LKIM: The Linux Kernel Integrity Measurer

J. Aaron Pendergrass and Kathleen N. McGill

The Linux Kernel Integrity Measurer (LKIM) is a next-generation technology for the detection of malicious modifications to a running piece of software. Unlike traditional antivirus systems, LKIM does not rely on a database of known malware signatures; instead, LKIM uses a precise model of expected program behavior to verify the consistency of critical data structures at runtime. APL and the Research Directorate of the National Security Agency (NSA) developed the LKIM prototype and are now working to transition the technology to a variety of critical government applications.

INTRODUCTION

Despite the techniques available to increase confidence that software is correctly implemented and free from exploitable vulnerabilities, running software is still vulnerable to a variety of threats. Many of these threats remain relevant to executing software because of inherent limitations of the static techniques, whereas others arise from a lack of a trust in the environment in which software is run. For example, even an application that has been proved to be free of exploitable vulnerabilities may be subverted by a malicious operating system or hardware component. For this reason, solutions that provide confidence in the correct execution of software are an important component in the design of trustworthy systems. Dynamic integrity measurement is a technique we developed in collaboration with the Research Directorate of the National Security Agency (NSA) for periodically establishing confidence that a piece of executing software is behaving consistently with its static definition. Although dynamic integrity measurement cannot guarantee that software is trustworthy in the sense of not being exploitable, it is able to establish that any assurance gained by static analysis is maintained by the executing software.

The Linux Kernel Integrity Measurer (LKIM) is an implementation of a dynamic measurement technique targeting the Linux operating system kernel. Unlike most other systems for malware detection, dynamic integrity measurement systems (IMSs) such as LKIM do not rely on a database of known malware signatures. This means that LKIM is able to detect previously unknown "zero-day" malware. Although LKIM was originally developed to verify the integrity of the Linux kernel, researchers in the Asymmetric Operations Department of APL have reconfigured LKIM to target

other operating systems as well as application-level software. Recently, work on LKIM has shifted from prototyping activities to real-world deployments, including use on APL's enterprise network and a variety of critical government applications.

Traditionally, the arms race against cyber adversaries has been reactive. As new attacks and malicious software are discovered "in the wild," defensive tools are enhanced to defend against these threats. Often, by the time these enhancements are deployed, the attackers have moved on to new techniques that are once more invisible to the defender. Antivirus software is probably the best-known tool used to combat malicious software. Most antivirus software relies on a set of signatures or "virus definitions" that precisely identify known malware. These definitions must be updated frequently to protect against the latest known threats. Because of its dependence on known signatures, antivirus software is fundamentally unable to defend against novel threats.

Dynamic integrity measurement follows a different approach: rather than attempt to recognize malicious software signatures, it characterizes how legitimate software is expected to behave and identifies any deviation as malicious. This idea is not fundamentally new; "anomaly detection"-based systems have frequently tried to model legitimate system behavior to flag anything out of the ordinary as malicious. LKIM differs from these systems in that it does not rely on statistical modeling; instead, it relies on the fact that software behaves in very predictable ways according to its source code. The key insight behind LKIM is that malware often changes the kernel's state in a way that is inconsistent with the kernel's source code. By detecting these inconsistencies, LKIM is able to detect previously unknown attacks.

We developed LKIM to identify modifications to a running Linux kernel because malware capable of making such modifications to the kernel, known as "kernel-level rootkits," has complete control over the behavior of all software running on a system. Further, because most detection software runs as a process controlled by the kernel itself, these rootkits are notoriously difficult to reliably detect. LKIM uses virtualization and a technique called virtual-machine (VM) introspection to analyze the state of the kernel when running outside of that kernel's direct control. Kernel-level rootkits for other operating systems, such as Windows or Mac OS, present a similar security threat. Although LKIM was not initially designed to target these systems, the same techniques can be used to verify their integrity.

MEASUREMENT PROPERTIES

LKIM is designed to serve as the *measurement agent* (MA) within an IMS capable of supporting a range of different *decision makers* (DMs). Within an IMS, the MA is responsible for collecting evidence describing the state of a piece of software (referred to as a *target*). This evidence is presented to the DM in a process called *attestation* that supports the trustworthy evaluation of the target's state. The DM is responsible for evaluating the presented evidence to determine whether it was collected by a valid mechanism and represents a valid state of the target. In general, an IMS should provide the following properties to support a meaningful evaluation of the target's state:[1]

- **Complete results:** An MA should be capable of producing sufficient measurement data for the DM to determine whether the target is in an expected state for all attestation scenarios supported by the IMS.
- **Fresh results:** An MA should be capable of producing measurement data that represent the target's state recently enough that the DM considers the represented state sufficiently close to the target's current state.
- **Flexible results:** An MA should be capable of producing measurement data with adaptability to fulfill the requirements of all attestation scenarios supported by the IMS.
- **Usable results:** An MA should be capable of producing measurement data in a format that the DM can easily evaluate to determine whether the represented state is expected.
- **Protection from the target:** An MA should be protected from the target so the target cannot corrupt the measurement process or measurement data without the DM's detection.
- **Minimal impact on the target:** An MA should not require modifications to the target, and execution of the MA should not diminish the target's performance.

In many cases, these properties represent trade-offs between the assurance provided by the measurement system and the impact the measurement system may have on the target. For example, LKIM can be configured to perform very extensive measurements; however, because LKIM is competing for computational resources with the running client, the processing time required to compute these measurements may be an unreasonable imposition on the target.

LKIM can be run on demand and always produces results reflecting the state of the target at the time it is run; thus, its results are potentially very fresh. This is in stark contrast to static or load-time IMSs, which can only provide evidence of the state of the target at the time it was loaded into memory. However, running LKIM frequently may cause unacceptable performance degradation, so caching results for some time may be

advantageous. The exact frequency with which to run LKIM is still an open research question. Because LKIM's results represent only a moment in time, a long time period between measurements may allow an adversary to break into a system, accomplish his mission, and restore the kernel's integrity without causing a failed measurement. In some sense, any window is too long because some adversary missions, such as stealing cryptographic keys, may be accomplished in microseconds. A recommended practice is to perform a fresh LKIM measurement as part of an access control decision such as network access control. This scheme allows the access control policy to make its decision on the basis of fresh evidence, without unduly burdening the target.

Integrity measurement data may be as complex as a complete image of the target's memory state or as simple as a single bit indicating whether the target has integrity. These extremes capture the trade-off space between the flexibility and the usability of measurement results. Although a complete memory image may be able to satisfy a wide range of DM policies, it is exceedingly difficult to process. Similarly, a single bit is trivial to process but offers no flexibility. LKIM strikes a balance between these concerns by abstracting the target's state into a graph of data values and the interconnections between these data. The abstraction allows DMs to easily process LKIM's results, whereas the maintenance of structural information allows for flexibility in the policies they may evaluate. Notably, if a new class of attacks is discovered that modifies the parts of the target's state that are already measured but were not detected by existing DM policies, archival measurements could be evaluated against a new policy to determine when the attack was first present in a set of targets.

Protection of the MA from the target is vital to establishing trust in the measurements. LKIM uses virtualization to collect measurements of a target VM's kernel from a separate measurement VM. Virtualization technology introduces a layer of software known as a VM manager or hypervisor below the standard operating system kernel and allows multiple operating systems to run simultaneously in VMs on the same physical hardware. This places trust in the VM manager that it fully isolates the measurement VM from any attacks that may be launched from the target VM and also that it provides an accurate view of the target VM's state. To minimize the trust, we have researched other features of the PC architecture that would allow LKIM to run outside the control of any operating system or VM manager. Because many current and legacy systems do not use virtualization, LKIM can also be run in a hosted mode to measure the kernel on which it is running. This mode offers none of the trust benefits provided by the virtualized solution; a clever rootkit could modify the interfaces that the hosted LKIM uses to inspect the kernel to hide itself. However, this mode is easier for deployment on legacy systems and does make it somewhat more difficult for rootkits to hide.

It is impossible to develop a measurement system with no impact on the target. Any measurement engine running on the same hardware as the target will have to compete for the finite computational resources available, such as processor time or memory. We have made efforts to minimize the impact LKIM poses on the target both by optimizing LKIM's code to reduce its use of these resources and by leveraging architectural features such as VM snapshotting to avoid activities such as pausing the target for long periods of time. Beyond these performance impacts, a measurement system may impact the development or deployment of updated targets. LKIM requires a precise description of the data structures used by the software it is measuring. This means that legitimate updates to the target may cause LKIM to become confused and generate false alarms. We partially address this problem by separating the target-dependent aspects of LKIM into a configuration file that should be deployed in sync with updates to the target software. This solution imposes some management cost to deploying LKIM; we are working in pilot deployments to better understand how high this cost is and how it can be best addressed (see the *LKIM Transition* section for more detail).

HOW LKIM WORKS

Although LKIM provides a generic measurement capability, the majority of work on applying LKIM has focused on the Linux kernel itself. This section provides an overview of LKIM's measurement algorithm using the configuration developed for the Linux kernel as an example. Efforts to retarget LKIM to measure other software, including other operating system kernels and application-level software, have followed the same process with minor changes to account for differences in file formats and software architecture.

LKIM divides the task of verifying the Linux kernel's integrity into three distinct phases:

- **Static baselining:** the identification of valid program states
- **Measurement:** the collection of evidence
- **Appraisal:** the evaluation of the collected evidence against the baseline

Figure 1 indicates how these three phases work together to determine the integrity of the system. The baselining phase combines an expert understanding of the Linux kernel's behavior with information found in the kernel's executable file. The measurement phase inspects the state of a running instance of the kernel to summarize those aspects of its state relevant to the integrity decision; this summary constitutes a "measurement" of the running kernel's state. The appraisal phase

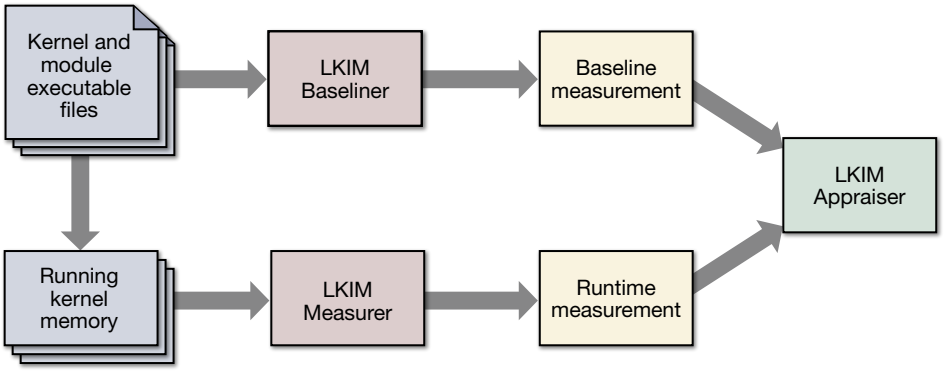

Figure 1. The LKIM system consists of three components: the *baseliner*, which analyzes the kernel and module executable files to produce a ground-truth *baseline measurement*; the *measurer*, which analyzes the runtime memory of the kernel to produce a *runtime measurement*; and the *appraiser*, which compares the runtime measurement against the ground truth.

consumes both the baseline data and the measurement to determine whether the values observed during the measurement phase are in the set of values identified as valid during the baseline. If all values found in the measurement are consistent with the baseline, the kernel is judged to be operating correctly; if not, LKIM assumes that the deviation is the result of a malicious modification and raises an alarm.

Static Baselining

The goal of the static baseline phase is to determine what a running instance of the software should look like in memory. LKIM accomplishes this by inspecting the kernel's executable file and extracting information about the code and data structures in the image. This information is included in the debug information of Executable and Linkable Format (ELF) binary files. In this phase, we use expertise about both the ELF standard and the Linux source to locate the appropriate information within the kernel image and to transform that information into an accurate representation of the image when it is loaded into a machine's memory. The process generates ground truth against which a runtime measurement can be compared. A key advantage of this strategy is that we are constructing a valid software state directly from the on-disk binary image. An approach used by other systems is to use a measurement taken at a time when the target is believed to be in a good state as the baseline. Such systems are vulnerable to a class of attacks in which the target is modified in memory before the baseline measurement is taken. Because LKIM's baselines rely only on the on-disk executable file, LKIM is immune to such attacks.

The baselining process begins by simulating the kernel's load procedure. This simulation is based on how a bootloader such as the Grand Unified Bootloader would read the kernel from the hard disk at system boot time and position its various sections in memory as well as some of the initialization procedures of the Linux kernel, such as optimization of its code segments for the hardware. The load procedure also includes simulating the load process for any loadable kernel modules present in the measured system. Simulating module loading is similar to the simulation of the core kernel code but relies on some dynamic details that can only be determined by the client. Specifically, the Linux kernel loads modules by allocating a buffer containing the module code and data and then performing a process called relocation to adjust the module on the basis of the address of the buffer. To simulate a module load, our baseline process must know this address value. Fortunately, we can use the address values reported by the measured kernel without reducing our trust in the outcome. If the kernel misrepresents the set of loaded modules, it will cause the baseline process to either omit certain loaded modules or to load them in a way that is inconsistent with the kernel's true behavior. In either case, comparison of the generated baseline with the measurement data will fail because references in the measurement to the actual module load location will not match the data found in the baseline. Because of the difficulty of handling modules, our baseline process can be split into two phases: one to generate a baseline for the core kernel and a second to extend that baseline with data on the set of loaded modules.

The most significant challenge in the static baseline process is the accurate simulation of code transformations that occur when the kernel's binary image is loaded into memory. The Linux kernel includes a variety of "self-modifying" code features that provide enhanced functionality for the kernel software while it is running:

- **Alternative instructions:** instruction optimizations for specific processors
- **Paravirtualized operations:** optimizations for guest VM execution
- **Kernel function tracers:** debugging and profiling facilities
- **Symmetric multiprocessing locks:** software support for multiprocessors

These features may enhance functionality, but they are particularly challenging for integrity measurement. The transformation of self-modifying code is closely tied to the software and hardware configuration of a system. Any change in kernel source, kernel configuration, or platform hardware can alter the process. LKIM requires information about the target system's software and hardware configuration to apply this information in constructing the baseline. LKIM currently implements multiple versions of each transformation function corresponding to the implementations found in different versions of the kernel. LKIM selects which version to apply on the basis of identifying features of the kernel being baselined. Mechanisms that are better able to automatically identify and simulate new transformations are an active area of research.

Once this simulation is complete, the baseline process first records a cryptographic hash of the kernel and loaded module's executable code sections and then traverses the debugging information packaged with the kernel and its modules to locate any instances of structures that should appear in the baseline. To support a variety of devices and file systems, the Linux kernel has internal programming interfaces that driver authors implement to connect the details of their device to the generic functionality of the kernel. These interfaces are exposed as structure types containing pointers to the functions implementing device-specific behavior. Instances of these structures are typically defined as constant global data structures and passed by reference to the generic kernel code. Our baseline process is tuned, on the basis of expert human analysis, to locate instances of these structures using the debug information and record their values. This analysis must be periodically updated to reflect new kernel interfaces or major changes in existing interfaces. Fortunately, more of the structures measured by LKIM are central to kernel operations and thus tend to be stable across versions.

The hash of the executable code sections combined with the values recorded for instances of key structures comprise the generated baseline. The measurement phase collects similar data representing the state of the executable code and data structure instances present in memory at the time of measurement. During appraisal, these two sets of data are compared to determine whether the executable code at runtime is identical to that found in the baseline and whether the recorded values in the measurement are consistent with those in the baseline.

Measurement

The primary challenge of the measurement phase is determining which aspects of the kernel's state are important to the integrity decision and producing a structured representation of these data. Later, during the appraisal phase, these data will be compared against the baseline to determine whether the kernel has been modified. LKIM decodes the memory in use by the kernel as a set of data values with relationships describing which data are reachable from other data. The full set of active data and their connections is known as the "object graph" of the kernel; LKIM actually computes an abstraction of this graph, eliding any information that is not relevant to the integrity decision of the appraisal phase.

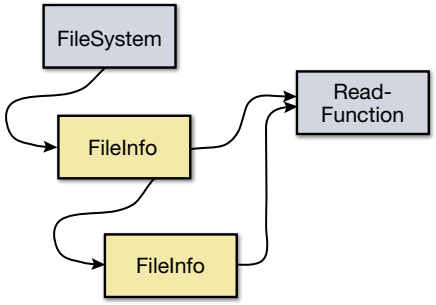

Figure 2. A graph representing a subset of the data structures active at the time of measurement. LKIM records each identified data structure and the linkages between them. During appraisal, this graph may be evaluated against the constraint that all FileInfo structures reference the correct ReadFunction for the FileSystem.

Figure 2 shows an example of such a graph; each box represents some data found by LKIM, and the arrows connecting the boxes show how these data are connected in the kernel's memory. LKIM performs this decoding on the basis of a set of rules describing the expected layout of data structures in the kernel's memory. Table 1 lists examples of the rules that may be used to generate the graph from Fig. 2. Rule 1 tells LKIM that the FileSystem (the Linux kernel actually calls this structure a super_block) data structure can be found at the memory address 0x1234 (expressed in hexadecimal). This structure is used by the kernel to store data about the system's hard drive. In particular, the FileSystem structure maintains a list containing metadata for each file stored on

Table 1. The set of rules that might have given rise to the graph shown in Fig. 2

Rule	From	To	Memory Offset
1		FileSystem	0x1234
2	FileSystem	FileInfo	4
3	FileInfo	FileInfo	4
		ReadFunction	8

Rule 1 says that a FileSystem structure can be found at memory address 0x1234 (expressed in hexadecimal). Rule 2 says that once a FileSystem has been found, a FileInfo structure can be found by following a pointer at offset 4 from the FileSystem. Rule 3 says that once a FileInfo is found, another FileInfo can be found by following a pointer at offset 4 and that a ReadFunction can be found by following a pointer at offset 8.

the hard drive. The second rule tells LKIM how, given the file system it located using the first rule, it can find the first FileInfo (this structure is actually called an inode or "index node" in the Linux kernel) structure in this list by examining the value at offset 4 from the base of the FileSystem. The FileInfo structure provides information for a file on the hard drive, such as who created it, when it was last modified, and who may access it, and a reference to the code that the kernel should use to read the file from the drive. The final rule tells LKIM how to use the data in one FileInfo to find the next FileInfo and a ReadFunction (this is actually a whole table of functions called inode_operations, including open, read, write, close, and other standard operations on files) by examining the data at offset 4 and offset 8 from the original FileInfo. The last two rules show how LKIM uses the information identified in one step to find more information. LKIM will repeatedly attempt to apply these rules until no new data remain to be discovered. This allows LKIM to traverse the kernel's object graph and record critical information for each datum, or node, it visits. We have developed automated tools to help craft these rules for different kernels. Unfortunately, the determination of which data are important to kernel integrity requires an understanding of how the kernel operates, so tuning these rules is still a largely manual process.

The LKIM rules define the measurement variables of interest and how to measure them. These variables include data structures as well as memory segments, such as the kernel text section. The measurement begins with a short list of top-level variables. For each variable, LKIM measures data to ascribe value to the variable and/or locate other variables to measure.

The complete list of variables that are measured depends on the target platform, but a typical LKIM measurement includes the following:

- Kernel and kernel module code
- Architectural entry points such as the Interrupt Descriptor Table
- System call table
- Task structures that store information on Linux processes
- Jump tables containing pointers to executable code, such as the ReadFunction in the example depicted in Fig. 2.

Table 2. Common appraisal constraints, and their associated measurement data, used by LKIM

Constraint	Constrained variables
All code sections in the measurement have a matching code section in the baseline.	Kernel core and kernel module code sections
Data structures in the measurement have a matching data structure in the baseline.	Jump tables (super_operations, inode_operations, etc.) and the system call table
Function pointers point to valid code sections in the baseline.	Miscellaneous function pointers, including dynamically allocated function pointers
All tasks, or processes, in the system's run queues and wait queues descend from the init task.	Task structures

Appraisal

The appraisal phase consumes these data and compares them with data computed from the on-disk representation of the kernel to determine whether the observed state is consistent with the original file. This determination relies on an expert understanding of how the kernel operates. Ultimately, LKIM provides a result indicating either that no modifications were detected or exactly which data in the kernel have been modified in an unexpected way.

LKIM's appraisal phase is specified as a series of logical predicates that are evaluated over both the baseline and measurement graphs. These predicates can refer to the graph node representing data at a particular address in memory, all nodes representing data of a particular type, or the relationships among multiple graph nodes. In the example given in Fig. 2, an example constraint is that "all read function nodes that descend from the given file system node must point to the correct executable code for reading a file from that FileSystem."

During LKIM appraisal, we apply a collection of constraints that we have defined for a healthy Linux kernel. Each constraint is applied to one or more measurement variables to detect unexpected modifications. Table 2 displays the typical constraints in an LKIM appraisal and the variables that they constrain.

IMPACT OF LKIM

In most cases, LKIM detects the impacts of a malicious attack rather than the attack itself. Generally, an attack will use some vulnerability in the system to gain privileges with the goal of establishing persistence on the system. To persist in the kernel, the attack must be able to inject an implant or modify the image in memory. LKIM detects precisely these implants and modifications.

There are two kinds of attacks that LKIM targets in particular: kernel code injection (or modification) attacks and kernel control flow attacks. A code injection attack attempts to introduce new code into the kernel or modify existing code to alter the intended execution of the system. LKIM detects these attacks by checking all code in the executing kernel against the static baseline. Any code that does not have a match in the baseline will be detected as corrupt.

Control flow attacks attempt to modify data structures, such as the system call table and function pointers, which direct the execution flow in the kernel. These modifications may be used to redirect execution to the attacker's code for some malicious means or to the wrong code in the kernel to cripple an intended service. LKIM detects control flow attacks by verifying that the system call table and function pointers have a corresponding match in the baseline and point to valid code.

A key advantage that LKIM has over static integrity measurement is its ability to measure and appraise dynamically allocated data structures. As the kernel is executing, it services requests from various system components. These service requests may require generation of new instances of data structures and function pointers. These new structures do not exist in the binary image of the kernel, so static integrity measurement does not measure them. The LKIM measurement algorithm uses runtime information to walk the kernel object graph. LKIM discovers these dynamically allocated structures during measurement and includes them in the measurement data for appraisal.

LKIM TRANSITION

LKIM has evolved from an early proof of concept to a full research prototype. We are committed to transitioning LKIM from a research prototype to a deployed system with a valuable operational impact.

The LKIM prototype is a collection of tools that perform static baseline measurement, runtime measurement, and appraisal of target systems. LKIM requires a management framework to integrate it with existing enterprise or tactical networks. This framework provides measurement initiation and scheduling; communication between target systems and the appraisal server; data management for the required LKIM inputs and storage of measurements, baselines, and appraisal results; and an interface with an alert management and response system. We believe it is ideal to integrate LKIM within existing management tools in targeted deployment networks to smooth integration and adaptation for end users.

We started the LKIM transition with a pilot deployment on the APL enterprise network. We continue to explore broader deployment in a variety of government applications. Each new deployment presents a unique set of challenges to LKIM, and we hope to ease the integration of LKIM as we gain operational experience.

We have identified several characteristics of the target environment that impact LKIM integration. These characteristics drive design decisions and preventative maintenance for LKIM deployment in the field:

- **Stability:** Changes in the target environment require updates to LKIM tools and the management framework. Typically, each new software or hardware configuration requires a reconfiguration of the LKIM tools. Managing LKIM tends to be much easier in carefully administered environments with homogenous configurations and infrequent updates. In any environment, updates to LKIM must be managed in conjunction with updates to the measured mission software.

- **Diversity:** All levels of diversity within the target environment affect integrity measurement. In particular, target diversity requires robustness of the LKIM static baseline process and leaves little room for errors in the deployed prototype implementation. Testing and evaluation of LKIM should be commensurate with the anticipated diversity of the deployment environment.

- **Scale:** As the number of target machines increases, the management framework utilization increases. Demands on the appraisal server(s) for target communications and appraisals may increase drastically. In addition, the efficient management of the database of the inputs and integrity measurement results becomes nontrivial. The management framework should be sized appropriately to accommodate the projected scale of deployment for the term of service.

- **Resource constraints:** The resource constraints of the target environment may have great impact on the integration of LKIM. In cases in which the target machines are user desktops, a typical 3-s LKIM measurement is barely noticeable. However, as we explore mission-critical targets, which experience stricter resource constraints, we may need to optimize LKIM and the management framework for the mission.

CONCLUSIONS AND FUTURE WORK

LKIM is a dynamic MA capable of providing detailed evidence of the operating state of a running piece of software. The techniques implemented by LKIM enable the detection of a wide range of long-lived in-memory software modifications and in particular target kernel-level rootkits that pose a significant threat to the trustworthiness of a computing system. Because LKIM does not rely on signatures of known malware, it is able to

detect zero-day infections, making it ideal for countering the "advanced persistent threats" of concern to many of APL's sponsors. Together with the Research Directorate of the NSA, APL has developed LKIM from a concept to a prototype solution and is now working toward deploying LKIM in high-impact environments for our broader sponsor base.

The major challenges to broad deployment of LKIM are automating the analysis of which data structures must be recorded in the baseline and how structures should be measured and appraised. This currently relies heavily on an expert understanding of the system being measured. Automating this process would enable the LKIM engine to be quickly retargeted at new versions of the Linux kernel as well as at totally new software targets, such as other operating systems or application software. Work on automated software invariant generation, as well as next-generation "measurement-oriented programming languages," may enable software developers to integrate the development of a MA such as LKIM into their existing process for developing mission-oriented software.

REFERENCE

[1]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*, New York, NY, pp. 21–29 (2007).

The Authors

J. Aaron Pendergrass is a member of the Senior Professional Staff in the Cyberspace Technologies Branch of APL's Asymmetric Operations Department. His work focuses on enhancing software assurance via formal methods for mechanized program analysis, automating reverse engineering, and dynamic software integrity measurement. **Kathleen N. McGill** is a member of the Senior Professional Staff in APL's Asymmetric Operations Department. She represents APL in the Trusted Computing Group's Mobile Platform Working Group and acts as a liaison between the greater trusted mobile computing community and researchers at APL. Her work at APL focuses on dynamic integrity measurement of desktop, server, and mobile devices to ensure the integrity of target software. For further information on the work reported here, contact J. Aaron Pendergrass. His e-mail address is aaron.pendergrass@jhuapl.edu.

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