SPEE: A Secure Program Execution Environment Tool Using Code Integrity Checking*

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 software vulnerabilities, 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 compilation process as well as operating system support to provide static verification of executables. Code integrity checking is performed by means of a hierarchical hashing scheme, which not only detects changes but also efficiently isolates them. This scheme provides a higher level of protection against code injection or modification than a simple chaining of the program blocks. 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. The SPEE tool is designed to function as part of the operating system kernel in order to provide a trusted computing system.

1 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 code that is downloaded over the Internet. It is becoming increasingly necessary to verify a program's correctness and integrity before execution. Ideally, users would like to run a trusted computing system, where all programs are only run by those authorized to do so and are not vulnerable to attacks either before or during execution.

Software protection has become an important issue in computer security [6] as the number and sophistication of attacks increase. Attackers exploit software vulnerabilities caused by programming errors, system or programming language flaws. Code is often attacked at

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

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 [16, 8, 4, 1, 14]. 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, many of 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 performs code integrity verification 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 before launching an executable. 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 [15], we are able to perform the necessary 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, which are hashed forming a hierarchy of levels. This method not only detects any changes to the executable and isolates them more efficiently than the simple linear chaining of blocks, but also provides a P^ℓ improvement in security level (where P is the probability of a hash collision and ℓ is the number of levels in the hash hierarchy) over the linear method (with probability P). 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. An example of a tool in the latter category is FormatGuard [1], which 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 or time 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 the analysis and experimental results of our current version of the SPEE tool, and section 5 concludes our paper.

2 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 and invulnerable software, various software protection methods have been devised. 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 [16] 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 [1] usually come from sloppy programming thus making it easier to scan for it before the program is compiled. Static analyzers serve as useful helper tools for the programmer, but run the risk of detecting too many false positives or false negatives. Some tools also take considerable time to perform the analysis, in some cases several hours for a moderately sized program [6]. Other approaches to static code analysis include code obfuscation methods [5] and proof carrying code [12]. Both these approaches are susceptible to code tampering via injection of malicious code.

Dynamic debuggers test program behavior under stress conditions. Such tools as ShareFuzz [19] present input that is bound to cause a program crash if there exists a printf() vulnerability or a potential buffer overflow in the code. ElectricFence [20] and MemWatch [21] prevent potential memory leaks by monitoring usage of malloc()'s and free()'s.

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 a function's return address with the address of their own code. There are several tools designed to prevent buffer overflow attacks such as StackGuard [14]. They work by inserting integrity protection into the stack around the return address. FormatGuard's purpose is to protect against the printf() vulnerability [1]. This tool implements a wrapper around the standard printf() function to check the number of arguments passed to it.

While the above 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 tampering is detected, a special kind of repair guard is called. This method eliminates a single point to failure. However, 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) [15] 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 [15, 17] 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) [13], and Linux Intrusion Detection System (LIDS) [11]. 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, 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.

3 Our Approach: SPEE

The software protection methods described above all contribute to the creation of a trusted computing system. Each tool tries to solve a narrow problem of software protection. Static analysis tools (software auditing) aim to predict potential vulnerabilities by analyzing 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 are identified. It is compiled and run (even with the presence 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? What happens if the inserted malicious code does not cause a printf() format string or buffer overflow? Our aim is to add to this collection of tools by concentrating on software integrity and authentication aspects of code security. 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.

3.1 SPEE Architecture

The general framework of SPEE is shown in Fig. 1. 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*,

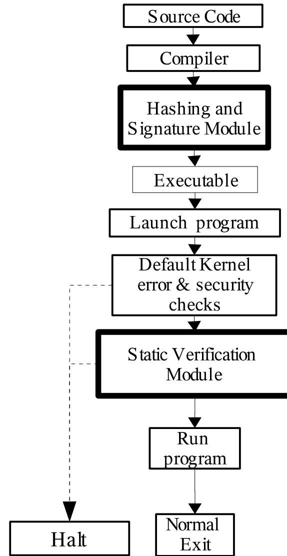

Figure 1: Architecture of Secure Program Execution Environment (SPEE) Tool

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. If at any stage the verification fails, the process is halted. This scheme will prevent unauthorized programs to be executed, as well as prevent modification of existing programs before 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. For run-time integrity checking, we are augmenting SPEE with the addition of a dynamic verification module using special purpose hardware such as an FPGA [9].

3.2 SPEE Implementation Components

Before we present a more detailed description of the SPEE architecture and implementation, we summarize the key concepts and system tools involved. Specifically, the ELF file format [18, 7], signed kernel modules [10], Linux security modules [15], and a signature scheme based on an image watermarking scheme [2] are key components of our system.

3.2.1 File Formats: ELF Binaries

The SPEE tool works with the ELF format files. The Executable and Linking Format (ELF) is a particular standard binary file format, widely used in Unix/Linux systems [18]. 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

```
There are 36 section headers, starting at offset 0x1d78:

Section Headers:
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .interp           PROGBITS        080480f4 0000f4 000013 00   A  0   0  1
[ 2] .note.ABI-tag     NOTE            08048108 000108 000020 00   A  0   0  4
[ 3] .hash             HASH            08048128 000128 000028 04   A  4   0  4
[ 4] .dynsym           DYNSYM          08048150 000150 000050 10   A  5   1  4
[ 5] .dynstr           STRTAB          080481a0 0001a0 00004c 00   A  0   0  1
[ 6] .gnu.version      VERSYM          080481ec 0001ec 00000a 02   A  4   0  2
[ 7] .gnu.version_r    VERNEED         080481f8 0001f8 000020 00   A  5   1  4
[ 8] .rel.dyn          REL             08048218 000218 000008 08   A  4   0  4
[ 9] .rel.plt          REL             08048220 000220 000010 08   A  4   b  4
[10] .init             PROGBITS        08048230 000230 000017 00  AX  0   0  4
[11] .plt              PROGBITS        08048248 000248 000030 04  AX  0   0  4
[12] .text             PROGBITS        08048278 000278 000160 00  AX  0   0  4
```

Figure 2: Sample readelf output

and run it. Fig. 2 shows a sample output of the program called *readelf*, which enables one to view section information for a particular executable (the output has been truncated for clarity). ELF sections hold such information as the symbol table, dynamic linking, relocation or the actual program code. We are mainly interested in working with the *.init*, *.text*, and *.fini* sections, which contain the actual executable instructions, and the *.data* section, which contains initialized data information. These sections will be checked for integrity and authenticated after the program is fully compiled.

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 by the loader.

3.2.2 Signed Kernel Modules

The concept of signed kernel modules is used in many operating systems. Kroah-Harman [10] 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*). A sample *readelf* output is shown in Fig. 2.

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

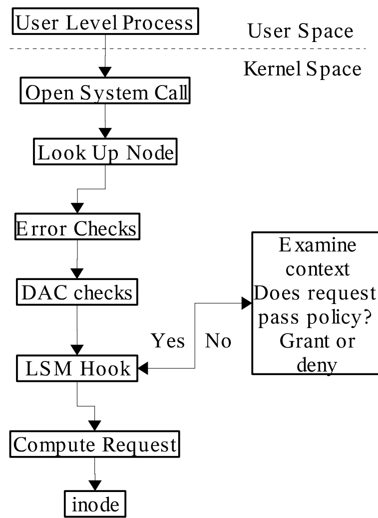

Figure 3: LSM Hook Architecture

modules' signatures first, before loading them. If the signature did not verify, the module has been either tampered with or improperly signed. In 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. 1). The precise algorithm used in the hashing module is described in a later subsection.

3.2.3 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) [15, 17] 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. For the purpose of our system, we need 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 verification module (Fig. 1).

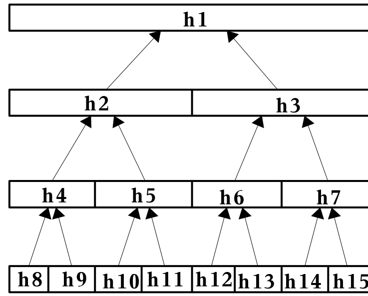

Figure 4: Hierarchical Hashing Scheme

3.2.4 Hierarchical Hashing based Signatures

The static verification module in our system checks the signatures of the executable before authorizing the start of execution. Its effectiveness is related to the hashing and signature methods selected. Our hashing and signature scheme is derived from a technique previously proposed for 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 stronger 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.

3.3 Code Verification and Authentication in SPEE

The SPEE Framework combines 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 extend this concept to any program executed on a system. Depending on the user's 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).

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

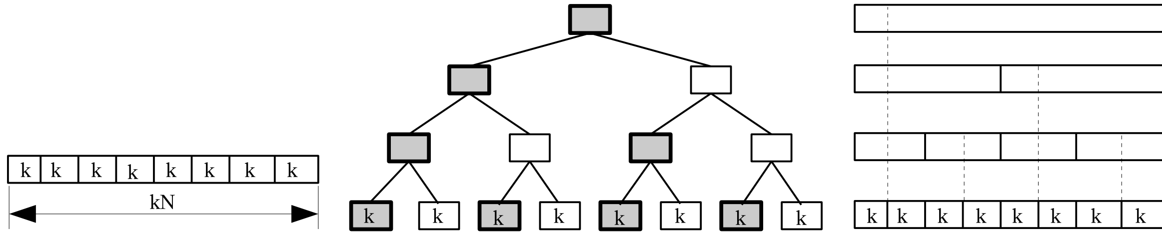

Figure 5: Linear vs Hierarchical Hashing Methods

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.

The crucial components, from the viewpoint of security, are the Signature and Hashing module and the Verification module. Given its significance, we next analyze its effectiveness and justify the reasons for choosing a hierarchical hashing scheme as opposed to a simple division of the executable into a series of chained blocks, *i.e.*, a linear hashing method. The ensuing discussion refers to Fig. 5.

3.3.1 Analysis of the Hierarchical Hashing Technique.

Before we begin the analysis of advantages of the hierarchical hashing technique over the linear hashing technique, let us present a short overview and notations for both methods. After compilation, the program is assumed to be a file of N blocks, each having k bytes. The linear hashing scheme consists of computing the hash of each block i (for i between 1 and N) and the hash of the entire file containing all N program blocks. The hierarchical hash structure resembles a full binary tree, where the leaves are the N program blocks. We start by computing the hash of the entire file containing all N program blocks (as in the linear case) and then proceed to the hash hierarchy construction, where at each level i, the left child of a node is the hash of its left half, and the right child of a node is the hash of its right half. The hash hierarchy contains ℓ levels (where $\ell = log_2 N + 1$). We will denote the hash function we are using as $h(k)$ (*i.e.*, the hash of a block of k bytes), where h can be any secure hash function.

The hierarchical hashing method provides stronger integrity protection than the linear method. Hash functions are vulnerable to a birthday attack [22], wheh the attacker tries to replace a block in the code with another one, which results in the same hash, *i.e.*, causes a hash collision. Let $P(h)$ be the probability of finding a block, which would result in a

hash collision using the hash function h. In the linear scheme, we would compute the overall hash of the file, and only if it does not verify would we start computing hashes of blocks in sequence until we reach the compromised block. The probability of finding a block, which would result in a hash collision is $P(h)$. But since the attacker would also have to make sure the overall hash is unchanged as well, the total resulting probability is $P^2(h)$ for the linear hashing method. Suppose that a hierarchical hash is used with ℓ levels in the hierarchy. The probability of finding a block, which would leave hashes unchanged at every level of the hierarchy is $P^\ell(h)$ (where $\ell = log_2 N + 1$, as we are using a complete binary tree). It can be clearly seen that it is harder to compromise the hierarchical hashing method, with the number of levels greater than 2, as compared to the linear method. Since the probability of finding a collision is now $P^\ell(h)$ as opposed to $P^2(h)$, we can say that the hierarchical method is ℓ (or $log_2 N$) times harder to circumvent (*i.e.*, inject or replace code) than the linear method.

There are two stages to both linear and hierarchical hashing methods: the calculation and the verification. Let us look at the calculation stage first. We assume that the file contains Nblocks (or leaves, as we are using a binary tree for the hierarchical hashing method), each of size k and that $h(k)$ is the hash function used. The calculation stage of the linear method involves the calculation of the overall hash and the hashes of each block as shown in Equation $eq1$. The first term contains $N-1$, since while computing the overall hash of the file, we also compute the hash of the first block of size k.

$$(N-1)h(k) + h(kN) \tag{1}$$

The calculation stage of the hierarchical (in our case binary tree) method involves calculating the hash of each node of the tree, as well as the overall hash of the entire file as shown in Equation 2.

$$\sum_{i=0}^{log_2 N} \frac{N}{2^i} h(2^i k) \tag{2}$$

Based on Equations 1 and 2, the hash calculation stage is faster using the linear method, but we are now going to show that at the verification stage, the hierarchical method does not differ in verification time from the linear hashing method.

The verification stage of the linear hashing method involves hash calculation and verification of the overall hash as well as of each block in the sequence as shown in Equation 3.

$$(N-1)h(k) + h(kN) \tag{3}$$

The verification stage of the hierarchical method involves calculation and verification of the overall hash as well as at least two hashes at each level of the hierarchy as shown in Equation 4.

$$2\sum_{i=1}^{log_2 N} h(\frac{kN}{2^i}) + h(kN) \tag{4}$$

Let us now look at Fig. 5 and notice that while we are calculating the overall hash value of the file (in the hierarchical method), we are also calculating intermediate hash values, which are hashes of the left children on each level of the subtree (we assume that we are using a streaming hash algorithm). As we can see, now our hierarchical method's verification stage involves computation and verification of the overall hash as well as 1 more hash calculation at every hierarchy level (the right child) as shown in Equation 5.

$$\sum_{i=1}^{log_2 N} h(\frac{kN}{2^i}) + h(kN) \tag{5}$$

If we compare Equations 3 and 5, we can see that the hierarchical method is not slower than the linear method. This is proven as follows.

$$\sum_{i=1}^{log_2 N} h(\frac{kN}{2^i}) + h(kN) \leq h(k) \sum_{i=1}^{log_2 N} \frac{N}{2^i} + h(kN), then \tag{6}$$

$$h(k) \sum_{i=1}^{log_2 N} \frac{N}{2^i} = Nh(k)(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{log_2 N}}) \tag{7}$$

Observe that the inner term is a geometric series, such that $\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^l} = \frac{2^l - 1}{2^l}$. In our case $l = log_2 N$, therefore $\frac{1}{2} + \cdots + \frac{1}{2^{log_2 N}} = \frac{2^{log_2 N} - 1}{2^{log_2 N}} = \frac{N-1}{N}$. Substituting this value into Equation 7, we get the following

$$Nh(k)(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{log_2 N}}) = Nh(k)\frac{N-1}{N} = (N-1)h(k) \tag{8}$$

If we substitute the result of Equation 8 into Equation 6, then the resulting Equation 10, shows us that the hierarchical hashing method is not greater in verification time than the linear method, shown in Equation 3.

$$\sum_{i=1}^{log_2 N} h(\frac{kN}{2^i}) + h(kN) \leq h(k) \sum_{i=1}^{log_2 N} \frac{N}{2^i} + h(kN) \tag{9}$$

$$\sum_{i=1}^{log_2 N} h(\frac{kN}{2^i}) + h(kN) \leq (N-1)h(k) + h(kN) \tag{10}$$

We must also note that the number of hash comparisons (*i.e.* verifications) is in the order of $O(N)$ in the linear case, as opposed to $O(logN)$ in the hierarchical case.

In summary, we have chosen the hierarchical hash method because it provides greater security protection for a file and does not differ in hash verification time from the linear method. The initial hash calculations take longer, but this is done once in the program's installation stage, thus providing no performance penalty.

4 Analysis and Experimental Results

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.
- By providing information on which part of the code has been modified, static verification can also provide "forensic" information about the attacker's intentions, thus helping fix the code vulnerability for the future.

We next discuss an analysis of our implementation including experimental results. Our system is implemented under Linux (RedHat 9.0 and Fedora Core 2). The hierarchical hashing scheme can be represented as a complete binary tree, i.e. the total number of nodes can be calculated at shown in Equation 11.

$$N = 2^{(\ell-1)} \tag{11}$$

$$\ell = log_2 N + 1 \tag{12}$$

$$N = \frac{S}{k}, therefore \tag{13}$$

$$\ell = log_2(\frac{S}{k}) + 1 \tag{14}$$

$$S_h = S + H \cdot (2\frac{S}{k} - 1) \tag{15}$$

where N $\rightarrow$ the number of blocks the executable is divided into

S $\rightarrow$ the size of the executable, in bytes

S_h $\rightarrow$ the size of the executable with the hash added, in bytes

k $\rightarrow$ the size of a block, in bytes

H $\rightarrow$ the size of a hash of an individual block, in bytes

ℓ $\rightarrow$ 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 12. Since the size of the file and the block size are known quantities, Equation 13 can be substituted into Equation 14, 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

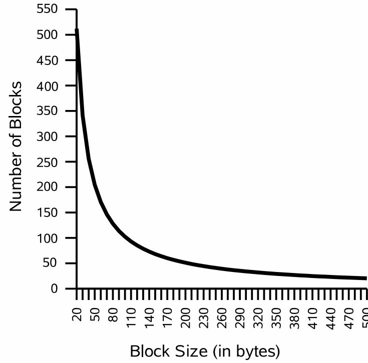

Figure 6: Hash block size relative to the number of blocks required (10Kbyte file)

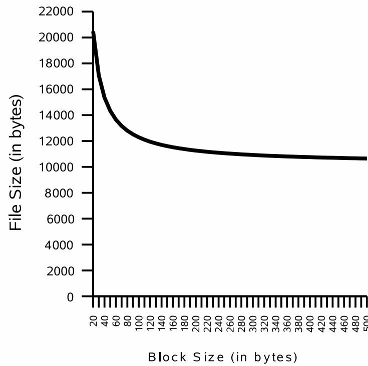

Figure 7: Increase in file size relative to hash block size (10Kbyte file)

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 in Equations 11 - 14, the following dependencies can be calculated: Fig. 6 shows the dependency of the number of hierarchy levels on the block size chosen. The number of levels in the hierarchy is directly proportional to the number of blocks used (see Fig. 8). Based in Fig. 6 and Fig. 8, block size of 300 - 350 bytes was chosen for the experiments. This is also supported by Equation 15, which shows the final file size after the hashes in the hierarchy have been calculated and added to the original file as a separate ELF section. Fig. 7 shows that tile file size is inversely proportional to the block size chosen for the hash hierarchy. The block size of 300 - 350 bytes proves to be optimal in this case as well.

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 at run-time will affect

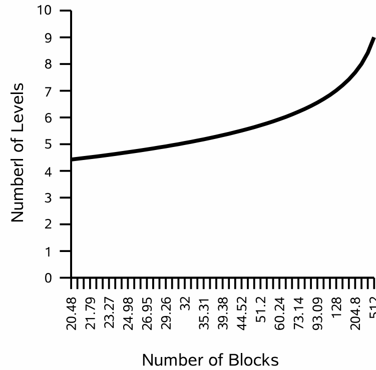

Figure 8: Number of blocks relative to number of hierarchy levels (10Kbyte file, 20Byte hash size)

performance, and this is the subject of our ongoing work. A dynamic verification scheme using special hardware, that was added to our SPEE framework was presented in [9].

5 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 code verification and authentication thereby preventing code tampering 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 addition of a 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 authors wish to acknowledge Poorvi Vora and Eric Noriega for their many valuable suggestions.

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*, Nov. 2000, pp. 160-175

[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, "Software 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] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, J. Zambreno, "CODESSEAL: Compiler/FPGA Approach to Secure Applications", *Proceedings of IEEE International Conference on Intelligence and Security Informatics*, Atlanta, GA, May 2005, pp.530-536.

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

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

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

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

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

[15] 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.

[16] 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

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

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

[19] ShareFuzz, http://www.atstake.com/research/tools/vulnerability_scanning/

[20] ElectricFence, http://perens.com/FreeSoftware

[21] MemWatch, http://www.linkdata.se/sourcecode.html

[22] RSA Security, "Crypto FAQ", http://www.rsasecurity.com/rsalabs/node.asp?id=2205