SPEE: A Secure Program Execution Environment Tool Using Static and Dynamic Code Verification

Olga Gelbart, Bhagirath Narahari, and Rahul Simha
The George Washington University
Washington, DC
{rosa, narahari, simha}@gwu.edu

Abstract— **With the growing number of successful computer attacks, especially those using the Internet and exploiting the vulnerabilities in the software and applications, software protection has become an important issue in computer security. This paper proposes a system – SPEE – for software integrity protection and authentication and presents performance results. Our system architecture utilizes key components from the compiler process and operating system features, and provides static verification before execution and code block verification at runtime thereby providing a secure program execution environment. Integrity checking is performed by means of a hashing scheme, which not only detects changes but isolates these changes in O(log N) time, where N is the number of code blocks in the executable. The SPEE tool is designed to function as part of the operating system kernel and provides a trusted computing system.**

I. Introduction

With the ubiquitous use of the Internet in the workplace and at home, it is becoming increasingly important for a computer system to function reliably and securely. Most attacks on computer systems are carried out through the use of networks, particularly the Internet, so it is becoming more and more necessary not only to verify a program's correctness and integrity (as well as authenticate it) before execution starts, but also during runtime. Ideally, we would like to run a trusted networked system, where all programs are only run by those authorized to do so and are not vulnerable to attacks either before or during execution.

With the growing number of successful computer attacks based on software vulnerabilities, software protection has become an important issue in computer security [6]. Attackers exploit software vulnerabilities caused by programming errors, system or programming language flaws. Code is often attacked at runtime to gain unauthorized access to the computer system. A number of software protection methods have been proposed to prevent or detect these kinds of attacks [15], [8], [4], [1], [13]. There currently exist a number of open-source projects focusing on specific areas of software security [6]. These include static code analysis tools, dynamic tools that insert runtime checks (such as buffer overflow protection) and various operating systems controls. While they secure the system against specific types of attacks, the current methods do not provide code integrity and authentication methods that address attacks through injection of malicious code.

We propose a software tool – SPEE (secure program execution environment) – that combines static and dynamic code analysis and uses operating system utilities in order to provide a secure execution environment for application programs. It provides integrity and authentication protection, added during the compilation and loading process, both statically, before launching an executable, and dynamically, when program blocks are checked for integrity and control flow and authenticated at runtime. During the compilation and loading process, the SPEE tool adds a signature, and hashes, to the executable code and data. These signatures are added as a separate ELF section into the executable, and the operating system kernel performs checks to verify the signatures before authorizing execution. By using the Linux Security Modules [14], we are able to perform the necessary runtime verification of our secure programs. For the purpose of integrity checking, the tool employs a hierarchical hashing scheme wherein the executable is divided into blocks and are hashed forming a hierarchy of levels This method not only detects any changes to the executable, but also isolates where the changes have occurred in O(log N) time, where N is the number of blocks in the hash hierarchy. The goal of our tool is to provide a secure path from compilation to execution of application programs, while complementing the variety of software protection tools, which are designed for a particular kind of an attack. For example, FormatGuard [1] is designed for the purpose of protection against printf() format string vulnerability.

While our tool adds additional information to the executable, it does not interfere with the source code, nor does it add any additional code to the executable. It takes advantage of the executable format, specifically the ELF format, to add integrity and authentication information. Verification is performed by the operating system kernel, modified to handle *protected* executables.

The paper is organized as follows. Section 2 summarizes different approaches to software protection from the literature and motivates our approach. Section 3 discusses the overall system architecture of SPEE (our approach) and the key concepts and system utilities. Section 4 presents our current version of the SPEE tool which performs static verification and provides initial performance results and section 5 concludes our paper.

II. Related Work in Software Protection

Ideally, software should do what it is supposed to do and nothing else. Since it is almost impossible to create perfect software, various software protection methods exist. As stated in [6] they can roughly be divided into three broad categories: software auditing, vulnerability mitigation and behaviour management.

Software auditing is a category of methods, which try to prevent vulnerabilities before an attacker has a chance to exploit them. These techniques employ either *static analyzers*, which analyze source code, or *dynamic debuggers*, which look for abnormal behavior by subjecting the executable to various unusual inputs.

Static analyzers scan the source code and alert the programmer about potential vulnerabilities. These types of programs are usually used to assist a programmer in code reviews. Such tools as BOON [15] and CQual [8] scan C source code to find potential buffer overflows or inconsistent usage of values. MOPS [4] uses a finite-state machine model of what is considered valid behavior for a particular program. If a property is violated, *i.e.,* an illegal state is reached, during analysis the programmer is alerted about a potential vulnerability. Vulnerabilities such as the format string vulnerability [17] usually comes from sloppy programming thus making it easier to scan for it before the program is compiled. Static analyzers serve as good helper tools for the programmer, but they always run the risk of detecting too many false positives or false negatives. Some tools also take a considerable time to perform the analysis, in some cases a matter of several hours for a moderately sized program [6]. Other approaches to static code analysis include code obfuscation methods [5] and proof carrying code [11]. Both these approaches are susceptible to code tampering via injection of malicious code.

Vulnerability mitigation is a category of methods, which usually insert protection mechanisms into the code during compilation and detect vulnerabilities at runtime [6]. These techniques are not used to eliminate program bugs or prevent intrusions, but to detect them and prevent further program execution if such events occur. These techniques are designed to minimize the amount of damage from an attack.

One of the most common types of attack is buffer overflow [6], when an attacker overrides the input buffer in hopes to replace function's return address with the address of his/her own code. There are several tools designed to prevent buffer overflow attacks such as StackGuard [13]. They work by inserting integrity protection into the stack around the return address. FormatGuard's [1], purpose is to protect against the printf() vulnerability. This tool implements a wrapper around the standard printf() function to check the number of arguments passed to it.

These tools protect from a very particular vulnerability. There are also runtime tools that use a more general method of protection such as Guards [3]. In this case, small pieces of code (or guards) are inserted throughout the code during compilation. Each guard is responsible for checksumming a particular piece of code. If a tamper is detected, a special kind of repair guard is called. This method eliminates a single point to failure, but it not only increases code size, but is also vulnerable to code analysis which can remove the guards before execution.

Most runtime tools have a narrow focus on a particular security violation, although they take less of a performance penalty as compared to the static analysis tools.

Behavior management techniques are operating system features, such as access control policies, which are designed to limit the program's execution environment, thus limiting the amount of potential damage if a program is compromised [6].

Linux Security Modules (LSM) [14] is an open-source framework approved by the Linux development community. It allows a programmer to implement various access control mechanisms customized for the systems particular needs. LSM provides hooks [14], [16] into file systems, tasks, program loading, inter-process communication, kernel modules, networking, host and domain names, and I/O ports among others. LSM enables one to add mandatory access control on top of the regular Unix/Linux discretionary access control model. Several access control mechanisms have already been implemented as LSMs, and examples include Security Enhanced Linux (SELinux) [12], and Linux Intrusion Detection System (LIDS) [10]. All of these tools provide additional access control rules (for example, by dividing users and files into domains and types) to further restrict program execution thus limiting possible damage if a compromise occurs. LSM provides a wide variety of possibilities for a security system designer. However, it is observed that they do not prevent malicious code introduced into an authorized user's application from executing on the system thus leaving them vulnerable to attacks from tampered code.

III. Our Approach: SPEE

Software protection methods described above all contribute to creation of a trusted networked system. Each tool tries to solve a narrow problem of software protection. As one can see in Fig. 1, each particulate tool aims to solve a narrow software protection problem at a partcular stage. Static analysis tools (software auditing) aim to predict potential vulnerabilities by analysing source code. Runtime tools (vulnerability mitigation) attempt to protect from a particular vulnerability. Linux Security Modules (behavior management) aim to solve mandatory access control problems by imposing various restrictions on users, files, and processes. The software protection methods described above leave the system vulnerable to attacks from tampered code and do not provide code verification techniques to prevent an authorized user's code from being tampered with. This could lead to an insecure system for a number of reasons. Suppose a program is run through static analysis tools and no potential vulnerabilities have been identified. It is compiled and run (even with the presense of some number of above-mentioned runtime tools). But how do we know that the program has not been modified and malicious code has not been inserted during runtime? What happens if the inserted malicious code does not cause a printf() format string or buffer overflow? Would we not want this event to be detected nonetheless? It is observed that the problem of software integrity and software authentication has not been

Tools	SPEE
Software Auditing: BOON, CQual, MOPS, RATS, ElectricFence, etc	*Static integrity & authentication verification*
Vulnerability Mitigation: StackGuard, FormatGuard, ProPolice, etc	*Dynamic integrity and flow control protection*
Behavior Management: LSM, SELinux, LIDS, RaceGuard, etc	*Can be created as an independent tool or as part of OS kernel*

Fig. 1. Software Protection Methods

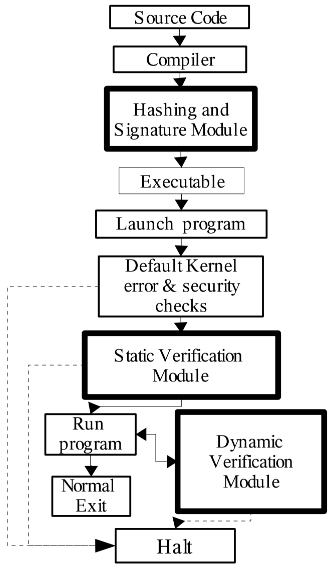
Source Code
Compiler
Hashing and Signature Module
Executable
Launch program
Default Kernel error & security checks
Static Verification Module
Run program
Dynamic Verification Module
Normal Exit
Halt

Fig. 2. Architecture of Secure Program Execution Environment (SPEE) Tool

widely addressed by the code security tools (whether we are talking about static, compile-time or runtime integrity verification). Our aim is to add to this collection of tools by concentrating on software integrity and authentication aspects of code security. As one can see in Fig. 1, SPEE aims to provide these services in every category of software protection. The static stage of SPEE provides executable integrity and authentication protection to ensure that the file has not been modified by an unauthorized user. At the vulnerability mitigation stage, software integrity as well as flow control protection are ensured at runtime. Our tool can also become part of the behavior management stage, as it can be implemented either as an independent tool, or as a Linux Security Module, or as part of the operating system kernel. By authorizing only trusted verified code to execute on the system, we aim to provide a trusted computing environment.

A. Overview of SPEE

Our goal is to create a secure program execution environment (SPEE) by complimenting the existing code security tools with the addition of program integrity checking and program/user authentication at various stages of the execution. The general framework of SPEE can be described as shown in Fig. 2. Our system is based on the interplay between the compilation process and operating system, and required the development of three modules (indicated in bold boxes in the figure).

The source code is compiled with a regular compiler (such as gcc), after which the newly created executable is run through a special module – the *signature and hashing module*, which adds integrity and authentication information. At this point, the executable is ready to be launched by a user. At first the regular operating system security checks are performed (such as whether or not the file exists or whether a user has privileges to run it). If the file passes this step, then an additional kernel tool – *Static Verification module* – checks the integrity of the executable and verifies its signature. After successful completion of this step, the executable is launched. As the program is executed, an operating system kernel – the *Dynamic Verification module*, which is responsible for running the program, performs additional integrity and authentication checks as program blocks are loaded and executed. By encoding the control flow of the program into the authentication information and signatures the dynamic tool permits only authorized control flow as defined by the properties of the program. If at any stage the verification fails, the execution is halted. This scheme will prevent unauthorized programs to be executed, as well as prevent modification of existing programs before and during execution. In other words, SPEE contributes to trusted software environment by providing a secure execution path from compilation to execution. As an additional benefit, it also provides forensic information in case of a verification failure by providing the user with information about which part of the program has been modified.

B. SPEE Implementation Components

Before we present a more detailed description of the SPEE architecture and implementation, summarize some of the concepts and system tools. Specifically, the ELF file format [17], [7], signed kernel modules[9], Linux security modules[14], and watermarking scheme[2] are key components of our system. These are needed to build our signature, static verification, and dynamic verification modules.

1) File Formats: ELF Binaries: Our tool will work with the ELF files. The Executable and Linking Format (ELF) is a particular standard binary file format, widely used in Unix/Linux systems [21].An ELF binary contains ELF and program headers and a number of sections, each with its own header. It provides a standard way for an operating system to create a program image and run it. ELF sections hold such information as the symbol table, dynamic linking, relocation or the actual program code. We are mainly interested in working with *.text* section, which contains the actual executable instructions, and *.data* section, which contains initialized data information. These sections will be checked for integrity and authenticated after the program is fully compiled and during runtime.

We utilize the property that the ELF standard allows users to add new sections to the executables. For example, DuVarney et al [7] used new sections to add security features (such as address obfuscation) to ELF executables. We use this feature to add additional sections containing all the necessary integrity and authentication information for our executables. Note that this does not interfere with the normal execution of the programs, since additional sections can be ignored.

2) Signed Kernel Modules: The concept of signed kernel modules is used in many operating systems. Kroah-Harman [20] first introduced the concept of signed kernel modules for the Linux operating system. Of relevance to our approach was the use of the ELF format for signature construction and verification. Since kernel modules in Linux are simply executables in ELF format, they can be manipulated to add hashes and signatures. One can examine ELF file sections by utilizing *readelf* program (part of *binutils*).

The signed kernel modules scheme extracted the executable code and initialized variables sections, ran them through a hash function (in this case SHA1 function from the kernel‘s cryptographic library), then signed the resulting hash using RSA algorithm (from the same cryptographic library). The signature then was added into the kernel module as a new section using another program from *binutils*, *objcopy*.

The Linux kernel code, responsible for loading kernel modules, was modified to check the modules‘ signatures first, before loading them. If the signature did not verify, the module has been either tampered with or improperly signed. It either case it was rejected.

The Kroah-Hartman scheme introduced the concept of adding integrity and authentication to Linux kernel modules. In SPEE, we extend this concept by adding integrity protection and authentication to any executable in the ELF format, including application programs. In terms of the SPEE architecture, ELF binaries and the concept of kernel modules are used to generate the signatures in the signature and hashing module soon after compiler stage (Fig. 2). The precise algorithm used in the hashing module is described in a later subsection.

3) Linux Security Modules: There have been many discussions on how to solve the discretionary access control problem and, as noted earlier in Section 2, Linux Security Modules (LSM) [14], [16] is one of the proposed solutions for the open-source community. LSM provides a general-purpose framework for implementing additional security solutions on top of the regular Linux DAC model. A Security system designer is presented with a wide range of ”hooks” into the operating system’s kernel. The hooks include file and process controls, networking and user control among others. Programmers choose which hooks to implement, thus creating their own customized security system. The hooks, which have not been implemented, use default operating system calls. Figure 3 depicts the LSM hook architecture.

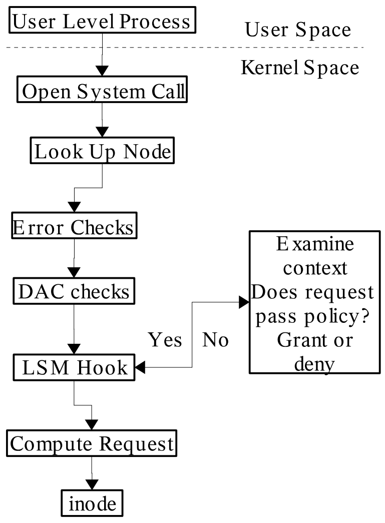

Fig. 3. LSM Hook Architecture

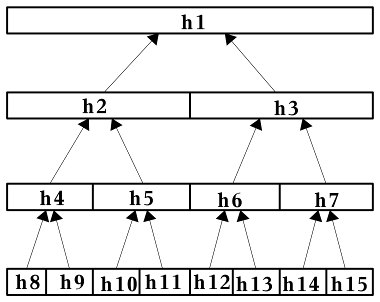

Fig. 4. Hierarchical Hashing Scheme

For the purpose of our system, we need an operating system kernel control for the runtime verification of our secure programs. Since LSM provides necessary hooks into files, processes, and program execution, SPEE utilizes the LSM framework for its dynamic verification module (Fig. 2).

4) Hierarchical Hashing: The static verification module in our system checks the signatures of the executable before authorizing the start of execution, and its effectiveness is related to the hashing and signature methods selected. Our hashing and signature scheme is based on a technique previously applied to image watermarking, but not explored in the context of software protection. Celik et al [2] describe a hierarchical method of hashing an image containing a watermark. This scheme computes a watermark of an image by dividing the image into blocks, which themselves form a hierarchy as shown in Fig. 4.

In general, the blocks of a lower level of the hierarchy form the blocks at the next level. At each level, block hashes are computed, and the hashes are inserted into the least significant bits of the image block. This way if part of the image is modified, it is possible not only to detect this event, but also to pinpoint the actual location of the modified bits. Celik et al [2] stress the importance of this feature, since it enables one not only to provide integrity protection for the image file in question, but also to provide localization of the error.

This technique is of interest to our SPEE framework, since by dividing the executable into blocks, we can also not only provide integrity protection and authentication, but also forensic information. By determining the exact part of the code, which has been modified, one can assess the potential intentions of the attacker.

C. SPEE Architecture

The SPEE Framework combines combine the methods and ideas of ELF, LSM, and hierarchical hashes, described above. The signed kernel modules method provides integrity and authentication only for modules. We would like to extend this concept to any program executed on a system. Depending on the user/security requirements, our method would apply to a wide range of executables from all to only specifically chosen security-critical ones (such as daemons or such programs as *passwd*, for example).

1) Static Verification and Authentication: During the compilation process, the executable has its code and data hashed and signed using the hierarchical hashing scheme. These signatures are added as a separate ELF section into the executable. For static verification, the kernel of the operating system performs checks before the executable is allowed to be launched. This is where the Linux Security Modules are useful, since they provide hooks into the file and process controls. The static verification process involves checking the hash and signature of the whole executable first. If the verification fails, only then the hash hierarchy is checked to not only let the user know that the executable has been compromised, but to also show the user the exact code segment modified. It can be shown that at the end of the static verification process, only a program that has not been modified after its compilation is allowed to proceed to execution. Thus, any program that has been tampered with is prevented from execution.

Note that we could have divided the executable into N chained blocks, but then the search time to find a modified block would have been in the order of O(N). Constructing the hashes in the form of a hierarchy (a binary tree in our case) is more efficient, since for N blocks the search time for a modified block is going to be in the order of O(logN). This traversal is only done in case of a top-level verification failure.

2) Dynamic Verification: If the static verification is performed successfully, the system allows launching of the executable and the dynamic verification module is responsible for preventing run-time attacks on the program. Our dynamic verification tool is based on key compiler concepts. The LSM framework hooks control the runtime verification of the executable by verifying the program blocks as they are loaded for execution. Our dynamic verification module is based on asserting the legal control flow in the program, *i.e.,* any changes made to the control flow graph of the program is equivalent to code tampering and the program is halted. The control flow of a program is usually defined using a control flow graph which captures that branch and loop conditions. We capture the control flow information of the program when we compute the hash of each block as described in what follows.

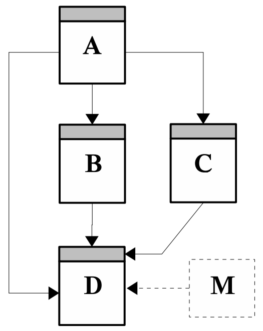

Fig. 5. Dynamic Block Verification Scheme

The hash of each block contains the information about both current and previous blocks as follows:

For every block X,

H(Y, X) denotes the hash of the block X, where

H - the hash function

Y - contains parents and children information

X - the actual block itself

This is done to preserve the control flow order and thus to prevent an attacker from inserting new code into the execution stream. For example, the program in Fig. 5 consists of four blocks A, B, C and D. B and C are only executed if their hash is verified and A is reported as their parent block. D is executed if either A, B, or C are the starting condition and D's hash is verified. Suppose a malicious block M is introduced (for example, a buffer overflow causes malicious code to start execution). M will fail the block signature and/or starting condition test, so it will not be executed. The program will be halted. This scheme will detect runtime attacks on an executable by preventing the execution of unverified code.

This approach does not require remembering the entire execution tree, just the parent block, so the verification can be done more efficiently. Guards [12] perform hash checking during execution, but are vulnerable to code analysis attack, when they can be altogether removed from the executable. Our approach does not insert additional code into the executable, but relies on the operating system kernel to perform both static and runtime verification. Thus code analysis would not be effective in this case. The issue remains to determine the initial verification condition for the first block. Performance testing will be done to determine the optimal block size, hash and signature algorithms to be used.

3) Analysis of SPEE: The SPEE tool is designed to protect from several kinds of attacks. The static verification module protects against attacks on the executable before it is launched, such as:

- attacks trying to replace existing executables with malicious ones with the same name
- attacks trying to modify existing executables' code section
- attacks on initialized variables section of the existing executable, when attackers might try to introduce their own initial values for the variables (in order to bypass a security check, for example)
- Static verification also offers protection agains expired keys. In any case, our tool makes sure that the executable has not been replaced or modified in any way and has been properly signed before it is launched.

The dynamic verification module protects from runtime attacks, such as:

- replay attacks – when program blocks are captured by the attacker and re-inserted into the program flow. Since dynamic verification not only checks block integrity but also control flow, these attacks will be prevented
- general control-flow attacks – which try to divert execution to attacker's code, will be prevented as well, since the newly-introduced program blocks will fail hash and signature checks.
- Since the tool does not add any additional code to the executable, the executable is not vulnerable to code analysis attacks, which try to remove protection code from the executable. An attempt to remove or replace hashes and signature will result in the execution being halted.

IV. Preliminary Experimental Results

Our system is being implemented under Linux (RedHat 9.0 and Fedora Core 2). Since the static hierarchical hashing scheme has been completed and tested, we present these results in this section.

The hash hierarchy is represented as a complete binary tree, i.e. the total number of nodes can be calculated at shown in Equation (1).

$$
\begin{aligned}
N &= 2^{(h-1)} && (1)\\
h &= log_2 N + 1 && (2)\\
N &= \frac{S}{B}, therefore && (3)\\
h &= log_2(\frac{S}{B}) + 1 && (4)
\end{aligned}
$$

$$S_h = S + H \cdot (2\frac{S}{B} - 1) \qquad (5)$$

where N $\rightarrow$ the number of blocks the executable is divided into (i.e. the number of nodes in a binary tree)
S $\rightarrow$ the size of the executable, in bytes
S_h $\rightarrow$ the size of the executable with the hash added, in bytes
B $\rightarrow$ the size of a block, in bytes
H $\rightarrow$ the size of a hash of an individual block, in bytes

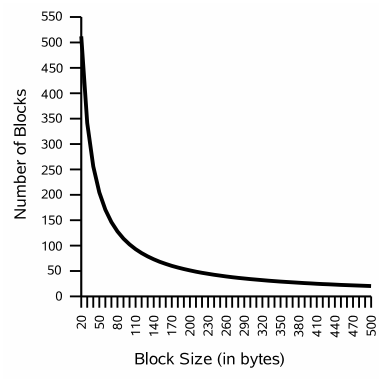

Fig. 6. Hash block size relative to the number of blocks required (10Kbyte file)

h $\rightarrow$ the number of levels of the hash hierarchy (or the height of the tree)

Thus the height of the tree (or the number of levels in the hash hierarchy) can be calculated as shown in Equation (2). Since the size of the file and the block size are known quantities, Equation (3) can be substituted into Equation (4), thus giving us the final formula for calculating the potential number of hash hierarchy levels, dependent on the file size and the size of a hierarchy block. Our evaluations considered a file of 10 KBytes and a hierarchy block of at least 20 bytes. Currently we used the SHA1 algorithm for hash calculations, which produces a 20 byte hash – note that this can be replaced by other hash schemes and we are currently evaluating various hash schemes. It would be impractical to choose a code block of less than 20 bytes. Based on Equations (1) - (4), the following dependencies have been calculated: Fig. 6 shows the block size (which the executable is divided into) is inversely proportional to the number of blocks created. The number of levels in the hierarchy is directly proportional to the number of blocks used. It has to be mentioned that a block used in this case is different from a disk block, although in practice a disk block size has to be taken into consideration.

Equation (5) shows the final file size of the executable after the hashes in the hierarchy have been calculated and added to the original file. The hashes are added as a stream of hash values (20 bytes each in the case of SHA1) as a separate ELF section of the executable. From Equation (5), we can observe that the file size is inversely proportional to the block size chosen for the hash hierarchy.

For a file of 10 Kbytes, with a block size for the hash hierarchy chosen at 300 bytes, and using SHA1 as a hash algorithm (i.e. the size of each resulting hash is 20 bytes) the static hierarchical hash file verification showed a 5.5% increase in the file size. The hash and signature verification is performed prior to the program execution. Thus it does not affect the program performance. Note that dynamic block verification scheme will affect performance, and this is the subject of our ongoing work.

V. Conclusions and Future Work

This paper proposed a software tool – SPEE – that provides a trusted computing environment via software protection. The goal of our tool is provide additional security to a computer system, especially a networked one, where software authentication and verification become especially important. Our system combines concepts from compilers, operating systems and watermarking to provide static and dynamic code verification and authentication. It prevents a number of attacks, including code tampering, replay attacks and control-flow attacks. In addition, it provides forensic information to the user about what exact part of the code has been attacked. The tool can be integrated into the operating system kernel, thus making it possible to perform software verification at the system level. Ongoing and future work includes completion of the dynamic verification module, and considerations of issues such such as key storage and assignment rules, frequency and method of rekey, optimal size and number of hash hierarchy blocks. We are also exploring the application of SPEE to various kinds of files, such as data files and databases, to provide integrity protection and authentication for data files to provide kernel level protection from unauthorized modifications.

Acknowledgments

The research is supported in part by NSF grant ITR-0325207.

References

[1] C. Cowan, et.al. "FormatGuard: Automatic Protection From printf Format String Vulnerabilities", *Proc. 10th USENIX Security Symposium*, Washington DC, August 2001

[2] M. U. Celik, G. Sharma, E. Saber, A. M. Tekalp, "Hierarchical Watermarking for Secure Image Authentication with Localization", *IEEE Transactions on Image Processing*, Vol. 11, No. 6, June 2002

[3] H.Chang, M.J. Attallah, "Protecting Software Code by Guards", *Proceedings of the 1st International Workshop on Security and Privacy in Digital Rights Management*, pp. 160-175, Nov. 2000

[4] H. Chen, D. Wagner, "MOPS: an Infrastructure for Examining Security Properties of Software", *Proceedings of ACM CCS*, 2002

[5] C. Colberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations", Technical Report, Dept of Computer Science, Univ. of Auckland, July 1997.

[6] C. Cowan, "Sotfware Security for Open-Source Systems", *IEEE Security and Privacy*, 2003

[7] D. C. DuVarney, S. Bhatkar, V.N. Venkatadrishnan, "SELF: a Transparent Security Extension for ELF Binaries", State University of New York at Stony Brook.

[8] J.S. Foster, M. Fähndrich, A. Aiken, "A Theory of Type Qualifiers", *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, May 1999

[9] G. Kroah-Harman, "Signed Kernel Modules", *Linux Journal*, Jan. 2004, pp. 48-53.

[10] LIDS, http://www.lids.org

[11] G. Necula, "Proof carrying code", *Proc. of POPL'97*, 1997.

[12] S. Smalley, C. Vance, W. Salamon, "Implementing SELinux as a Linux Security Module", NSA, NAI Labs, May 2002

[13] P. Wagle, C. Cowan, "StackGuard: Simple Stack Smash Protection for GCC", *Proceedings of the GCC Developers Summit*, pp. 243-256, 2003

[14] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel". *11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[15] D. Wagner, J.S. Foster, E. A. Brewer, A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities", University of California at Berkeley, *NDSS*, 2000

[16] C Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Modules Framework", *2002 Ottawa Linux Symposium*, Ottawa, Canada, June 2002.

[17] ELF Specification, http://www.muppetlabs.com/~breadbox/software/ELF.txt