechanisms that will inform LKIM of more events that should initiate remeasurement. While these triggers enable LKIM to update tainted portions of the measurement data in anticipation of future requests, there are admittedly some potential vulnerabilities associated with allowing the target to dictate when remeasurements occur. This capability could potentially be used to manipulate the measurement data. It remains as future work to study the trust relationships between LKIM and Linux to better understand under what circumstances it is safe to use the triggering mechanism.

Performance improvements to LKIM are also under way. In addition to making measurement more efficient and complete, it is also necessary to ensure that measurement results are coherent. Strategies are being explored to ensure that Linux is in a consistent state at the time measurement is started and for the complete duration of the measurement process. A two-pronged strategy might prove best to ensure coherency: detect a quiescent target and prevent future changes to the target from tainting a measurement in progress. The use of Xen's facility for tracking dirty pages as a means of isolating changes is being explored. Copy-on-write techniques will be employed to guarantee that no change to kernel data made during the measurement process is reflected in the result. This should produce a much more coherent result at only a slight cost to freshness.

The LKIM prototype was implemented to measure those portions of the Linux kernel that impact kernel operation. Although many kernel structures, including process address spaces, mount points, network interfaces and others, may

be critical to measurement in some attestation scenarios, LKIM does not currently measure them since they do not directly affect kernel execution. Investigation is required to determine the best way to produce these measurements. It would be very straight forward to extend LKIM using the techniques of contextual measurement, but it may be more appropriate to design a kernel service similar in concept to what was done in IMA and to use LKIM to measure it. This seems to be a logical way to break the problem, allowing LKIM to measure the kernel and the kernel to measure user-space visible kernel abstractions. Experimentation is planned to better understand how LKIM interacts with user space.

Work is necessary to understand the trust issues related to LKIM and the larger IMS system in which it is incorporated. LKIM resides in a separate Xen virtual machine to protect it from the Target Linux system. What are the requirements of LKIM's virtual machine and how might attestations about it get measurement data? How is the underlying Xen system measured and how is that measurement data used in attestations about Linux? How might a system using LKIM take advantage of TPM hardware? What are the interdependencies of these components and what is necessary to boot the different components into a consistent and measured state? Ongoing research is underway to help understand these issues.

7. CONCLUSION

In its current prototype form, LKIM stands as an excellent example that it is possible to produce complete, fresh, and usable measurements of a running modern operating system. Running LKIM in a protected separate Xen virtual machine from the target also demonstrates a method for protecting the measurement system from the target. The measurement results that LKIM produces, including its ability to detect unknown rootkit insertions, are encouraging and certainly warrant further development and experimentation. LKIM serves as a building block to improve measurement and attestation techniques. The work underway to extend LKIM's coverage of Linux, to implement better remeasurement triggers, and integrate LKIM into a larger IMS should form a good basis for experimentation into how to solve some of the problems such as measurement coherency and ultimately what is truly desired out of a total measurement and attestation system and the attestation scenarios it will support.

8. REFERENCES

[1] P. Barham, B. Dragovic, et al. Xen and the art of virtualization. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[2] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation - a virtual machine directed approach to trusted computing. *Proceedings of the 3rd USENIX Virtual Machine Research & Technology Symposium*, May 2004.

[3] D. Heine and Y. Kouskoulas. N-force daemon prototype technical description. Technical Report VS-03-021, The Johns Hopkins University Applied Physics Laboratory, July 2003.

[4] P. Iglio. Trustedbox: a kernel-level integrity checker. In *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*, page 34. IEEE Computer Society, 1999.

[5] Intel Corporation. *IA-32 Intel Architecture Software Develper's Manual*, 2004.

[6] T. Jaeger, R. Sailer, and U. Shankar. Prima: Policy-reduced integrity measurement architecture. *SACMAT '06: Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, 2006.

[7] G. Kim and E. Spafford. *The Design and Implementation of Tripwire: A File System Integrity Checker*. Purdue Univiversity, November 1993.

[8] J. Levine, J. Grizzard, and H. Owen. Detecting and categorizing kernel-level rootkits to aid future detection. *IEEE Security and Privacy*, 2006.

[9] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. *Proceedings of the FREENIX Track*, June 2001.

[10] P. Loscocco, P. Wilson, et al. Measuring the linux kernel using contextual measurement. Technical Report AI-07-077, The Johns Hopkins University Applied Physics Laboratory, August 2007.

[11] Mindcraft, Inc., `http://www.mindcraft.com`. *WebStone 2.x Benchmark Description*.

[12] G. Mohay and J. Zellers. Kernel and shell based applications integrity assurance. In *ACSAC '97: Proceedings of the 13th Annual Computer Security Applications Conference*, page 34. IEEE Computer Society, 1997.

[13] N. Petroni Jr., T. Fraser, et al. Copilot - a coprocessor-based kernel runtime integrity monitor. *Proceedings of the 13th Usenix Security Symposium*, pages 179–194, August 2004.

[14] N. Petroni Jr., T. Fraser, et al. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. *Security '06: 15th USENIX Security Symposium*, 2006.

[15] R. Sailer, X. Zhang, et al. Design and implementation of a TCG-based integrity measurement architecture. *Proceedings of the 13th Usenix Security Symposium*, pages 223–238, August 2004.

[16] A. Seshardri, M. Luk, et al. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. *ACM Symposium on Operating Systems Principles*, October 2005.

[17] J. Sheehy, G. Coker, et al. Attestation evidence and trust. Technical Report 07_0186, MITRE Corporation, March 2007.

[18] Tool Interface Standards Committee. *DWARF Debugging Information Format Specification v2.0*, May 1995.

[19] Tool Interface Standards Committee. *Executable and Linking Format (ELF)*, v1.2 edition, May 1995.

[20] Trusted Computing Group, `https://www.trustedcomputinggroup.org`. *TCG Specification Architecture Overview – Specification Revision 1.2*, April 2004.