ABSTRACT
Assuring that system files have not been tampered with over time is a vital, but oft-overlooked, aspect of system security. File integrity checkers provide ways to assure the validity of files on a system. This paper concerns itself with a review of file integrity checkers. It pays particular attention to what the minimum requirements for an integrity checker are, the different approaches taken to integrity checking, the strengths and weaknesses of each approach, major divisions in checker design, innovative or unusual features of certain checkers, and appropriate situations under which each type of checker should be used. The design of an integrity checker which combines the best features of various checkers is described and discussed.

KEY WORDS
security, integrity checking, file checking, file checker, file security

FILE INTEGRITY CHECKERS: STATE OF THE ART AND BEST PRACTICES

Yusuf M Motara, Barry Irwin

g00m2420@campus.ru.ac.za, b.irwin@ru.ac.za

ABSTRACT
Assuring that system files have not been tampered with over time is a vital, but oft-overlooked, aspect of system security. File integrity checkers provide ways to assure the validity of files on a system. This paper concerns itself with a review of file integrity checkers. It pays particular attention to what the minimum requirements for an integrity checker are, the different approaches taken to integrity checking, the strengths and weaknesses of each approach, major divisions in checker design, innovative or unusual features of certain checkers, and appropriate situations under which each type of checker should be used. The design of an integrity checker which combines the best features of various checkers is described and discussed.

KEY WORDS
security, integrity checking, file checking, file checker, file security

1 INTRODUCTION

File integrity checkers provide a way to ensure the validity and integrity of files on a system. Being able to verify that only trusted binaries are executing on the system is a first step towards building a secure system; this paper discusses various projects that have an impact on the problem and examines the problem itself. When implementation is discussed we use metaphors and language familiar to users of the Linux operating system: Linux is freely modifiable and easily obtainable, and is therefore a good choice for discussions from which a practical outcome is possible. Despite this bias, the general principles discussed below should be applicable to almost any operating system and software environment.

1.1 Background

It is initially assumed by many that monitoring file integrity is a fundamentally easy goal: little could be simpler than comparing the MD5 checksums of existing files against a known-good list of checksums. Indeed, that approach forms the core of a variety of projects, but the problem is larger than simply verifying file contents. Below is a brief list of the fundamental qualities, influenced by the seminal work [7], that all integrity checkers should have.

Meta-information checks File integrity is not only about the file contents: it is also about the meta-information of the file, such as owner, timestamp and permissions. A "trusted" non-SUID executable may very well be an untrusted SUID executable.

Automation An integrity checker should not depend on being run by the user; indeed, the apathy of even technically-minded users when it comes to security

The assistance of the Deutscher Akademischer Austausch Dienst (DAAD) and the National Research Foundation (NRF) is gratefully appreciated and acknowledged. The helpful comments and suggestions of the anonymous reviewers have also been of great assistance in creating the final draft of this document, and we extend to them our thanks.

matters is a cause of concern. In the same vein, a tool that requires user input before action is taken is essentially useless until a systems administrator intervenes – leading to a time lapse between detection and action which may well be exploitable. Examples of this are the Code Red, Slammer, Nachi/Blaster and Sasser worm epidemics which would never have taken place had systems administrators applied the patch that had been readily available for a number of days[2]. The perception seems to be that security is a non-essential operation, and that running security checks may be skipped if one is short on time or simply forgetful.

Relevance The problem of automation is taken a step further if the tool used provides copious amounts of output for a human to sift through. Once again, this places the burden of processing on the human – and whilst computers rarely make mistakes, humans are prone to do so. Missing a detail in a mass of irrelevant information is easy. Any file integrity checker that insists that a human must go through output every day, or every week, seems to be inherently flawed.

Self-protection If files on the system are being modified, it both is prudent and reasonable to assume that an attacker has gained privileges that he should not have. In such a case alteration of an unprotected integrity checker database or other crucial files is trivial, and it may be that the checker then does more harm than good in providing the false security of assuring the administrator that all is well when all is not very well at all!

Continuous checking An integrity checker that only spots untrusted binaries after they have potentially been executed is less useful than one which can detect an untrusted binary before it has been executed: the primary difference is that the latter reduces the chance of damage being done to the system. In fact, the latter may prevent a system compromise entirely in the case of an executable needing to be run in order to breach security and gain unauthorized capabilities.
Related to this point is the fact that checking files periodically leads to an "opportunity gap" for an attacker, who has the amount of time between checks to do as he would like on the system with no fear of detection. Importantly, this can lead to the compromise of related systems in a networked environment, after which detecting and fixing the damage caused on the original system still leaves the network vulnerable to outside influence.

Upgradeable As new vulnerabilities in programs are discovered, or as new versions of a program come out, it should be easy to make the integrity checker recognize the new version as valid *and* recognize the old version as invalid. A failure to do the former can lead to a denial of service as trusted executables are not allowed to execute, and a failure to do the latter leads to a situation in which a new version of a program may be replaced by an old (and potentially flawed) version, leading to an exploitable machine. Both situations represent a failure of the integrity checker.
Ideally, upgrading the system should require little or no user interaction.

Catuogno and Visconti in [5] differentiate between a "strong" form of intrusion and a "weak" form, with the former being an intrusion that is able to "colonize" the system, and the latter being an intrusion that is unable to do so. Given the above qualities, a file integrity checker is able to consistently reduce a "strong" intrusion to a "weak" intrusion attack. In the absence of the ability to permanently take over the system, then, an attacker must resort to continually re-exploiting weaknesses in the system and must also deal with system administrators who have been alerted to his attempts and are now prepared and waiting for him to try again.

2 EXISTING WORK

Much of the existing work in this area overlaps in terms of features or implementation. For ease of reference, therefore, existing work has been separated into a categories based on where integrity checking takes place. In the case of each category, certain implementations are singled out as exemplary and examined more closely; other works that fall into the same category are mentioned for the sake of completeness. Note that a work may be classified under more than a single heading.

2.1 Userspace Checkers

Integrity checkers that operate in userspace are generally more vulnerable than checkers which operate in kernelspace; this is because they are more susceptible to modification by users, and are only protected by the same system mechanisms that protect the rest of the system. Since we are working under the assumption that the system is compromised (for this is the primary situation in which a file integrity checker proves its usefulness), we must also assume that certain protections that the system has laid out will no longer be effective. Most integrity checkers based in userspace therefore recommend that the checker binary and any data files be kept on read-only media to ensure their validity. This precaution does little to ease upgrading.

Another disadvantage of a userspace-based solution is that continuous checking (see 1.1) becomes more difficult. Without trapping calls in the kernel, or without the system providing a means for a userspace program to be notified of each execution, a userspace program is unable to continuously check files before they are executed. Furthermore, periodically checking the entire filesystem instead of simply certain files on it as they are used imposes a computational penalty every time that that a check is carried out; in fact, the more files there are to be checked, the greater the penalty.

An advantage of an integrity checker in userspace is that it is far easier to implement and maintain than a non-userspace solution. This is due to the variety of libraries and resources that it has easy access to, as well as the high-level constructs that a programmer may use during implementation. A userspace solution may also be more flexible and configurable; it may easily take data from a variety of sources (e.g., the filesystem, user input, a network device, etc) whereas there are far fewer (and far less convenient) methods for communicating with kernelspace, such as the use of ioctls, “special” filesystems (/proc and /sys on Linux being examples) and sysctls.

2.1.1 Case Study: Tripwire

Tripwire monitors “key attributes” of files (such as a hash of file contents, number of links, permissions, and file size) and notifies the user if a file has been changed[12]. It is cross-platform, operating on both Unix and Windows platforms. After taking a “snapshot” of the system, the existing system is checked periodically against this snapshot and differences are noted. This means that Tripwire checks all files, not simply binary files, against tampering. If a change is made to a system, a new “snapshot” must be taken and used. Tripwire cannot tell the difference between a new and an old snapshot.

2.1.2 Summary

Other userspace file integrity checkers are YAFIC[1], AIDE[2], afick[3], Nabou[4], Integrit[5], Ionx Data Sentinel[6], and Xintegrity[7]. An interesting feature employed by some is signed databases (to ensure database integrity); aside from that innovation, they largely resemble Tripwire.

As mentioned, userspace file integrity checker is reasonably easy to implement; it may be that this is the reason for such a plethora of userspace tools to exist. However an integrity checker is easy to implement but difficult to implement correctly: most of the above-mentioned tools use a Tripwire-like approach and are content to periodically check the integrity of files; this leaves systems absolutely unprotected in the interval between checks. Whilst most periodic file integrity checkers do allow the time at which checks take place to be set flexibly, we would contend that this does not do more than ameliorate the "opportunity gap" problem (see 1.1: even if one had heuristics in place to test only when, for example, "suspicious" file activity was taking place, the check would still take place too late to stop file execution. In the case of simple periodic checking, the situation is even more dire: it is exceedingly unlikely for one to be faced with an attacker polite or unfortunate enough to launch an attack on a system just before, or during, a system check!

2.2 Non-Resident checkers

A non-resident integrity checker is defined as one that checks file integrity on one or more machines ("hosts") from another machine ("server"). If we assume that the hosts have no way of influencing anything happening on the server, we can easily see that is is an extremely secure approach – the most secure approach of all those mentioned in this paper.

A non-resident integrity checker faces three problems that are not as applicable to integrity checkers that are resident on a machine. The first of these questions asks which action should be taken by the host if the server cannot (for any reason) be contacted. Should the host continue nonetheless, or should a break-in be suspected – or should the client resort to a standalone mode of operation? All of these options have drawbacks, and it is difficult to reliably choose one to work with. The second question is concerns the "opportunity gap" mentioned in 1.1: how often should checks be done? Considering that networked communication is generally far too slow to make real-time integrity checking a reality, checks can only be performed at intervals. This leads to an opportunity gap that may be exploited by an attacker. The last question to be answered is that of securing the client program that runs on the host: how is it to be done? If the client program is compromised, the server may start to receive false information and continue to believe that a compromise of the host has not occurred; this problem devolves into one of ensuring the integrity of an executable on a single host, and is therefore covered by 2.1 and 2.3.

2.2.1 Case Study: Osiris[8]

Osiris positions itself as a "Host Integrity Monitoring System". Osiris never alters the initially-created "trusted" database, as do other tools; instead, it depends on the administrator to make such changes by design. This makes changing large

[1]YAFIC homepage `http://www.saddi.com/software/yafic/`
[2]AIDE homepage `http://sourceforge.net/projects/aide`
[3]afick homepage `http://afick.sourceforge.net/`
[4]Nabou homepage `http://www.daemon.de/Nabou`
[5]Integrit homepage `http://integrit.sourceforge.net/`
[6]Data Sentinel homepage `http://www.ionx.co.uk/html/products/data_sentinel/`
[7]Xintegrity homepage `http://www.xintegrity.com/`
[8]Osiris homepage `http://osiris.shmoo.com/`

parts of the system tedious, but also helps to ensure that any changes made are deliberate. The configuration file language is flexible and intuitive, and a database of file attributes is kept on a central server instead of a host machine. Communication between machines happens in a secure and scalable fashion using a client program on each host and a server program on the central database machine. Osiris uses notification filters to selectively output only important information.

2.2.2 Case Study: Radmind[9]

Radmind, like Osiris, is a system consisting of a server and multiple clients, each of which runs a client daemon. An interesting feature of Radmind is the concept of overlayed “loadsets”, each of which describes a set of files. Files that should be managed (or ignored) on client machines are specified via “transcripts”. The Radmind client occasionally sends a summary of filesystem data to the server, which then fixes any changes and sends replacement files back to the client. Network traffic may optionally be secured using certificates. A prime difference between Osiris and Radmind that makes the latter worth inclusion as a separate case study is the fact that Radmind not only checks and notifies one of differences, but furthermore automatically removes those differences.

2.2.3 Summary

Other integrity checkers not resident on the system are Veracity[10], Samhain[11] (in networked mode), and GFI LANguard System Integrity Monitor[12] (requires the purchase of another product from the vendor). It should also be noted that some of the tools listed in this section are not specifically designed to be integrity checkers. For example, Radmind is primarily aimed at keeping a set of systems homogenous; however, since the effect of Radmind is to do integrity checking as a side-effect, it has been noted here as an integrity checker.

2.3 Kernelspace checkers

Kernelspace checkers are more difficult to write, and are inherently less portable, than userspace checkers. The design of such integrity checkers must be carefully thought out in terms of performance as well as effectiveness since incurring a large enough penalty every time that an executable is run can quickly lead to a slow or unusable system.

Nevertheless, kernelspace checkers are better suited for checking file integrity than userspace checkers. Self-protection is easier since the checker is no longer limited to the protections provided by default, and may add protections of its own. Continual checking is also easier since all binaries must be processed by the operating system kernel before being executed. More meta-information in the form of internal kernel structures is available to test the binary against. As a final point in favour of placing an integrity checker in kernelspace, a kernelspace integrity checker is able to allow or deny execution of a given binary, which is something that a userspace program may find difficult.

2.3.1 Case Study: I^{3}FS

I^{3}FS, pronounced I-Cubed-F-S, is the *I*n-Kernel *I*ntegrity Checker and *I*ntrusion Detection *F*ile *S*ystem. It takes the form of a stackable filesystem – one which may be layered just below the Virtual File System (VFS) and the real filesystem

[9]Radmind homepage `http://rsug.itd.umich.edu/software/radmind/`
[10]Veracity homepage `http://www.rocksoft.com/rocksoft/veracity/`
[11]Samhain homepage `http://samhain.sourceforge.net/`
[12]GFI LANguard homepage `http://www.gfi.comm/lansim/`

in a kernel. I^{3}FS uses a Tripwire-like model of comparing the existing system to a known-good database to discover whether a file has been tampered with; if so, I^{3}FS "blocks access to the affected file and notifies the administrator"[9].

I^{3}FS checks files with an associated policy instead of checking all files. This leads to an increase in speed, but also means that unauthorized code is allowed to run on the system since it (by definition) has no policy associated with it. Files are checked using both a cryptographic hash of file contents and a file metadata. As a self-protection measure, I^{3}FS hides the database files it uses on the filesystem and furthermore blocks direct access to them. I^{3}FS has many advanced and innovative features such as per-page checksumming of files, caching the results of a check, periodic checks for certain files.

2.3.2 Case Study: DigSig

DigSig is a security module created using the in-kernel hooks provided by the Linux Security Module (LSM) Framework. It checks only Executable and Linkable Format (ELF)[11] binary files, and does so by using a userspace program to embed a cryptographic hash of the file encrypted with a private key – a digital signature – within a segment of the file itself. DigSig "verifies these signatures at execution time and denies execution if the signature is invalid"[1].

A problem that has plagued DigSig in the past is that of signature revocation: it has been easy to replace a signed binary with another, older signed binary without the system registering any discrepancy. To remedy this, recent versions of DigSig have implemented a revocation list system. For performance reasons DigSig (like I^{3}FS) caches the results of security checks; for security reasons, it does not cache results for any files that are mounted via the network using the Network File System (NFS)[8, 10].

2.3.3 Case Study: WLF

WLF modifies the kernel handlers for various binary formats, thus distributing the effort of verifying executable integrity over a reasonably large section of code[4]. Each file is signed by a specific userspace handler for that format. WLF implements a secure cache that copies frequently-accessed files to kernel memory, then runs them from there; this means that even files on network filesystems can be executed securely.

Revocation in WLF is handled through a rather interesting system of directory modification, which effectively reduces down to a certification system with the certificate heirarchy being the directory hierarchy.

2.3.4 Summary

Other kernel-based file integrity checkers are SOFFIC[13], TrojanProof[14], CryptoMark[3] and Umbrella[6]. Though they offer some features that differ from the case studies above, none are significantly different enough to warrant a case study of their own.

2.4 Kernelspace/Userspace (hybrid) checker

A hybrid integrity checker is one that checks files both from userspace and from kernelspace. Only one such checker, to the best of the author's knowledge, exists. It is described in [7] and exists only as a proof-of-concept implementation. In the "Signed Binaries" design, the signature is embedded within the file to be checked. Upon file execution being attempted, the kernel validates the integrity of the file and either allows or disallows execution. Interpreted files such as scripts are validated by modified interpreters, each of which is signed: the interpreter validates the file and passes the result to the kernel. WLF (see 2.3.3) takes many ideas from the

"Signed Binaries" design, though it chooses to do verification through modified kernel handlers rather than modified interpreters.

A hybrid integrity checker is an attractive proposition as it can build on the advantages of both kernelspace and userspace. The respective advantages of each of these approaches has been dealt with above; the advantages and disadvantages specific to a *hybrid* design shall be discussed now. Chief among advantages is the fact that a hybrid design can handle any number of different executable formats whilst other integrity checkers may be reduced to only validating a single format (most commonly ELF on Linux). This is offset by the disadvantage of maintenance: having to maintain a number of patches to different binaries makes it difficult to maintain the system into the future, and makes the cost of any design change or addition (such as the inclusion of a revocation list) extremely expensive. It is also difficult to communicate between a userspace utility and the kernel in a secure fashion, which may reduce the overall security of a hybrid solution

3 INTEGRITY CHECKER DESIGN

This section discusses the design of an ideal integrity checker based on the qualities given in 1.1 and the designs of integrity checkers listed in section 2.

3.1 Performing an Integrity Check

Testing of file integrity should be done through the use of a cryptographic hash function, and also include testing of metadata; the reason for this is laid out in 1.1. In this fashion both the file contents and the file itself are verified to be correct. Whilst it may be argued that any modification of a file would modify the metadata about the file, this is not correct in the case of direct access to a file via a block device or similar method: such access bypasses the filesystem layer entirely and acts on the data on-disk.

3.1.1 Situating the Integrity Checker

The integrity of a file could be tested from userspace, kernelspace, both, or a third-party machine. We exclude userspace as an option for reasons of security: it is either necessary to host the checker binary and any files needed by it on a separate read-only medium, or to accept that our checker integrity cannot be assured. We may also discard userspace as an option because it makes continuous checking very difficult. Either of these reasons will suffice to remove it from our consideration. A hybrid userspace/kernelspace solution is only effective if userspace may be trusted; in the event of a compromise this cannot be assured, and it is because of this that we discard a hybrid as an option.

A third-party machine is an expensive solution in terms of both latency and downtime; in addition, unless the client program on a system can be secured it is difficult to have a truly secure system. In the case of a third-party machine, the client program would either have to exist in kernelspace or userspace. Placing it in kernelspace necessitates adding a great deal of complexity required for secure communication to the kernel, and much development and maintenance time; and for reasons already mentioned, placing it within userspace is insecure.

The operating system kernel seems to be an ideal place to situate a checker. It is reasonably secure, provides most of the facilities required to check files, and makes continuous checking, automation, self-protection, and meta-information checking easy enough to be viable. It is for these reasons that kernelspace seems to be the best place to situate a file integrity checker.

3.1.2 Performance

In an integrity checker that is placed within the kernel, performance is paramount. Implementing a cache of verified and/or frequently-used files, as has been done by [1, 9, 7, 4], is an excellent way of increasing performance. Another modification that could be made is to optionally make certain files immutable – untouchable by any userspace process – in which case they need not be checked upon execution, but may simply be executed as-is. This may be extremely useful for certain frequently-used commands such as *ls*.

3.2 Selective Checking

Whilst many integrity checkers only check binary files (and some only check binary files of a specific format), it seems necessary to check all executable files and some non-executable files to ensure that a "strong" intrusion is always reduced to a "weak" form. The reason for this is that certain executables (for example, *sshd*[13]) use configuration files to determine how they should behave. Changing the configuration file could lead to additional security holes being created on the system. For efficiency reasons it makes sense to have a policy file that contains the associated non-binary files that should be checked if a certain binary is invoked; the alternative would be to check every 'special' non-binary file every time that any binary is executed.

The selective checking of files also has implications for ensuring the integrity of libraries used by a given dynamically-linked binary. A library may either be an executable in its own right or an archive containing reusable code. In the former case, testing the integrity of the file is done through the usual executable test. In the latter, selective checking provides us with total assurance: we need only list the libraries that an executable file depends on as associated with that file, and any attempt to execute such a file would trigger a test of the libraries that it depends on. It is interesting to note that this solution to checking dynamically-linked binary files is entirely cross-platform as it makes no reference whatsoever to the *method* by which a given operating system may choose to load a library!

Ideally, the integrity checker should contain use at least three files:

database This is the database of file hashes and file metadata.

policy This contains details of which files must be examined when selected other files are executed.

config This contains meta-checking information, such as which cryptographic hash function should be used.

3.3 Checker Integrity Assurance

3.3.1 Choosing a Medium

Storing a database containing file hashes and metadata could happen either on-disk or on-media. The former has the advantage of convenience; the latter has the advantage of security since the media may be read-only and/or removable. Given that a system may have to be updated every day (or even more frequently!), storing the file data on media may be arduous: one would have to make the media writable, then update the file data, and then make the media read-only once more; this would involve some sort of user interaction, for if it were possible to do programmatically there would be very little point in having the data read-only in the first place!

Given that we may protect a database at a low level from within the kernel, we may safely choose convenience at this point and store the data on-disk in a

[13]Part of the OpenSSH package, widely used on UNIX platforms

secure fashion. Protections that we may provide from the kernel include placing the database in a directory that is entirely shielded from userspace: nobody, not even the administrator, can gain a handle to this directory in any way whatsoever. That also means that the directory is effectively invisible. We could also disallow raw disk access for the specified device. This should help to guard against attempts to replace the file with another one, or to edit it in any way.

3.3.2 Assuring Integrity

The integrity of the database can be assured by having it hashed and signed by a third-party. The public key can be constant and compiled into the kernel image. When the database needs to be checked, the hash can be checked. Nobody but the third-party can create the hash. Importantly, this method can be used to safeguard every file used by the system. We choose not to do so because the overhead imposed by public-key operations such as decryption of RSA-encrypted text is high, and the penalty imposed on the system may be prohibitive, as shown by [3]. Instead of signing each file, under this scheme each file is in fact *indirectly* signed via the signature on the database – and the entire system is made much more efficient.

3.3.3 Upgrades

The system should be easily updatable to take into account upgrades that have occurred to the system. We propose an update mechanism that helps to ensure that the old database is never replaced with a new one without the replacement being verified as being both authentic and newer than the current version. This is done in the following manner:

1. Each *filename* contains, as the first line, the date+time on which the file was created, a space, and a n-byte-maximum profile name.

2. The full path of the replacement file is provided to the system. This may happen via many different means: on Linux a custom system call, a file entry in /proc, or the use of sysfs will all accomplish the same objective. The *filename* to be replaced is also specified by the same means.

3. If *filename* does not exist within the shielded directory, the upgrade is allowed if the signature on the file is valid. If *filename* does exist, the replacement date+time must be greater than the date+time of the current file; in addition, the profile name must match the profile name of the current file exactly; and, of course, the signature of the file is checked. If one of these checks fails, the upgrade is not done.

4. The replacement file is copied to a secure location. This may be done at the same time that the hash-check is being done to avoid replacement attacks.

5. The old file is renamed to *filename.old*.

6. The new file is renamed to *filename*. At this point, both *filename* and *filename.old* exist.

7. Synchronization done; buffers are flushed to disk.

8. *filename.old* is removed.

A side-effect of this upgrade process is that, on system startup, a test for *filename.old* must be done if a test for *filename* fails. In (1) we provide a consistent upgrade format for the policyfile, databasefile, and any other files along the way. In (2) we provide a consistent way to upgrade or add files. In (3) we verify the file authenticity.

In (4) we guard against replacement attacks. In (5), (6), (7) and (8) we ensure that the system remains in a consistent state even if any one of these steps is interrupted.

In a worst-case scenario, all protections have been defeated and the files used by the system have been compromised. In the unlikely event that this occurs, we provide a userspace program that gets a specified filename's time+date, profile name and hash and passes it to the trusted third-party. This can occur over SSH-encrypted tunnel, via SMTP, via a SSL-encrypted web connection, via physical means, and so forth. The third party verifies that the file is indeed the latest that has been issued and replies with a "Yes" (the file is OK) or "No" (the file is not OK). In the case of a total system compromise by an insider, this last guard should detect the issue and allow an alert administrator to detect the problem.

3.4 Action

Actions taken in the event of an attempt to execute unauthorized code include

Deny The execution attempt is denied

Log The execution attempt is logged

Lockdown Access to the file that triggered the problem is restricted.

System lockdown Operations on any files become restricted: no deletion, move, rename, or link operations are permitted.

The above are not mutually exclusive. It is therefore possible to, for example, both *Deny* file execution and *Log* the attempt.

3.5 Conclusion

Though there are many products which are designed to check the integrity of files on a system, few of them are designed to do the task effectively. In this case it happens that doing the task ineffectively may be more dangerous than not doing the task at all, as it may lead to a false sense of security. Taking the ideal qualities and existing qualities of tools available today into account, it is clear that the problem can be addressed effectively and a design has been proposed that does so. The design proposed is not simply theoretical and abstract: instead, it describes software that could be implemented and it is conceivable that future work on this topic may include an even more detailed architecture that follows on from the proposed design, as well as a concrete implementation thereof.

References

[1] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. DigSig: Run-time authentication of binaries at kernel level. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, November 2004. Pre-print.

[2] William A. Arbaugh. A Patch in Nine Saves Time? *Computer*, 37(6), June 2004.

[3] Steven M. Beattie, Andrew P. Black, Crispin Cowan, Calton Pu, and Lateef P. Yang. CryptoMark: Locking the Stable door ahead of the Trojan Horse. Technical report, July 2000.

[4] Luigi Catuogno and Ivan Visconti. A Format-Independent Architecture for Run-Time Integrity Checking of Executable Code. In *Proceedings of the Third Conference on Security in Communication Networks*. Dipartimento di Informatica ed Applicazioni, Universita di Salerno, September 2002. Available: http://libeccio.dia.unisa.it/wlf/scn02/index.html.

[5] Luigi Catuogno and Ivan Visconti. An Architecture for Kernel-Level Verification of Executables at Run Time. *The Computer Journal*, 47(5), September 2004.

[6] Soren Nohr Christensen, Kristian Sorensen, and Michel Thrysoe. Umbrella: We can't prevent the rain... - But we don't get wet! Master's thesis, Aalborg University, June 2004.

[7] Gerco Ballintijn Leendert van Doorn and William A. Arbaugh. Design and Implementation of Signed Executables for Linux. Technical Report CS-TR-4259, June 2001.

[8] Bill Nowicki. RFC 1094 - NFS: Network File System Protocol Specification. Technical report, Sun Microsystems, Inc., March 1989.

[9] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004. Stony Brook University.

[10] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC 3010 - NFS version 4 Protocol. Technical report, Sun Microsystems, Inc.; Hummingbird Ltd.; Zambeel, Inc.; Network Appliance, Inc., December 2000.

[11] Tool Interface Standards Committee. Executable and Linkable Format (ELF). Specification, Unix System Laboratories, 2001.

[12] Tripwire, Inc. Tripwire for Servers Datasheet. Technical report, Tripwire, Inc., 2005.

[13] Vinicius da Silveira Serafim and Raul Fernando Weber. The SOFFIC Project. Technical report, UFRGS Security Group - GSeg. Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, RS, Brazil, 2002.

[14] Michael A. Williams. Anti-Trojan and Trojan Detection with In-Kernel Digital Signature testing of Executables. Technical report, Security Software Engineering: NetXSecure NZ Limited, April 2002.