An Architecture for Kernel-Level Verification of Executables at Run Time

LUIGI CATUOGNO[1] AND IVAN VISCONTI[2]

[1]*Dipartimento di Informatica e Applicazioni, Università degli Studi di Salerno, Via S. Allende, 84081 Baronissi (SA), Italy*
[2]*Laboratoire d'Informatique, École Normale Supérieure 45, rue d'Ulm, 75230 Paris Cedex 05, France*
Email: luicat@dia.unisa.it, ivan.visconti@ens.fr

Digital signatures have been proposed by several researchers as a way of preventing execution of malicious code. In this paper, we propose a general architecture for performing the signature verification as part of the kernel execution process. The proposed architecture does not require any change in the interpreters used to execute code and it can accommodate any executable format. We also report on our implementation for the Linux operating system that focuses on ELF and script executables. Experimental results show that our solution is of potential interest as virtually no slowdown is experienced in the execution.

Received 24 June 2003; revised 4 November 2003

1. INTRODUCTION

External intrusion is one of the most serious threats to the security of a system that is connected to a network. Typically, an attacker exploits a bug of a network daemon to obtain superuser privileges in the form of a shell session owned by root.

Once this has been accomplished, the attacker has complete control of the system and access to all the data stored on the machine. Obviously, if for some reason the machine is rebooted, the attacker has to start again. Moreover, the legitimate administrator of the system could detect the ongoing intrusion, kill the shell session and terminate the intrusion. If the software bug that allowed the intrusion has been discovered, the system administrator can install a new version of the network daemon and thus the attacker cannot repeat the same attack on the machine; instead, he or she has to find another weak daemon or exploit a weakness of the new version of the same daemon (which, unfortunately, most of the times is easy to do). We refer to this form of attack as a weak intrusion attack.

A more serious threat comes from an attacker that, once root privileges have been gained, tries to colonize the system; i.e. the attacker tries to keep control of the machine across reboots. We refer to this kind of attack as a strong intrusion attack. One way of doing this is to install new malicious code and/or modifying existing executables. This has two main goals. First of all, every time the legitimate system administrator executes the modified executables the attacker regains control of the machine in the sense that the malicious code is executed again. Second, the attacker modifies tools used by the administrator in order to hide the ongoing activity. For example, the `ifconfig` command could be replaced by its corresponding malicious version, which hides the promiscuous mode of a network card used to sniff passwords and a fake `ls` command could hide new executable files installed in a directory. Using these techniques an attack could become resistant to reboots and attack detection strategies.

In this work, we address the problem of strong intrusion attacks and present a security architecture that prevents the installation of malicious code that can be executed across reboots. Thus, as a consequence, an attacker cannot use the techniques described above to keep control of a machine permanently. We stress that our work does not address the general problem of buggy software (e.g. daemons that are subject to buffer overflow attacks) or, in general, the problem of system intrusion. Our architecture does not guarantee that an attacker cannot obtain root privileges but only that, once the attack has been successful, the administrator can detect it. In other words, we reduce the strong form of intrusion to the weak one.

In Section 2, we present some of the techniques used by the intruders for their malicious purposes. In Section 3, we describe the most interesting proposed architectures (only some of them are currently implemented) to protect servers. In Section 4, we propose our approach against strong intrusion attacks. We have implemented the proposed infrastructure and in Section 5 we show details of our implementation; in particular, in Section 5.5 we illustrate the management of a cache that has an important role to make our solution practical. In Section 6, we propose two solutions for a possible attack to our architecture. In

Section 7, we discuss some open issues about the execution of application in a UNIX-like system, and their implication on our architecture. Finally, in Section 8, we conclude the paper by re-emphasizing the features of our approach and by pointing out some open questions.

2. TECHNIQUES FOR INTRUSION

In this section, we describe the main techniques used by intruders in order to gain and maintain the control of a machine. We will stress the potential threat passed by each technique so that we can later contrast it with the benefits of our architecture.

2.1. Rootkits and code modification

A rootkit is a subset of common system commands and daemons that have been corrupted in order to perform malicious operations, hide their effects, and possibly set up some 'back-doors'. Once an intruder obtains the access to a victim machine, he or she installs the kit. Since malicious executable files take the place of the widely used commands (e.g. UNIX commands like `ls`, `ps`, etc.), execution of malicious code is performed by legitimate users of the system including the superuser. We illustrate the following scenario as an example. An intruder installs a network sniffer [1] (e.g. `tcpdump` can be used for this purpose) that periodically logs the network traffic. In order to prevent the detection of his or her malicious activity, the intruder replaces the commands `ls` and `ps` with their respective corrupted counterparts, which avoid the visualization of the sniffer's log files, and hide the sniffer's process. Moreover, the intruder can also install a malicious version of the `ifconfig` command, in order to hide the network interfaces that have been set in promiscuous mode by the sniffer.

It is also technically possible to inoculate 'parasite code' into executables of some binary format (without re-compiling them), taking advantage of certain properties of their memory image. In such a way, an intruder could even modify applications that he or she cannot replace. However, this technique is cumbersome, inefficient and it strongly depends on the operating system, the hardware architecture, the binary format and the memory layout of the target executables.

Moreover, many UNIX distributions include several scripts (written in several interpreted languages like Perl, Python and so on) that automatically take care of set-up and configuration procedures of the operating system. Malicious modifications to these scripts could invalidate software re-installations or upgrades avoiding the removal of malicious code or misconfiguring the system, thus guaranteeing backdoors and vulnerabilities for future intrusions.

These attacks cannot be detected easily and they can be so invasive that recoveries often require the re-installation of the operating system from scratch.

2.2. Installation of untrusted software

Another threat to system integrity is the download and installation of untrusted software. Many packages are currently distributed over the network already in binary format with no integrity check information, thus there is no guarantee that the application has not been tampered. Actually a digest of each file that can be downloaded is also published so that the user can check the integrity of the file. However, the corruption can affect the digest as well and even if a secure and authenticated channel is used (e.g. by using the TLS [2] protocol), the corruption of the files cannot be detected. In these cases, it is very difficult to realize what the installed application really does.

2.3. Code injection via buffer overflows

Buffer overflow [3] is probably one of the most serious software vulnerabilities. Some applications (e.g. daemons that provide network services) do not take much care (or do not take care at all) about the bounds of data areas during their execution. Thus, it could happen that by providing a network daemon with an amount of input data that is greater than the one assumed by its designer, some areas of the process image that are contiguous to the I/O buffer could be overwritten. In this case, process data, behavior and even execution flow can be altered. Attackers can take advantage of this weakness by inserting malicious code into the process image, and then by starting its execution. These attacks (and their related defenses) have been studied following different approaches. We invite the reader to consult [4] and [5] for details. As mentioned above, the prevention of buffer-overflow attacks is out of the scope of this work.

2.4. Run-time kernel corruption

Several UNIX-like systems allow to load into memory, on demand, some sections of the kernel at run time. These sections are named loadable kernel modules (LKMs). Usually, the modules provide new features to the system as filesystems, device drivers and so on. Unfortunately, there is no way to prevent an intruder, who has gained root privileges, from pushing malicious code into the kernel using a loadable module. As is shown in [6], an LKM could access and modify any kernel data structures, even the system call table. Thus, for example, the intruder could redirect some process system calls to its own table and, in such a way, he might modify the behavior of the processes (even the verification tools) without modifying them.

Moreover, the devices that represent the memory of the system (e.g. Linux's `/dev/kmem`), allow the processes in user-space (executed with root privileges) to perform read/write operations on the memory and can be used to write malicious code directly to the memory, or stealing secret data and encryption keys. An example of this attack can be found in [7].

We point out that no verification strategy can be used successfully if the kernel is not trusted. Thus, we make the following assumptions:

(i) The kernel boots in a secure state (see Section 4.1).
(ii) The LKMs support is disabled.

(iii) Write operations to `/dev/kmem`-like devices are not allowed.

3. RELATED WORK

In this section, we briefly discuss the most used solutions to prevent the execution of malicious code.

Database of known malicious code segments. A traditional approach employed to detect malicious code is the one followed by the usual virus detectors like some of the products from McAfee [8] and Symantec [9]. These tools are provided with a database containing characteristic code-segments from many known viruses. The user can configure the antivirus to run periodically, and then the tool simply searches for possible occurrences of the segments into some files. Several drawbacks of this approach are easily identified. First of all, running the antivirus significantly affects the machine performances; second, the database of segments has to be upgraded constantly, and moreover, it is always possible that at any time, the detector will not recognize a virus that has not been yet (or will never be) covered by the antivirus provider.

Tripwire. Tripwire [10, 11] is one of the most widely employed tools for the prevention of unauthorized modification of files. Tripwire's approach comprises in storing in a secure database, a digest of each file currently present in the file-system. Periodically, an agent computes the digests of each file in the file-system and checks it against the digest stored in the secure database. If a mismatch is found an alert message is sent to the system administrator. The main weakness of this approach is that malicious code activities are allowed between two executions of the agent. If the agent checks the file-system very often, then the performance of the system could be heavily affected.

File-system-based approach. A different approach moves the task of checking the integrity of files and executables to the file-system layer. This can be achieved in two ways. The most obvious solution is to store the executables on a read-only support (e.g. a CD-ROM or a DVD). Indeed, several Linux distributions provide a live file-system on a read-only support that can be used as a rescue system. In this case, the system administrator first installs and configures the system and then he connects the system to the network. The main drawback of this approach is that software upgrading is particularly cumbersome.

In a more sophisticated approach, each file is digitally signed and the signature is checked by the file-system layer each time the file is opened or read. Different implementations of this concept (or of some of its variations) exist (the Transparent Cryptographic File-System [12], the Read Only Self-Certifying File-System [13] and the SUNDR filesystem [14]), but we believe that the file-system approach suffers due to some fundamental drawbacks that make it unsuitable for our setting. Indeed, we would like to make it possible for software developers and distributors to sign their distributions so that system administrators can verify the source of the software they install. If signature checking occurs at the file-system level, the format of the signed files must take into account the file-system that would eventually host them and should take care of their verification. This implies that software distributors and system administrators must agree upon a common file-system to handle the binaries. Worst still, even if a common file-system is agreed upon then it must be the only file-system used on the system. Indeed, since executables are only verified at the file-system layer, potentially malicious executables that live on a different file-system will be executed without any check.

Integrity verification based on reflection. An interesting approach, outlined by Spinellis [15], consists of allowing software to 'answer to some questions about itself' in order to realize possible unauthorized modifications. More precisely, software to be verified is assumed to run on a remote device (the client) with some expected environmental parameters like the processor status, assumptions about the content of the device's unused memory and predictions about processor performance on known sequences of instructions. A trusted entity, called server, periodically asks the client for the cryptographic hash of some randomly chosen intervals of memory and a summary of the processor status. Under some assumptions and according to the knowledge over the client environment, the server can realize whether the client has been tampered with. Unfortunately, a 'general-purpose' scenario (as could be a UNIX server connected to the Internet) does not satisfy all assumptions stated in [15] and making assumptions or predictions about the behavior of the monitored software and the host system can be really difficult, especially if the set of clients is heterogeneous.

Another approach to verify that honest software is being executed is given by Lie *et al*. [16]. This work proposes an architecture that provides an idealized model of the execution of a given application. This model is compared with an actual model that comes from the execution of the monitored software, and that includes 'adversary' operations. Whenever the actual model (i.e. the real program) does any transition that makes it inconsistent with the ideal model, the architecture guesses that the machine is under attack.

Moreover, we suggest the reader looks at [17] in order to have more details on the development of trusted software.

4. VERIFYING EXECUTABLES AT RUN TIME

In this paper, we follow the approach of adding digital signatures (see [18] for more details) to executables and verifying the signature as part of the execution process (and not at the file-system level) as depicted in Figure 1.

The main advantage of using digital signatures is quite evident. Executables can be signed off-line, i.e. when the system is not connected to the network and is under the complete control of the legitimate software issuer. Only in this state (which we call secure mode) does the system have access to the private key. Once the 'signature process' of new executables ends, the system can be switched in normal mode, i.e. re-connected to the network. As it follows trivially from

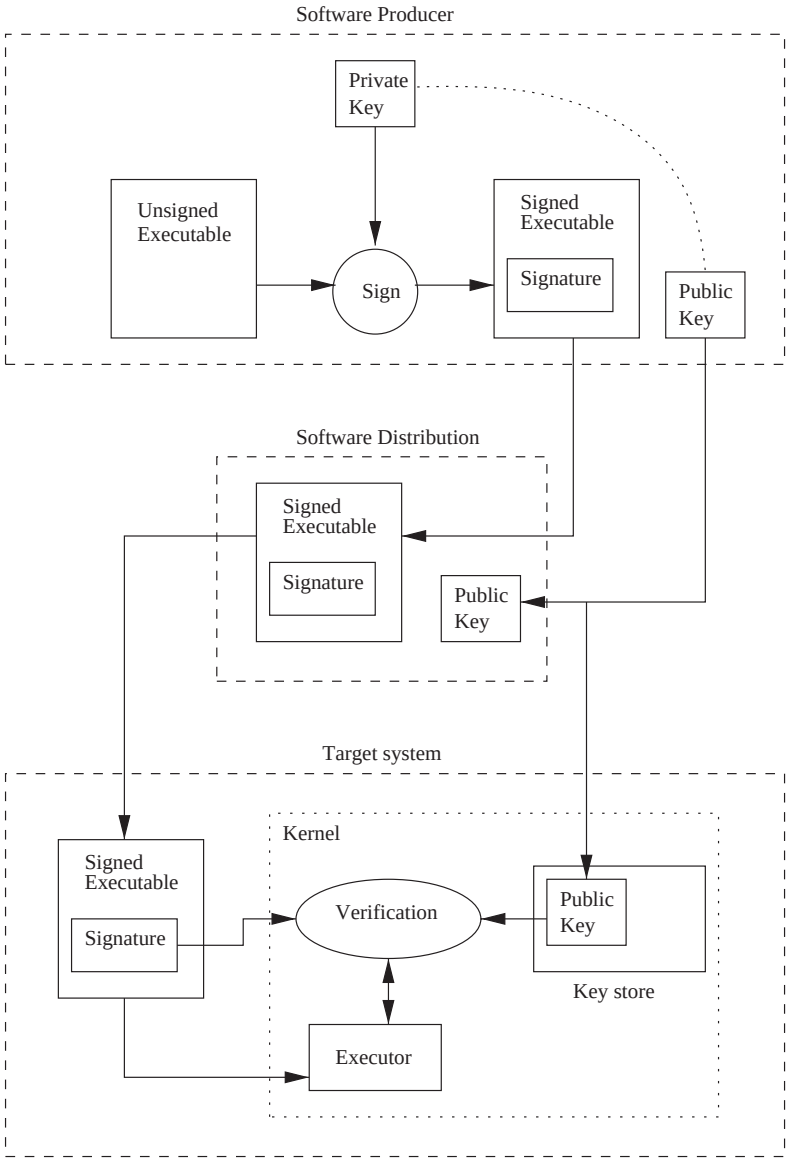

FIGURE 1. Verification at run time of executables.

the properties of asymmetric cryptography, the knowledge of the private key is not necessary to verify the signature. Thus, systems in normal mode do not need to access the private key. Moreover, in order to be protected, the private key could be stored on an external memory device (e.g. a smart card), and made available to the system only when it is in secure mode.

The idea of signing executables in order to be able to authenticate an executable before it is executed is contained in [19] as part of a comprehensive theory of authentication in distributed settings. Based on the theory outlined in [19], [20] describes the implementation of a secure authentication system for the Taos operating system. There, in a way similar to Kerberos [21], an authentication agent (one for each node of the network) handles the checking of all credentials (from processes wanting to access a printer to users seeking access to a file) and applications access the local agent through a well-defined interface. In [20], the authors describe the implementation of the authentication system for Echo, a distributed file-system extensively used within Taos. Our approach follows the same line of thought and, motivated by the need of combating intrusion detection, focuses on the authentication of executables. This allows us to dispense with the need of an authenticating agent and the need to re-write applications in order to interface with the agent. As we will explain later, our approach calls for a minimal modification of the operating system and does not affect existing and legacy applications. The idea of signing executables has also been applied to Java classes and Java archives [22, 23] in a PKIX scenario [24]. However, the approach is only limited to the Java virtual machine.

Our approach instead follows the lead of van Doorn *et al.* [25], who proposed a mechanism to sign and verify Executable and Linking Format (ELF) binaries [26, 27] for the Linux operating system. The approach of [25] is very

close to ours and in the rest of this section we present the architecture of [25] and stress some points which we consider as the weaknesses of their approach.

Signatures of ELF binaries are computed using the MD5 hash function and the RSA digital signature scheme [28]. A signature is added to an ELF executable by storing it in a new ELF segment.

When an executable is loaded, the ELF format manager extracts the signature from the file and verifies each referenced segment. If the verification fails then the execution is aborted.

The kernel verifies the signature of text segments and provides a new system call: `verify` that should be called by user-space interpreters (e.g. `ld.so`, `sh`) in order to verify dynamically loaded objects and script files. This choice, however, implies that every interpreter should be modified in order to request signature verification. Moreover, as pointed out in [25], this approach increases the number of points where the signature verification takes place, and does not cover all those scripts that are given to interpreters as input files (this problem is not solved by our solution either).

The architecture presented in [25] features a cache in order to avoid the verification of signatures at each execution. Each file-cache entry contains the pathname of the cached file and the result of the last verification. When the system executes a cached file, it checks the related cache entry. If that entry is still valid (i.e. the file has not been modified since the last verification), the kernel will not verify the file signature again. On the other hand, the kernel traces each `open` invocation and, if the subject of the call is a cached file then the corresponding entry is invalidated. Unfortunately, since it is not possible to trace accesses to files stored in remote file-systems, only files on local volumes can be cached.

Before introducing our approach, we summarize some of the aspects of the work of van Doorn *et al*. [25]:

(i) the kernel directly verifies the ELF binaries that are loaded by the `exec` system call;
(ii) executable scripts and dynamic segments of ELF binaries are verified in user-space by their respective interpreters; and
(iii) in order to verify its executable files, each interpreter has to invoke the `verify` system call. In other words, all interpreters (e.g. `ld.so`, the Perl interpreter) have to be modified in order to verify and run signed executable files.

4.1. Our approach

In our approach we follow the lead of [25] but we propose a new and more flexible architecture. Indeed, we look at the execution process at the kernel level and modify it to introduce signature verification in a manner that is independent of the format. More precisely, all verification steps are performed inside the kernel by the handler of the executable format. The user space interpreters of the various formats only perform the tasks they have been designed for. This has several advantages.

(i) We can fit any executable format that is known to the kernel into our architecture.
(ii) All verification steps are performed at the kernel level.
(iii) Interpreters need not be modified.

Thus, our approach is not limited to one executable format (e.g. ELF) and does not need to modify the user space interpreters of the various formats. Moreover, we have to remark that our approach fails in two notable cases: the first case happens when the executable code inside a shared object is dynamically loaded by invoking the function `dlopen` of the `libc` library; the second case happens when an interpreter executes code that is not handled by the kernel.

To validate our approach we also discuss the implementation of the architecture for the Linux operating system. In Section 5.7, we describe the implementation issues we had to address.

We emphasize two important assumptions on which we based our work.

(i) At the end of system bootstrap, the system is assumed to be sane. The AEGIS [29] project proposes the design of a secure bootstrap system with a high assurance bootstrap process in which the integrity of the kernel loaded at boot time is guaranteed. In [30], an improvement to AEGIS called sAEGIS has been proposed to protect users from malicious administrators supporting a large set of operating systems. We assume that our infrastructure lies on a secure boot system like AEGIS and such that a system takes care of collecting (during the bootstrap) all public keys required to verify all signed executable files.
(ii) All types of executable files of any format have to be signed before they are installed on the system. The entity that distributes the package uses its private key to sign all files that contain executable code and each signature is appended to the corresponding file. This approach perfectly fits the software distribution scenario. Consider a major Linux distribution (e.g. SuSE, RedHat, Debian, Mandrake) with its pair of public/private RSA keys. When a new release of the distribution is available, all executable files are signed using the private key and then the signed executable files along with the public key are released. Each time the execution of an executable is requested, the signature is checked by using the trusted public keys.

Since different formats of executable files are structured and parsed in different ways, we provide a signing tool and a verification procedure for each executable format. For example, in order to allow the signature of scripts, we provide the utility `scriptsign` to be used to sign a script and the API function `verify` for verifying signatures. We point out that, as discussed in Section 5, our `verify`, unlike the one in [25] is not called by the user-space script interpreter (e.g. shell, Perl) which, consequently, need not be modified since our `verify` is invoked by the kernel handler.

Moreover, our proposed architecture satisfies the following requirements.

(i) Integrity of executable files is verified at run time. This property guarantees that no untrusted executables is executed by the system unless it is currently under a successful attack. As we pointed out in the Introduction section, we do not aim to make intrusions impossible, but only to limit intrusions to transient ones.

(ii) Signed executables are completely compliant with non-verifying handlers (i.e. it must be possible to execute a signed executable even on a system that does not feature signature verification). This property guarantees that executables can be signed by the developers, and then used both on systems that feature signature verification and on systems that do not. Otherwise, developers should be forced to release two versions of the same software which is inconvenient.

(iii) Integrity verification is under the sole responsibility of the kernel. In this way, we keep the verification phase inside a trusted zone and, moreover, we shall no longer need to modify user-space interpreters in order to allow them to verify the scripts.

(iv) There is no need to have any private key in memory when the system is connected to the network and thus potentially vulnerable to attacks. This property is crucial otherwise developers would have to distribute the private key along with the software. Moreover, if the private key was present on the system an attacker would be able to get it and sign his own malicious code that would then be considered trusted by the system. For the two reasons discussed above, we need to use digital signatures and it is not possible to use authentication algorithms that need the private key for verification.

(v) Impact on users and administrators is minimal. This is a general requirement for all security architectures as otherwise users will not cooperate and administrators will be reluctant to adopt the architecture. We observe that our proposal is completely transparent to users and only requires administrators to manage the list of trusted public keys that are used to verify the signatures.

4.2. Implementation strategy

When a file is executed, the kernel loads it in memory and reads the magic number from the file. This number specifies the format of the invoked executable.

Using the magic number, the kernel looks for the appropriate handler, and (if available) executes it.

The handler verifies the integrity of the current file. If dynamic parts (e.g. shared objects) are present, they are verified before they are merged in the process image.

We discuss, as an example, what happens when a signed script is executed. On the basis of the script's magic number (i.e. the sequence '#!' at the beginning of the file), the kernel executes the script handler that extracts the pathname of the interpreter and then runs it providing, as a command-line argument, the invoked script file. The interpreter might be, for example, an ELF binary and so it will be independently verified by its own handler. In our approach, the verification of a script file is performed by the kernel script handler that accomplishes it by invoking the `verify` procedure of the format. In Figure 2, we describe the steps of the verification of a shell script in our architecture and in the one proposed by van Doorn *et al*. [25].

van Doorn *et al.*	This paper
`user runs the bar shell script`	
`the kernel loader searches for the script handler`	
	`script handler verifies bar`
`the kernel loads, verifies and runs:`	
`(a modified) /bin/sh`	`the standard /bin/sh`
`/bin/sh loads and verifies bar`	`/bin/sh loads bar`
`/bin/sh executes bar`	

FIGURE 2. Executing signed script `bar` in the two architectures.

The main difference is that in our architecture the verification is performed by the format handler within the kernel. Instead in the architecture of van Doorn *et al*. [25] the verification is performed by the interpreter (i.e. `sh` in this case) that must be modified.

Working at the kernel level allows us to provide a catch-all solution for executable formats (provided that they are understood by the kernel) with the two notable exceptions discussed previously. Instead in the approach of [25], it is necessary to modify one-by-one all the interpreters in order to add verification capabilities. This is a gigantic task as there are several interpreters to be modified (e.g. `ld.so` for ELF, shell interpreters and interpreters for script languages like Perl) and each is a very complex object.

The execution of dynamic libraries is accomplished in the same way: the kernel handler extracts pathnames of each dynamic library used by the application, then it verifies each library. If all verifications succeed, the process execution is allowed (this procedure can be expensive and thus we make the verification process more efficient using a caching mechanism).

In Figure 3, we compare the verification steps for ELF binaries performed by our scheme with the one of van Doorn *et al*. [25]. Note that in our scheme the verification step for both executables and libraries is performed by our handler and then the normal execution continues while in van Doorn's scheme the verification process is performed by both the kernel that verifies the executable and a modified `ld.so` that verifies the referenced dynamic libraries.

As we can see, in our architecture, the signature verification is always performed by the kernel, thus interpreters need no modifications.

van Doorn *et al.*	This paper
user runs the foo ELF binary	
the kernel loader searches for the ELF handler	
ELF handler verifies foo	
ELF handler loads, verifies and runs (a modified) ld.so	ELF handler extracts pathnames of dynamic parts of foo (including ld.so) and verifies them
ld.so extracts pathnames of dynamic parts of foo and verifies them	
foo is executed	

FIGURE 3. Executing signed ELF `foo` in the two architectures.

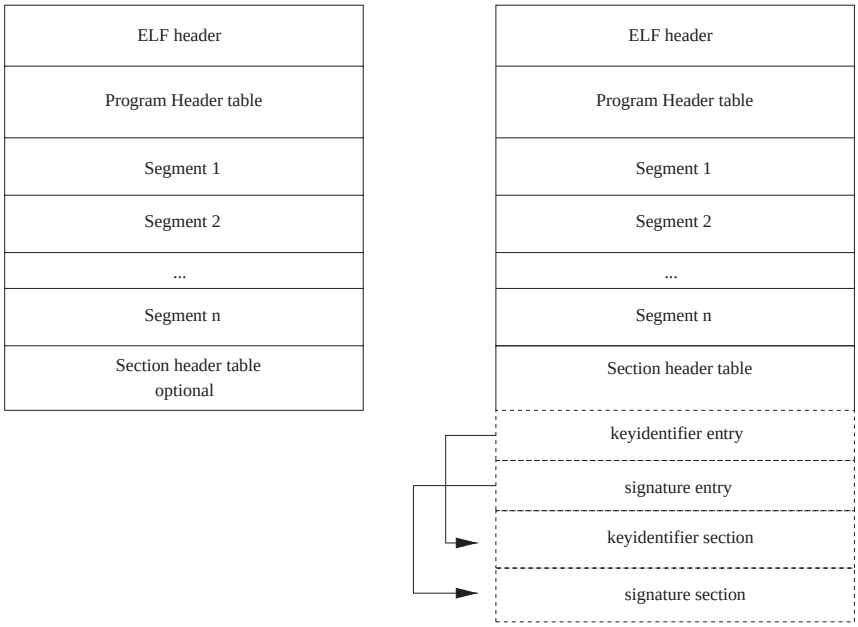

FIGURE 4. The format of unsigned and signed ELF files.

5. IMPLEMENTATION FOR LINUX

In this section, we discuss some aspects of our implementation for the Linux operating system. The goal of our experimental work was to provide an implementation for the case of ELF binaries and scripts as a proof of concept. Obviously, our implementation can be easily accommodated with other executable formats (e.g. COFF, a.out). Our implementation can be divided in two parts. First, we modified the Linux kernel for the two formats in order to add the verification capabilities. Second, we developed the utilities (`wlfsign` and `scriptsign`) to add signatures to the executables. Our prototype following the implementation choices of [25] uses the RSAREF [31] library and the PKCS7 [32] format for computing and encoding digital signatures.

5.1. The ELF format

The ELF [26, 27] format was developed to provide a binary interface that is operating system independent. Three types of ELF files have been identified:

(i) *relocatable files* that can be linked to have an executable or a shared object file (these are the `.o` files);
(ii) *shared object files* that can be used by the dynamic linker to create a process image (these are the `.so` files);
(iii) *executable files* that hold code and data suitable for the execution.

An ELF file starts with the ELF header, which is followed by the program header table, the segments and the optional section header table (see Figure 4).

The ELF header describes the organization of the file and its fields, specifies the offsets of the program header table and of the section header table, the size of an entry of each header table and the number of entries in each header table. The program header table has the information needed to locate the segments of the file that contain data

```
struct linux_binfmt {
        struct linux_binfmt *next;
        long *use_count;
        int (*load_binary)(struct linux_binprm *, struct  pt_regs *);
        int (*load_shlib)(int fd);
        int (*core_dump)(long signr, struct pt_regs *);
};
```

FIGURE 5. The `linux_binfmt` structure.

required to create a process image. The program header has an entry for each segment that specifies, among other information, the location and the type of the segment. The segment type specifies the purpose of the segment and how it can be used by the loader. For the aim of this brief overview, we are interested in load, dynamic and interpreter segments. Load segments are mapped to the process memory image. They include text of executable files, data and so on. Dynamic segments include references to other ELF files that will be loaded into the process image; additional ELF files could be loaded at run time by using either the `uselib` system call or the `dlopen` library function. The interpreter segment (it appears only once) contains the reference to an interpreter, which is a program that allows ELF executable files to load shared objects dynamically at run time. The operating system retrieves the interpreter and merges it with the process image. Since the interpreter is loaded into a new process image, it is executed in user-space.

On the one hand, the section header table has the information required to locate the sections of the file that are used for linking. The section header table has an entry for each section that specifies the offset, the size of the section and the type of section. The program header table is required for executable and shared object files and is optional for relocatable files. On the other hand, the section header table is required for relocatable objects and is optional for executables and shared objects. Moreover, by inspecting executable files generated using the `gcc` compiler, we noticed that executable files always have a section header which is always found at the end of the file.

5.2. The prototype

Execution of processes is probably the main feature of an operating system. In the Linux system, this feature is performed by the `execve` system call.

The Linux kernel has the ability of executing files of different binary formats [33]. Actually, the adjective binary is misleading as some binary formats are not binary (e.g. scripts are text files). Each binary format is associated with a magic number that is found in the first few bytes of a file. For example, scripts have magic number equal to '#!'.

We crucially rely upon on a feature of the Linux kernel that allows the registering of new handlers for binary formats. A binary format is registered by executing the `register_binfmt` procedure passing as argument the `linux_binfmt` struct (see [26] and [33] for details) filled with pointers to the functions that are going to handle the format.

Although it is possible to register a new binary format at run time by providing an implementation in a loadable kernel module, we strongly discourage this practice (for details, see Section 5.2 on the security issues regarding loadable kernel modules) and we suggest statically linking this feature into the kernel.

The Linux kernel maintains a linked list of `linux_binfmt` structures, one structure for each binary format understood by the kernel. Each structure contains, among other fields (see Figure 5), pointers to functions called `load_binary`, `load_shlib` and `core_dump`. The function `load_binary` is used to load and execute the binary file, while the function `load_shlib` is invoked to load dynamic libraries. The third function `core_dump` is invoked in case the execution aborts and an image of the process is to be created. The execution of a file is performed by the `do_execve` function that can be found in the file `fs/exec.c` of the Linux kernel source tree.

The work of `do_execve` is very simple: on input the pathname of an executable, the arguments and the environment in which the file is to be executed, the function `do_execve` scans the list format. For each format found `do_execve` invokes the `load_binary` function until one is found that is able to run the executable.

We concentrated on the ELF format for binary files and on the kernel script handler. In order to have a manageable experimental framework where signed and unsigned executables can coexist, we introduced two new executable format handlers: the Worldwide Loadable Format (WLF) to be used instead of the ELF handler, and the Signed Scripts (SSCRIPT) handler for our signed scripts. It is understood that all our experiments could have been conducted by extending the original formats but then the whole set of executables present on the system would have to be signed.

For example, we now take a brief look at how the Linux kernel executes scripts. All the scripts start with sequence '`#!`', followed by the pathname of the interpreter to be used. When the user runs an executable script, he issues a command line that contains the script name as first argument followed by a (possibly empty) list of arguments. The kernel gets the user's command line and searches the file indicated as first argument for the magic number and then it invokes the script handler. The handler extracts the

interpreter name from the file and re-builds the command line by adding at the beginning the pathname of the interpreter that becomes the first argument of the new command line. For example, suppose that the user typed the line `script arg1`, and suppose that `script` is a Bourne shell script, then the resulting command line is `/bin/sh script arg1`. Eventually, the kernel recursively executes the newly created command line.

In order to experiment with verifying handlers, we provide a suite of simple tools to sign executables and to generate keys. The `wlfsign` application makes a WLF file from an ELF executable or a shared library file. The `scriptsign` works analogously with the script files. The `wlfgenkey` application randomly generates a pair of keys and stores them in the selected media.

5.3. Signing and verifying WLF files

Our technique to sign ELF files is quite similar to the one proposed in [25]. We are interested only in the formatting of an ELF file from an execution point of view and thus we only consider the segments of an ELF file and not the sections.

When an ELF file has to be signed, the `wlfsign` utility (that has been provided with the proper key pair) computes a digest (using the MD5) of each file segment, and then computes a digital signature of their concatenation. The identifier of the public key (its MD5 representation) and the computed signature are stored in two new ELF sections: the `identifier section` and the `signature section`. We will refer to the resulting signed executable as a WLF file. In more detail, the `wlfsign` tool extracts the ELF header and fills an `Elf32_Ehdr` structure. We only set three fields for the newly added sections: the field `sh_type` carrying information about the type of the section to `SHT_NULL` to denote a section to be ignored, the field `sh_offset` that will point to the offset at which the section starts and the field `sh_size` that will contain the length of the section in bytes. The Section Header Table of the ELF file is extended by `wlfsign` with two new entries: the identifier entry pointing to the identifier section that contains the hash of the public key and the signature entry pointing to the signature section that contains the digital signature. The signature is encoded in the PKCS7 [32] format. The digital signature is computed simply by computing digests of each file segment, and then the signature algorithm (RSA, in our case) on the concatenation of the digests is run. Our signing tool can append a digital signature to any ELF file, including dynamic libraries represented by shared object files (i.e. the ones with extension `.so`). Besides `wlfsign` that actually computes the signature, we have implemented the following utilities for the management of signed ELF files:

(i) `wlfread`: this tool takes as input the name of a signed ELF file and outputs the corresponding signature, signer identifier and the referred shared objects.
(ii) `wlfso`: this tool takes as input a signed ELF file and outputs all shared objects required for the execution of the file.
(iii) `wlfverify`: this tool takes as input the name of a file in which a public key is encoded and the name of a signed ELF file, verifies in the signed ELF file the signer identifier and the signature with respect to the given public key and outputs a message based on the result of the verification.

5.3.1. The WLF handler

As we stated in Section 4, in order to preserve the generality of our system, we chose not to modify the `ld.so` interpreter and thus we have developed a handler for WLF executables.

The handler for the WLF binary format specifies a new `load_binary` function and a new `load_shlib` function and uses the same `core_dump` function as ELF.

The `load_binary` function for WLF checks if the file has the correct magic number (a WLF magic number has been defined), then it asks the cache management module for the `isTrusted` attribute. If `isTrusted` is true then the verification steps can be skipped else the signature is verified and in case of success the cache management module is invoked in order to set the `isTrusted` attribute. The verification of the signature is performed by searching for the segments that contain the digital signature and the key identifier. Then for each shared object referred to in a `Dynamic` segment of the executable file the following procedure is executed:

(i) a cached copy of the shared object is searched for by using the cache management module; if it is already in cache then the procedure successfully ends;
(ii) for each shared object referred to in the `Dynamic` segment inside the current shared object the procedure is recursively executed; if the procedure fails for one of the referred shared objects then the current procedure fails too; and
(iii) the signature verification process of the shared object is performed; if the verification fails the procedure fails, otherwise the procedure ends with success.

The procedure described above might enter into a deadlock if two shared objects have an entry in the `Dynamic` segment that refers each to the other. We have fixed this problem by using a stack that stores all shared objects referred to during the recursion. When a shared object is in the stack its verification is not performed.

Then the `load_binary` of the WLF handler loads the shared object interpreter specified by the `INTERP` segment (typically, `ld.so`) and checks recursively the executable.

We now point out a potential security vulnerability of the Linux kernel.

5.3.2. Choosing dynamic libraries at run time

Besides the shared libraries specified in the executable, the Linux kernel allows a process to dynamically load a shared library during the execution. One way of doing this is to invoke the `uselib` system call specifying the path of the shared library to be included. The `uselib` system call in turns invokes the function `load_shlib` for the handler for

the specific format of the shared library. If the shared library is a WLF library then the verification of the signature is performed. In particular, the WLF `load_shlib` function is called and we added the verification at the beginning of this function.

A second way involves the use of the C-library function `dlopen`. This function uses the `ld.so` interpreter to load the library which is then directly memory-mapped. In this way, our checking of signatures is bypassed and malicious code could be executed. We stress that most of the work of loading a shared object by `dlopen` is done at the user level and thus escapes the checking of the kernel as opposed to `uselib`, which guarantees that the loading is performed by the kernel. At the same time, we observe that the `dlopen` mechanism is the one mostly used by applications needing shared libraries and thus it is an important open problem to extend our architecture to deal also with `dlopen`. One easy way would be to patch `ld.so` but we would like to see a more general approach.

As is clear from the discussion above, in our architecture the verification of the binaries is performed at the operating system level. This has the advantage of making our approach completely transparent to the application developer.

EXAMPLE. We now give an example of how to create and install a signed executable, starting from a standard Linux distribution. We stress that this example is given only for experimental purposes since the goal of our architecture is the realization of trusted Linux distribution that has to be installed from scratch along with a static kernel. Instead our example explains how to add one signed executable to an existing Linux system and is to be considered only for didactic purposes.

First, it is necessary to generate the RSA key pair and for this purpose, we run the command `wlfgenkey` as follows:

```
#->wlfgenkey wlftest 1024 sdfjdhsfdjshfdsjk
```

by passing on the command line the length of the key and some random data to be used as seed for randomization. The output of this command is a file `wlftest.sbk` that holds the RSA private key and a file `wlftest.pbk` that holds the corresponding RSA public key. Both keys are encoded in PEM format and thus the generated files can be inspected with standard commands as `more, less, cat`. The next step is the creation of a device that works on a key repository. This can be accomplished by the following command:

```
#->mknod /dev/wlfkeyrepository c 101 0
```

The values `101` and `0` correspond to the major and minor number of a device whose management is performed by our key-handler module. Even though we suggest using a static kernel containing our modules, in our experimental setting we insert at run time our modules in the standard modular kernel. The key-handler module has to be inserted in the kernel by the superuser as follows:

```
#->insmod wlf_bkm_module.o
```

The tool `putkey` can be used by the superuser to add a previously generated RSA public key to the key repository as follows:

```
#->putkey /dev/wlfkeyrepository wlftest.pbk
```

For the sake of the example we only sign the `ls` executable. To allow co-existence of signed and unsigned `ls` executables we make a copy of `ls` to be signed.

```
#->cp /bin/ls /bin/wlfls
```

The same has to be performed for the shared objects required for the execution of `/bin/wlfls`; in particular, the references in the executable `/bin/wlfls` have to be changed so that they specify the trusted shared objects. We have embedded this feature in the `wlfsign` tool that could be run as follows:

```
#->wlfsign wlftest.sbk wlftest.pbk /bin/wlfls
```

The file `wlftest.sbk` is used to compute and append the signature while the file `wlftest.pbk` is used to compute and append the key identifier.

The signed `/bin/wlfls` ELF file can now be verified and inspected by running the following commands:

```
#->wlfverify wlftest.pbk /bin/wlfls
That's a WLF file!
The signature is good.
#->wlfread /bin/wlfls
That's a WLF file!
Signer Identifier (PEM):
   StzF6yjn77QCVrUP0tfdig==
Digital Signature (PEM):
   gbDmQzpNQxplnHeaFoRjOG48ZaDfKUbtKIpHm
       AyBjP+/7Y+7LVMIv3Lja/M
   jXfYJypBKj+u1WK7wt7iBXrQulCynvHRJ0hil
       5VM2gAQq3B4H/RwKzuOmlb
   996CqJV/Ru6KSWzgiNh7z+7jafwdksoMg9jK7
       HuXi87JNX879vXHQ=
Shared object ==> wlftermcap.so.2
Shared object ==> wlfc.so.6
```

In particular, `wlfread` outputs the shared objects referenced to by `/bin/wlfls`, which in our case are `wlftermcap.so.2` and `wlfc.so.6`, since `wlfsign` changes the first three characters of the name of each shared object (typically 'lib') to 'wlf'. Thus, we require signed shared objects to be in the directory `/wlf`:

```
#->cp /lib/libc.so.6 /wlf/wlfc.so.6
#->cp /lib/libtermcap.so.2
        /wlf/wlftermcap.so.2
```

The shared objects have to be signed preserving the `ELF` magic number since they are managed by the standard shared object interpreter `ld.so` that supports only the `ELF` magic number and we do not want to modify the interpreter.

```
#->wlfsign wlftest.sbk wlftest.pbk
      /wlf/wlftermcap.so.2 -ELF
#->wlfsign wlftest.sbk wlftest.pbk
      /wlf/wlfc.so.6 -ELF
```

We now check for other shared objects that could be recursively referenced:

```
#->wso /bin/wlfls
Shared Library: wlftermcap.so.2
Shared Library: wlfc.so.6
Shared Library: wlflinux.so.2  NOT FOUND
```

The tool says that the shared object interpreter `wlflinux.so.2` has not been found. Thus, we fix this with the following commands:

```
#->cp /lib/ld-linux.so.2 /wlf/wlflinux.so.2
#->wlfsign wlftest.sbk wlftest.pbk
      /wlf/wlflinux.so.2 -ELF
```

For the execution of a signed ELF file it is necessary to insert in the kernel the corresponding handler:

```
#->insmod wlf.o
```

The verification of signed ELF files can be traced by monitoring `/var/log/messages`. Before the execution of `/bin/wlfls` the environment variable `LD_LIBRARY_PATH` must include the directory `/wlf`. Finally, the signed `ls` command can be successfully executed:

```
#->/bin/wlfls
```

5.4. Signing and verifying script files

A script file is a text file that contains instructions that are executed by an interpreter. In the implementation of our prototype, we provide a tool that extends a script file with a digital signature formatted according to PKCS7 [32]. The PKCS suite of standards specifies data and message formats for various cryptographic primitives and protocols. The PKCS7 deals with digital signatures and digital envelopes.

The signature is then codified in PEM [34] (a popular standard for encoding a binary string into a printable form) and is added at the end of the file. The code appended to the original file is not executed by the script interpreter because we add it as a comment for script interpreters. The script handler performs the verification phase before running the script interpreter that is referred into the script (i.e. the command line that follows the magic number '#!').

The `scriptsign` application changes the script magic number to '#@' and adds, at the end of the script file, a comment containing the identifier of the signer and the signature encoded in PEM format.

The SSCRIPT handler takes care of executing signed scripts whose magic number is the sequence '#@'. The command line parsing process is the same as in the usual scripts but in the last step, the SSCRIPT handler verifies the signature of the script before running the new command line. It is clear that during the new execution cycle, the interpreter will be verified too.

5.5. Cache management

The execution of the verification steps each time that the execution of a file is requested could have a severe impact on the performance of the system. Repeated verification of the same executable, within brief periods, can be avoided if there is a guarantee that the file has not been changed since the previous verification. To do so, the file should be kept 'safe' after it has been validated.

Our implementation of this idea is quite simple. Each executable, once it is successfully validated, is copied in a memory cache. Since the cache is assumed to be a 'safe place', cached files need not be verified again when they are invoked, provided that they could be read from the cache itself. To do so, we bind each cache entry (i.e. each copy of a verified executable) to a device on the file-system. Each access to a cached file (including the ones by the kernel) is redirected to the corresponding device instead of the inode of the original file. Each cached file is actually stored in kernel memory and thus can only be updated by the kernel (see discussion on the assumptions in Section 2.4). The main advantage of this caching mechanism is that it covers both local and remote files, although it is quite expensive in terms of memory usage.

In order to integrate such a mechanism in the infrastructure discussed above we follow the following strategy:

- when the execution process of an executable file begins, the cache is checked;
- if the file is already in the cache then the execution process continues using the cached file with the `isTrusted` attribute set to 0;
- else the file is inserted in the cache and then the execution continues using the cached file.

We have designed and implemented a kernel module that manages the cache and provides the `fetch` function that other kernel modules can use in order to get the pathname of a cached file from the pathname of an executable file. The `fetch` function inserts a file in the cache if it is not already there.

We modified the `do_execve` function (see the file `exec.c` in the directory `fs` of the Linux kernel sources) which drives the execution of files in the Linux kernel in the following way. We add a call to the `fetch` function which returns the pathname corresponding to the cached version of the file to be executed. Next, `do_execve` calls the `search_binary_format` function to identify the right handler for the format. The format handler should then verify the integrity of the cached file and start the execution. The cache-management module attaches to each file a bit, called `isTrusted`, that is set to 1 if the file has been already verified successfully. Thus if `isTrusted` $= 1$, the handler can skip the verification and proceed directly with the execution.

In Figure 6, we illustrate the different states that compose the execution phase of a file in our architecture.

5.6. Key management

Key management is an important component of our architecture. We do not need to handle any secret key as executables are assumed to be signed off-line by software distributors. However, public keys are needed in order

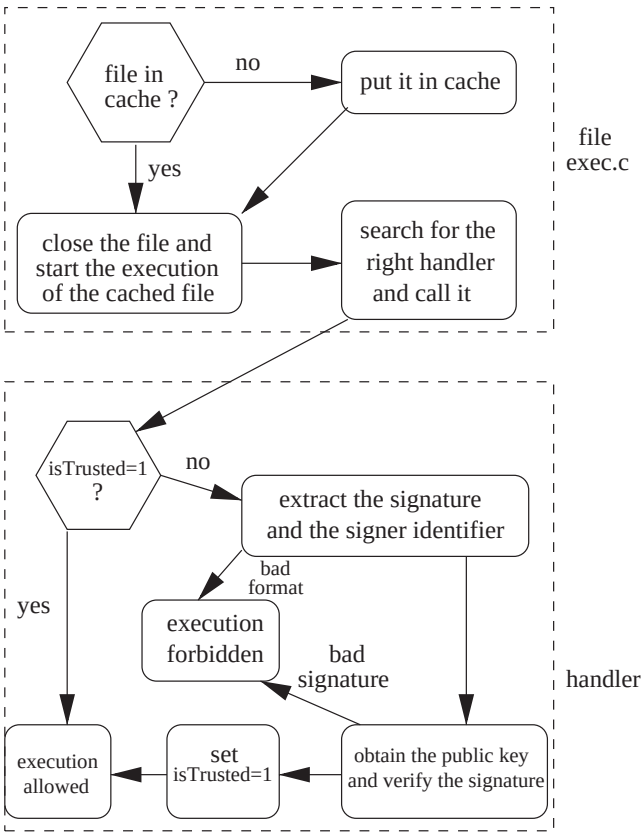

FIGURE 6. The execution of a file in our architecture.

to verify signatures. The key management subsystem is crucial for the security of the system because if the adversary manages to add his or her own keys permanently to the list of trusted keys, then malicious code can be inserted and will never be eradicated. The management of public keys has to take into account two points. First, executables signed by different providers (e.g. with different private keys) could be present on the system. Second, public keys might be stored on different supports (e.g. smart cards, CD-ROM).

We take a look at a typical scenario as an example. Our system runs the 'FOO' Linux distribution whose executables are signed with the private key of FOO, Inc. This corporation distributes its software products with the appropriate public key, stored on a smart card. Furthermore, our system administrator has also installed software produced by BAR Corp. (which distributes its public keys on a CD-ROM) and moreover, some self-produced tools. Thus, our system needs to manage a list of public keys. It should be able to choose the correct public key for verifying a signature.

Now, we will show how our architecture addresses the needs raised by this common scenario.

To allow maximum flexibility, we have abstracted the functionality of the key management into a different kernel module. In this way, different key management schemes can be implemented without affecting the handlers. Each key pair is provided with a key-id that identifies it. The key-id is included with the signature and is used by the format handler in order to ask the key management for the proper public key. Public keys have to be loaded into the kernel memory at bootstrap. In our prototype, we wrote two simple key management modules that loaded the keys either from a file, a read-only floppy disk or a CD-ROM.

Our system can also employ a hierarchical approach to key management similar to the one of X509 [24]. Instead of having one public key for each software provider we have a few trusted keys. Each software provider will have its key signed by a trusted entity (which acts as a certification authority) and then each executable will contain the identifier of the signer of the public key and the signed public key (similar to an X509 certificate).

5.7. Experimental results

The proposed architecture has been implemented providing tools for the generation of private and public keys, for the signature of ELF and script files, for listing the shared objects referenced by ELF files and for pushing public keys to a device. Modules for the kernel 2.4.17 have been implemented for the run-time integrity check of ELF binaries and shell script files, for the key management and for the cache management.

Our software uses the RSAREF library [31] that provides the required implementations of cryptographic primitives. We have chosen only RSA keys for our implementation because the signature verification is faster as we can choose low public exponents.

Experimental results show that run-time integrity check of executable code can be performed with reasonable performances and thus our architecture and implementation are a practical mechanism to combat permanent attacks.

We performed two kinds of test: one to measure the performance of the system and one to assess its robustness. The table in Figure 7 shows the results of performance measurement of running some of the largest general purpose Linux executables. As we can see, the loading of signed executables takes twice the time needed for loading the unsigned ones. The slowdown depends on both the size of the files and the number of dynamic objects that have to be loaded together with the main program. For example, the slowdown incurred by `rpm` is the smallest as `rpm` has no shared objects.

Anyway, we believe that the overhead shown is decidedly reasonable as it only affects the start of the execution process of an executable and is amortized over longer executions of the commands: the table refers to very short executions where only the version number of the program is requested. Moreover, such overheads should be considered as upper bounds, since measurements have been carried out without using the cache.

The second kind of test focuses on the robustness of the implementation. We set-up and signed all executables of an Apache Web Server. We also wrote and signed some `cgi` both in script languages (whose interpreter was also signed), and in C. The server neither crashed nor failed to answer any query.

elf binary	size	cmdline	time	sig. size	sig. time	dyn.
/bin/bash	519964	–version	0.020	520125	0.040	4
/usr/bin/perl	708188	–version	0.020	708349	0.050	6
/bin/rpm	1580104	–version	0.040	1580264	0.050	0
/usr/bin/gdb	1779900	–version	0.050	1780061	0.090	5

FIGURE 7. Performance of signed ELF executable files. Execution times are in seconds and have been measured on a VMWare virtual machine built on top of a Pentium IV 1.7 GHz with 256 MB RAM and kernel 2.4.17.

6. THE OLD-VERSION ATTACK

We now discuss a possible attack to our architecture. Suppose that an adversary discovers some weaknesses of some executables that are installed (and thus signed). The adversary may keep one copy of the old signed executables. Whenever the administrator of the victim machine upgrades that software, the adversary could replace the new executables with the old ones, which still have a valid signature, and that still successfully pass the verification phase. In order to make this kind of attack ineffective, which we call the old-version attack, we propose two different schemes. The first one is based on the use of file revocation lists and requires a read-only support that is able to store a few megabytes (a smart-card cannot be used while a CD-ROM can) while the second one is based on some more trusted information maintained in the file-system that must be updated by using a private key (the system must be disconnected from the network when such a task is performed) each time new files are installed in the system.

Note that this problem in its generality cannot be handled by a simple key revocation mechanism. Consider the example of a Linux distribution. It is thinkable, though impractical, that each version of the distribution is signed with a different key. Thus each time a new version is installed the key of the old version is revoked using standard key revocation schemes. Instead, we can consider the case in which specific packages (i.e. those that have security problems) are upgraded.

6.1. File revocation lists

The old-version attack is similar to the case of a digital certificate whose corresponding private key has been discovered and thus the certificate must be revoked. The list of revoked certificates is then released by a trusted authority that appends a digital signature to the list. Each time a party receives a digital certificate, the verification phase involves the checking of the certificate with respect to the list of revoked certificates.

We can introduce in our architecture the same mechanism illustrated above by adding to the repository of public keys the corresponding file revocation lists released by the software distributors. In order to avoid the execution of revoked files, each format handler should first check the file with respect to the files revocation lists and then proceed with the verification steps. The check can be performed efficiently in the following way.

- A serial number can be added to each file that contains executable code.
- The list of revoked files can be implemented with a data structure that guarantees efficient lookups (e.g. hash table).

When a revoked file is installed on the system and someone tries to execute it, the appropriate handler extracts from the file the serial number and performs a lookup with the data structure that holds the revoked files. Only in the case where the file is not revoked the verification task proceeds. If the intruder tries to change the serial number in order to pass the previous test then the signature is not valid any more and the execution will be forbidden by the signature verification step.

6.2. The version tree

In this solution, we need to store some information in the root directory and in each directory in which signed executables and shared objects are placed. This solution has been inspired by [14] although our scheme is considerably simpler.

First, we describe new information we have to store in the file-system and into the file signature. A signed segment that contains the file version string (that could be a serial number, the hash of the whole file, etc.) is added to each executable. We place a special file called directory version record in each directory which contains executables or shared objects (e.g. `/bin`, `/usr/lib` and every other directory in `$PATH, $LD_PATH` or reported into the `ld.so.conf` file). This record contains a copy of version strings of all files contained by the directory, and a directory version string that is the hash of its list of file version strings. The directory version record is signed. All the directory version records are reported into a root version record that is formed like a directory version record, thus we also have a root version string. A copy of the root version string is stored in the public key repository (the smart card, the ROM BIOS, etc.). The root version record is signed. Figure 8 shows the layout of this structure.

In order to install a new package, the administrator follows the usual scheme we introduced in Section 4, then he or she extracts the file version strings from new executables and updates: the directory version record, the root version record and the root version string on the key repository. Note that the administrator has to sign each structure he or she modifies, and he or she does that when the system is in secure mode so private keys are not exposed.

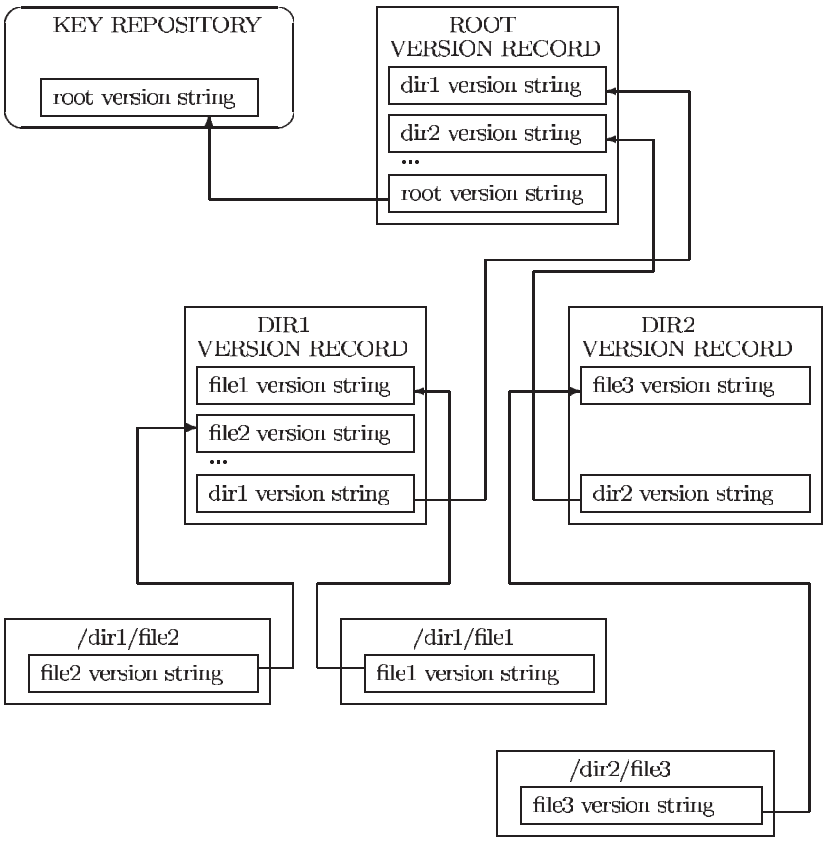

FIGURE 8. The tree of version records.

At boot, the kernel loads the root version record and verifies the integrity of the root version string. Each time the kernel runs an executable, it first loads the related directory version record and verifies both the signature and the validity of the directory version string. Finally, the kernel loads the executable, verifies its signature and the file version string. If everything works, the kernel runs the invoked application. Otherwise, if an error occurs at any phase, the execution fails (e.g. if any record is missing, or a version string mismatch occurs at any level).

Since the number of directories in the `$PATH` and `$LD_PATH` is usually small, and the related directory record never changes while the system is working in normal mode, all record files can be easily cached. Moreover, once an executable has been cached, no more verifications are needed until its entry expires. So, this scheme does not heavily affect the system performances, although it lengthens the installation process.

7. ABOUT EXECUTABLE HANDLING

The user's perception of executables, in a UNIX-like system, actually wraps different objects. Binaries and scripts are just invoked in the same way but the system handles them quite differently and, moreover, many differences exist also among execution processes of different binary formats. Even the same binary format can be handled differently according to the design of the program. For this reason, the design of a general mechanism of verification at run-time of executables is a non-trivial task and probably it is not currently possible. We think that in a 'pure' model, integrity verification is simply a phase of execution. Thus, first, an entity that performs the verification has to be always present and second, verification that has to be performed on anything is handled as an 'executable'.

The approach proposed in [25] is a workable solution but breaks that model. Introducing the `verify()` system call implies that developers can choose not to verify and, moreover, delegating verification to user-space interpreters, like `ld.so` or `bash`, implies that 'sometime' the system does not provide any verification.

On the other hand, the Linux operating system implements the `exec` chain of the different executable format handlers by means of a sort of object-oriented interface, quite complete and general, but the execution chain of some kinds of executables breaks that model. This is the case of loading ELF dynamic objects by means of the `dlopen()` library function, that bypasses the kernel handler and, thus, the verification step. Currently, the `dlopen()`/`dlclose()` suite of functions is the most employed way of managing dynamic parts of executables in many UNIX-like systems with respect to operating system interfaces like the `uselib()` system call in Linux.

A possible way to achieve a solution that is closer to the 'pure model' is to develop a standard kernel interface for loading dynamic objects, on top of which the `dlopen()`

interface could be re-modeled. Currently, the `dlopen()` function loads the target dynamic object into the memory by means of an `mmap()` system call[1] that does not involve the ELF handler. Introducing a new system call, essentially a dedicated version of `mmap()` that loads executable code into the memory, under the control of the ELF format handler, could be a good solution in order to allow our model to catch this execution case, saving portability.

Scripts open a serious issue. In some way, script files have been considered as executables for 'historical' reasons, but actually, they should be considered as data files while script language interpreters are simple user-level applications. Handling execution of scripts and binaries in the same way is in some cases impossible. Consider, as an example, the `source` command of the `tcsh` interpreter.[2] For the shell, invoking `source` is like a 'dynamic code loading', but from the kernel point of view, this operation appears simply as a user-level application that opens a file, hence, involving the kernel in this operation, within the 'pure model', is clearly impossible.

Currently both [25] and our architecture cover this point with an *ad hoc* solution. The issue is still open.

8. CONCLUSIONS

We have presented a solution for run-time integrity check of executable code. Our architecture can be used for any type of executable file format provided that a tool for the signature and a kernel handler are available. A format independent cache management mechanism has been designed and implemented to improve the performance of the system. We provided a proof of concept implementation for ELF and script files.

Our approach is suitable for workstations that provide services to end users and not for the development of applications. In fact, the simple compilation of a program generates an executable file that can become trusted only if the author signs it and its public key is in the list of trusted public keys.

Experimental results show that our solution is very efficient but other issues need to be addressed in order to protect a workstation from attacks based on malicious code.

Attacks to our infrastructure are possible by changing the kernel during its execution; however, such attacks are not as easy as the installation of a root-kit and can only have a temporary success.

Next we point out two drawbacks of our approach and leave them as open questions. First, we observe that scripts that are invoked by passing the name of the file as a command line argument to the interpreter are not verified at all. Our architecture only guarantees that the interpreter is verified. One solution, as suggested by [25], is to modify all interpreters, thus losing the universality of the approach. Moreover, we believe that it is not an easy task to modify complex objects like the Perl interpreter and thus a better solution to this problem is needed. A second weakness of our architecture concerns dynamic libraries. In our architecture, dynamic libraries that are specified at linking time (and thus are referenced in the executable) are verified.

A preliminary version of this work appeared in [35].

[1] The `mmap()` system call maps files or devices into memory, allowing the user to state the starting address and several access permissions (e.g. read, write, execution) of the target memory buffer.

[2] The `source` command asks the shell to parse and execute the sequence of commands contained in a given file. Note that the file is not necessarily an executable, and that when `source` is invoked, the shell does not create a new process.

ACKNOWLEDGEMENTS

We would like to thank Pino Persiano. Pino's suggestions and comments improved notably this work. We also thank Marco Cesati, Louis Granboulan and the anonymous reviewers for their remarks about our work.

REFERENCES

[1] McCanne, S. and Jacobson, V. (1993) The BSD Packet Filter: a new architecture for user-level packet capture. In *Proc. 1993 Winter USENIX Conf.*, San Diego, CA, January 25–29, pp. 259–269. The USENIX Association, Berkeley, CA.

[2] Dierks, T. and Allen, C. (1999) *The TLS Protocol, version 1.0*. RFC-2246, IETF Network Working Group.

[3] Cowan, C., Wagle, P., Pu, C., Beattie, S. and Walpole, J. (2000) Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proc. DARPA Information Survivability Conf. and Expo (DISCEX)*, Hilton Head Island, SC, January 25–27, Vol. 2, pp. 119–129. IEEE Computer Society Press, Washington DC.

[4] Cowan, C. *et al*. (1998) StackGuard: automatic adaptive detection and prevention of buffer overflow attacks. In *Proc. 7th USENIX Security Conf.*, San Antonio, TX, January 26–29, pp. 63–78. The USENIX Association, Berkeley, CA.

[5] Ko, C., Fraser, T., Badger, L. and Klipatrick, D. (2000) Detecting and countering system intrusions using software wrappers. In *Proc. 9th USENIX Security Symp.*, Denver, (CO), August 14–17, pp. 145–156. The USENIX Association, Berkeley, CA.

[6] Halflife (1997) Bypassing integrity checking systems. *Phrack Magazine*, **51**, September 1997, http://www.phrack.org.

[7] Anonymous (2001) Linux on-the-fly kernel patching without LKM. *Phrack Magazine*, **58**, December 2001. http://www.phrack.org.

[8] Network Associates Inc., McAfee Security Home Page http://www.mcafee.com/.

[9] Symantec Corporation (2004) http://www.symantec.com.

[10] Kim, G. H. and Spafford, E. H. (1994) The design and implementation of Tripwire: a system integrity checker. In *Proc. Second ACM Conf. on Computer and Communications Security CCS'94*, Fairfax, VA, November 2–4, pp. 18–29. ACM Press, New York.

[11] Kim, G. H. and Spafford, E. H. (1994) Experiences with Tripwire: using integrity checkers for intrusion detection. In *Proc. 3rd USENIX Systems Administration, Networking and Security Conference*, Washington DC, April 4–8, pp. 89–101. The USENIX Association, Berkeley, CA.

[12] Cattaneo, G., Catuogno, L., Del Sorbo, A. and Persiano, G. (2001) The design and implementation of a Transparent Cryptographic File-System for Unix. In *Proc. FREENIX Track (FREENIX-01) of the 2001 USENIX Ann. Technical*

Conf., Boston, MA, June 25–30, pp. 199–212. The USENIX Association, Berkeley, CA.

[13] Fu, K., Kaashoek, M. F. and Mazieres, D. (2000) Fast and secure distributed read-only file-system. In *Proc. 4th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2002)*, San Diego, CA, October 23–25, pp. 181–194. The USENIX Association, Berkeley, CA.

[14] Mazieres, D. and Shasha, D. (2002) Building secure file-systems out of Byzantine storage. In *Proc. Twenty-First ACM Symp. on Principles of Distributed Computing (PODC 2002)*, Monterey, CA, July 21–24, pp. 108–117. ACM Press, New York.

[15] Spinellis, D. (2000) Reflection as a mechanism for software integrity verification. *ACM Trans. Inform. Syst. Security*, **3**(1), 51–62.

[16] Lie, D., Mitchell, J., Thekkath, C. A. and Horowitz, M. (2003) Specifying and verifying hardware for tamper-resistant software. In *Proc. 2003 Symp. on Security and Privacy*, Berkeley, CA, May 2003, pp. 166–177. IEEE Computer Society Press, Washington DC.

[17] Devanbu, P. T., Fong, P. W. L. and Stubblebine, S. G. (1998) Techniques for trusted software engineering. In *Proc. 20th Int. Conf. on Software Engineering (ICSE' 98)*, Kyoto, Japan, April 19–25, pp. 126–135. IEEE Computer Society Press, Washington DC.

[18] Menezes, A. J., van Oorschot, P. C. and Vanston, S. A. (1997) *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL.

[19] Lampson, B., Abadi, M., Burrows, M. and Wobber, E. (1992) Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, **10**(4), 265–310.

[20] Wobber, E., Abadi, M., Burrows, M. and Lampson, B. (1994) Authentication in the Taos operating system. *ACM Trans. Comput. Syst.*, **12**(1), 3–32.

[21] Neuman, B. C. and Ts'o, T. (1994) Kerberos: an authentication service for computer networks. *IEEE Commun.*, **32**(9), 33–38.

[22] Sun Microsystems: *JavaTM Security Evolution and Concepts*. Technical Articles. http://developer.java.sun.com.

[23] Sun Microsystems Corporation (1996) *Java Code Signing*. http://java.sun.com/security/codesign.

[24] Housley, R., Ford, W., Polk, W. and Solo, D. (2002) *Internet X509 Public Key Infrastructure: Certificate and CRL Profile*. RFC-3280, IETF Network Working Group.

[25] van Doorn, L., Ballintijn, G. and Arbaugh, W. A. (2001) *Signed Executables for Linux*. University of Maryland Technical Report CS-TR-4256, University of Maryland, College Park, MD.

[26] The SCO Group (2001) System V Application Binary Interface. http://www.caldera.com/developers/gabi, Lindon, Utah, USA.

[27] Lu, H. (1995) *ELF: From the Programmer Perspective*. http://citeseer.nj.nec.com/lu95elf.html.

[28] Stinson, D. (2002) *Cryptography: Theory and Practice* (2nd ed). Chapman and Hall.

[29] Arbaugh, W., Farber, D. and Smith, J. (1997) A secure and reliable bootstrap architecture. In *Proc. 1997 IEEE Symp. on Security and Privacy*, Oakland CA, May 1997, pp. 65–71. IEEE Computer Society Press, Washington DC.

[30] Itoi, N., Arbaugh, W. A., Pollak, S. J. and Reeves, D. M. (2001) *Personal Secure Booting*. LNCS 2119, pp. 130–144. Springer-Verlag, Berlin, Germany.

[31] RSA Laboratories (1994) *RSAREF: A Cryptographic Toolkit for Privacy-Enhanced Mail*. http://www.aus.rsa.com.

[32] RSA Laboratories (1993) *PKCS7 Cryptographic Message Syntax Standard*. ftp://www.rsasecurity.com.

[33] Bovet, D. P. and Cesati, M. (2001) *Understanding the Linux Kernel*, pp. 552–574. O'Reilly & Associates Inc., Cambridge, MA.

[34] Linn, J. (1993) *Privacy Enhancement for Internet Electronic Mail*. RFC-1421, February 1993, IETF PKIX Working Group.

[35] Catuogno, L. and Visconti, I. (2002) A format-independent architecture for run-time integrity checking of executable code. In *Proc. Third Conf. on Security in Communication Networks (SCN 02)*, Amalfi (SA), Italy, September 11–13, LNCS 2576, pp. 219–233. Springer-Verlag, Berlin, Germany.