www.ietdl.org

Published in IET Information Security
Received on 9th November 2012
Revised on 1st March 2013
Accepted on 6th May 2013
doi: 10.1049/iet-ifs.2012.0346

ISSN 1751-8709

Operating system security by integrity checking and recovery using write-protected storage

Jerzy Kaczmarek, Michal R. Wrobel

Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Gdansk, Poland
E-mail: wrobel@eti.pg.gda.pl

Abstract: An integrity checking and recovery (ICAR) system is presented here, which protects file system integrity and automatically restores modified files. The system enables files cryptographic hashes generation and verification, as well as configuration of security constraints. All of the crucial data, including ICAR system binaries, file backups and hashes database are stored in a physically write-protected storage to eliminate the threat of unauthorised modification. A buffering mechanism was designed and implemented in the system to increase operation performance. Additionally, the system supplies user tools for cryptographic hash generation and security database management. The system is implemented as a kernel extension, compliant with the Linux security model. Experimental evaluation of the system was performed and showed an approximate 10% performance degradation in secured file access compared to regular access.

1 Introduction

Computer systems are commonly used to store important information regarding technical, financial and personal data, which require effective protection from unauthorised access or data fraud. Numerous threats for computer system security may be mentioned, including viruses, Trojan programs, rootkits and others. Most computer system attacks are performed using network access and the likelihood of a security breach is increased if the computer system is not adequately protected. Methods for system protection have been designed and applied, including firewall systems, access control security policies and intrusion detection systems. However, there are still serious threats of successful attacks and we can assume that there will always be a risk of successful intrusions on computer systems. Therefore, it is necessary to apply additional methods that detect intrusions to a computer system apart from methods that attempt to prevent such attacks. Methods for file modification monitoring are one of the most effective methods of intrusion detection.

File integrity checking systems [1] utilise the mechanism of a unique digital fingerprint that is calculated from the contents of a file. The fingerprint, also known as a checksum, is stored in a secured database and retrieved to check if an unauthorised modification to a file has been made. Algorithms used to calculate files fingerprints are critical for the effectiveness of the method, as they must ensure that any file modification will result in a different checksum value. Additionally, it must be impossible to reconstruct file contents from its checksum and simultaneously checksum calculation should be computationally effective.

Algorithms that generate fixed length cryptographic hashes are most commonly used for file checksum generation. There are cryptographic functions available such as Message Digest Algorithm, Secure Hash Algorithm and Tiger Message Digest Algorithm. The algorithms must be resistant to attacks such as first pre-image attacks, second pre-image attacks and collision generation [2].

Although the existing integrity checking systems improve system security [3], they also have weaknesses that may be used by an intruder to eliminate the protection in certain situations. Typically, the systems insufficiently protect the security database and are vulnerable to attacks that modify its binaries. The weakness concerns both systems that are run as user applications and systems that are integrated in the kernel. Additionally, the existing systems do not provide file backups, which results in the need of additional administrative work to restore infected files each time an intrusion occurs.

We propose a novel system that enables file integrity verification – integrity checking and recovery (ICAR) – implemented as an in-kernel Linux security modules (LSM)-based module [4]. Our system overcomes the shortcomings of existing solutions taking the following main assumptions:

1. Write-protected media is used to store crucial files: system kernel with ICAR module, backups of the protected files and cryptographic hashes database.
2. The system supplies a backup of protected files that enable automated recovery if the files have been modified in an unauthorised way.
3. ICAR is integrated with operating system kernel using the LSM mechanism and runs in the kernel space.

Experiments showed that our system is an efficient mechanism of operating system protection against unauthorised file modifications.

The rest of the paper is organised as follows. The next section presents related work and background for our research. Section 3 presents the ICAR protection model, architecture and applied algorithms. System implementation, installation and configuration are discussed in Section 4. Section 5 presents experimental evaluation.

2 Background and related work

Most file system integrity checking tools leverage a common pattern for operation steps as shown in Fig. 1 [5]. First, the integrity checking process reads configuration data to determine which files should be protected. Then the process calculates cryptographic hashes of the files and stores the hashes in a database.

During regular system operation, the process checks integrity by recalculating cryptographic hashes of files and comparing them with the values stored in the database. If a security violation is detected, appropriate security actions are taken. The actions may cover generation of a report about potential intrusions, notification to the system administrator, file access denial or system shutdown.

The Linux operating system distinguishes between user-level code and kernel-level code, which is an important feature for integrity checking tools. Two kinds of tools are distinguished in this context: those that run in user space with root privileges and those that run in kernel space.

User-level code covers regular applications, text and graphical user interfaces and operating system commands invoked by a user. Kernel-level code covers, among other things, applications for memory management, process management and file system operations. A typical threat to computer system security that occurs at user-level code is a situation in which an error, such as buffer overflow, in an application is used to overtake control of the computer system. This kind of threat is virtually impossible to eliminate because of a large number of applications and their high complexity.

User-level integrity checking tools are run as regular applications. Usually, the integrity checking procedure is executed periodically, so integrity violations are detected during explicit executions rather than automatically on occurrence. The Tripwire [5] tool developed in the Purdue University in 1992 is one of the first commonly used intrusion detection system. Tripwire is a seminal integrity checking system for other user-level tools such as AIDE, Veracity and Integrit [6]. The tool enables definition of protected file lists, signature generation, file verification and reporting of violations.

It should be noted that user-level integrity checking tools are not aimed at intrusion prevention, but rather effective detection of an already occurred security violation. The procedure of file integrity checking is triggered either periodically or manually by the administrator.

Kernel-level integrity checking tools supply higher security than user-level ones. Operating system kernel verifies security constraints during operations, for example, access privileges when a process requests a file access. Regular kernel operations may be extended with integrity checking if a security mechanism is a part of the operating system, in particular a procedure for verifying file cryptographic hash may be supplied. If hash verification indicates that the file has not been modified, file access is granted. However, if hash verification fails, which might indicate an unauthorised modification, file access may be denied.

Security systems can be integrated with the kernel in several different ways. Stackable file system [7] consists of a number of independent functional layers and may be used as an efficient solution that enables implementation of file system security in the kernel space. File system functionality may be extended by adding an additional layer responsible for security. The layer is typically located on the top of the stack during file system mounting and intercepts and controls file access by processes. The solution enables separation of file system integrity checking code and kernel code.

System calls is another approach that may be used to implement kernel-level security, as the mechanism enables programs to request kernel services [8]. In order to implement a security extension, standard system calls are replaced with custom functions implementing secure file

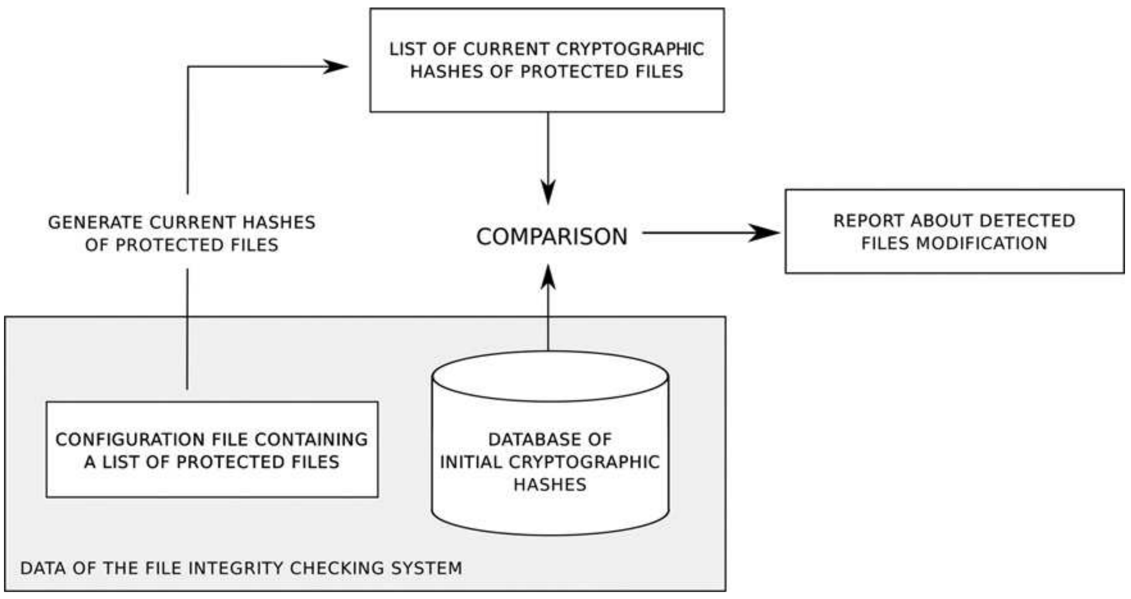
LIST OF CURRENT CRYPTOGRAPHIC HASHES OF PROTECTED FILES
GENERATE CURRENT HASHES OF PROTECTED FILES
COMPARISON
REPORT ABOUT DETECTED FILES MODIFICATION
CONFIGURATION FILE CONTAINING A LIST OF PROTECTED FILES
DATABASE OF INITIAL CRYPTOGRAPHIC HASHES
DATA OF THE FILE INTEGRITY CHECKING SYSTEM

Fig. 1 *Main components for file system integrity checking tools*

access. The mechanism, however, has many drawbacks [9]: it is not race-free, may require code duplication and may be limited in expressing the security context.

Finally, the LSM project [9] has been developed to supply an access control framework for the Linux kernel that enables different control models implemented as loadable kernel modules. The basic abstraction of the LSM interface is to mediate access to internal kernel objects. The mechanism uses hooks that are placed in the kernel code just ahead of the access. The hooks are used to call LSM functions that supply information about granting or denying a requested access. The LSM framework is used, among others in AppArmor and SELinux [10].

The secure on-the-fly file integrity checker (SOFFIC) [11] and in-kernel integrity checker and intrusion detection file system (I^3FS) [12] systems are important implementations of file system integrity checking that leverage kernel-level access. The SOFFIC system modifies system calls responsible for file access. It stores the list of protected files and their initial hashes in regular files, which is the primary drawback of the system because attackers can easily remove or change such important data. I^3FS system is an in-kernel, on-access integrity checking system that is designed on a stackable file system. It is capable of blocking access to affected files and notifying the administrator. The design assumes that the file system is the most appropriate location for security modules because most intrusions cause file modification. The system implements four databases located in system kernel space that concern: (i) policy options associated with files, (ii) checksums of file data, (iii) metadata checksums and (iv) access counters.

Both SOFFIC and I^3FS, similar to Tripwire, are vulnerable to attacks that modify the database of file hashes or the operating system kernel. Attacks of this kind may effectively disable the security mechanism. Our solution differs in that we leverage the LSM mechanism, which effectively protects our tool from unauthorised modification at the kernel level. ICAR supplies a backup mechanism that enables continuous work despite an intrusion. Additionally, we use write-protected storage to store cryptographic hashes, security module and system kernel ensuring that no modification can be made to them.

Different approach to file integrity checking are used in some network intrusion detection system [13] and hardware systems such as intrusion detection on disk [14]. But despite the good performance, such solutions are more complicated and expensive than those designed by us.

3 ICAR protection model

The design of a security mechanism requires analysis concerning scope, time and manner of protection. A running operating system may be viewed as both files and running processes that may become a potential target of an attack and should be protected. The Linux operating system is a convenient platform for research in the field of security as it is freely available in both binary and source code. It can be modified and extended using the open source approach and existing licenses. Therefore, the system was chosen as the testbed operating system for design and implementation work.

Although numerous methods and tools have been supplied for system protection, it cannot be guaranteed that an intruder will not violate system security and grant unprivileged access to system resources. The existing Linux security mechanisms supply a relatively more effective protection for running processes than for the file system. Therefore, a mechanism should be designed that protects file contents, especially considering important configuration and system files.

The proposed security system consists of three layers as shown in Fig. 2. System layers are as follows:

1. *Kernel-level layer* – is responsible for the verification of file system integrity.
2. *Data layer* – contains critical data for the security mechanism: database that stores initial cryptographic hashes and backup copies of protected files.
3. *Utility layer* – equips the end user with tools that simplify the administrators work in configuration and management of the system.

The module of kernel-level layer is loaded into RAM as a part of operating system kernel during system startup. It is responsible for calculating file cryptographic hashes that are further stored by the data layer. Given the known attacks on the elderly functions [15], it is recommended to use a SHA 256 algorithm or a new one. The layer also contains a cache that is used by the protection algorithm, described further in this section, to store the results of file verifications that have already been performed. The use of cache increases the speed of the security system as it limits the necessity of signature calculating for files that have already been verified.

File hashes stored by the second layer are used by the kernel-level layer to detect if an unauthorised modification

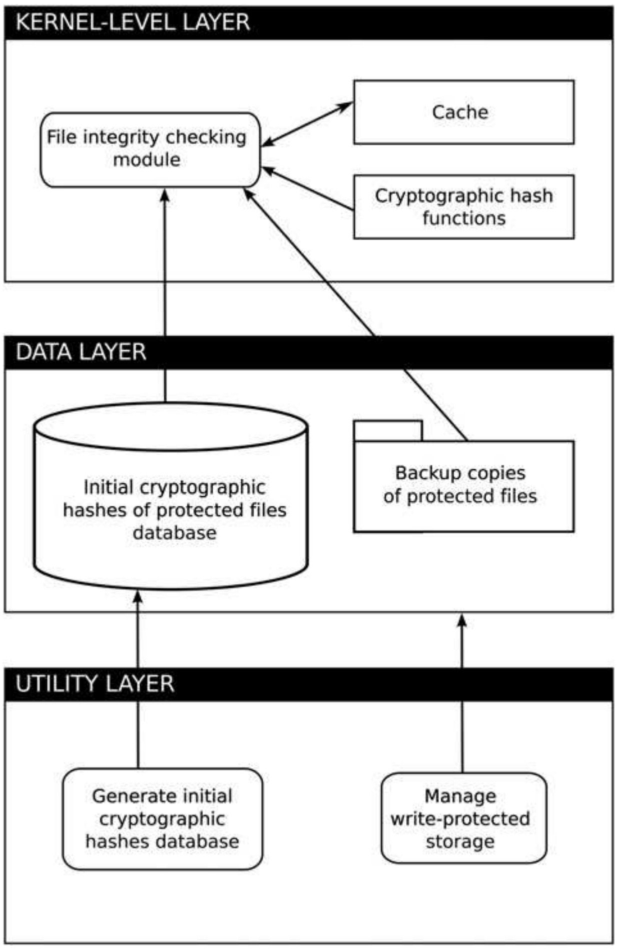
KERNEL-LEVEL LAYER
File integrity checking module
Cache
Cryptographic hash functions
DATA LAYER
Initial cryptographic hashes of protected files database
Backup copies of protected files
UTILITY LAYER
Generate initial cryptographic hashes database
Manage write-protected storage

Fig. 2 *ICAR system layers*

has been made to the system. Additionally, file copies are used to restore the original file contents if unauthorised file modifications have been detected (as described further in Section 3.1). Both cryptographic hashes and file copies are stored in write-protected data storage. The mount point of the storage is additionally protected by ICAR to prevent from mounting a fake security database. The kernel-level layer intercepts mount operations, analogous to intercepting file read operations, and blocks mount attempts of the directory with the security database.

Developed helper tools cover applications that enable selection of protected files, generate initial cryptographic hashes and files backup, which are stored in the data-layer. Tools also allow to monitor file system integrity. If necessary, a system administrator may receive alarm notifications from the monitoring system. Information about compromised files and file recovery may simplify administrators work in detecting intrusions.

3.1 File protection algorithm

Fig. 3 shows the main algorithm for integrity checking used by ICAR during reading or executing attempt. The algorithm for integrity checking is activated each time a process requests a read access to a file. First, the algorithm reads the security database to check if the accessed file is in the group of protected files. If there is no information about the file in the database, it is assumed that the file is not protected for integrity and access is granted for the requesting process.

If there is information about the file, which means that the file is protected for integrity, ICAR verifies if file contents has been modified since system startup. The check requires calculating the cryptographic hash from the present contents and then comparing that against the value of the initial hash stored in security database. However, despite the fact that the system uses a relatively effective kernel built-in cryptographic function, the calculation is time-consuming.

Considering performance issues, ICAR implements a cache that stores information about positively verified files. If a file has been checked on a previous access, there is no need to recalculate the hash of a file as long as the file has not been modified. Therefore, when a process requests access to a file that is already in the cache, permission is granted immediately. Otherwise when the cache does not contain information about a file, which is protected for integrity, it is necessary to calculate the cryptographic hash from the present file contents and compare it against the initial value. If the verification is successful, the security mechanism stores the information about the file in the cache for further use and grants access to the file for the requesting process.

If the calculated hash differs from the initial hash, a security procedure is activated covering an attempt to restore the original file content from backup and notifying the system administrator. In the first step, the mechanism attempts to restore the original file by copying it from

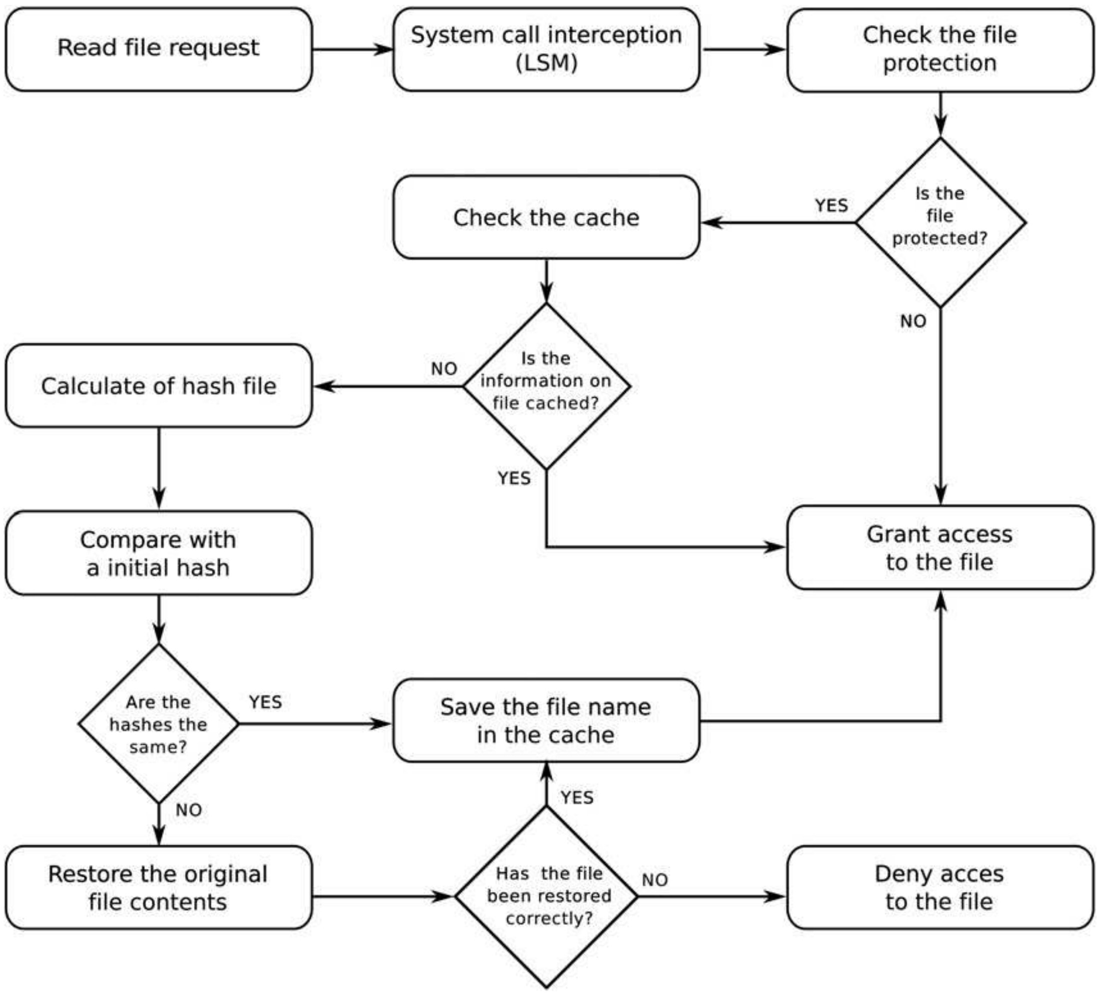
Read file request
System call interception (LSM)
Check the file protection
Is the file protected?
YES
NO
Check the cache
Is the information on file cached?
NO
YES
Calculate of hash file
Compare with a initial hash
Grant access to the file
Are the hashes the same?
YES
NO
Save the file name in the cache
YES
Restore the original file contents
Has the file been restored correctly?
NO
Deny acces to the file

Fig. 3 *File system protection algorithm*

the write-protected storage backup to the hard disk of the computer system. If file restoration is successful, the tampered file is replaced with the original one. Then, the algorithm follows steps for positive file verification granting access to the file for the requesting process. If, however, file restoration fails for any reason, file access is denied. The file is not accessible for users and processes running in the system for reading and executing.

The security system may take other security actions in response to a detected unauthorised modification of a file depending on configuration. The system may log the information in system logs, alert administrator, block IP addresses or even shutdown the operating system. System shutdown may be necessary if file modifications are detected frequently, which may indicate a considerable attack on the system.

Fig. 4 shows two helper algorithms. The first is responsible for cleaning cache during a write operation. Independently from integrity checking, the system monitors file modifications and clears cache information about a file when its modification is detected. The second algorithm describes how the system protects the mount point with the ICAR data against unmounting or remounting operations.

3.2 Cache construction

The structure of the cache is an important issue related to the performance of the ICAR system. The solution was designed with the assumption of low memory usage and fast file verification.

It is sufficient to store a unique identification number of verified files in the cache. Therefore, even when hundreds of thousands of files are protected, the size of the buffer will be in the order of megabytes. This approach also eliminates the risk of an attack, in which the contents of verified files is changed using hard links (information about these files is stored in the cache).

To ensure high search performance, it is necessary to use a hash table to store the information on previously verified files. A suitable hash function must be selected to minimise generation of collisions, as well as an algorithm for dealing with collisions occurrence.

Additionally, periodic cleaning of the cache is assumed, which allows reduction of the size of the cache and refresh its contents. This mechanism may be activated periodically or when exceeding a predefined limit.

3.3 Security issues of ICAR integration with the operating system

An effective implementation of the security system requires resolution of two design issues: a method of data storage and a method for integration with operating system kernel.

We use a write-protected storage for system kernel, the integrity checking module, initial cryptographic hashes and file backups. We ensure write protection using hardware mechanisms, which eliminates the threat of modification

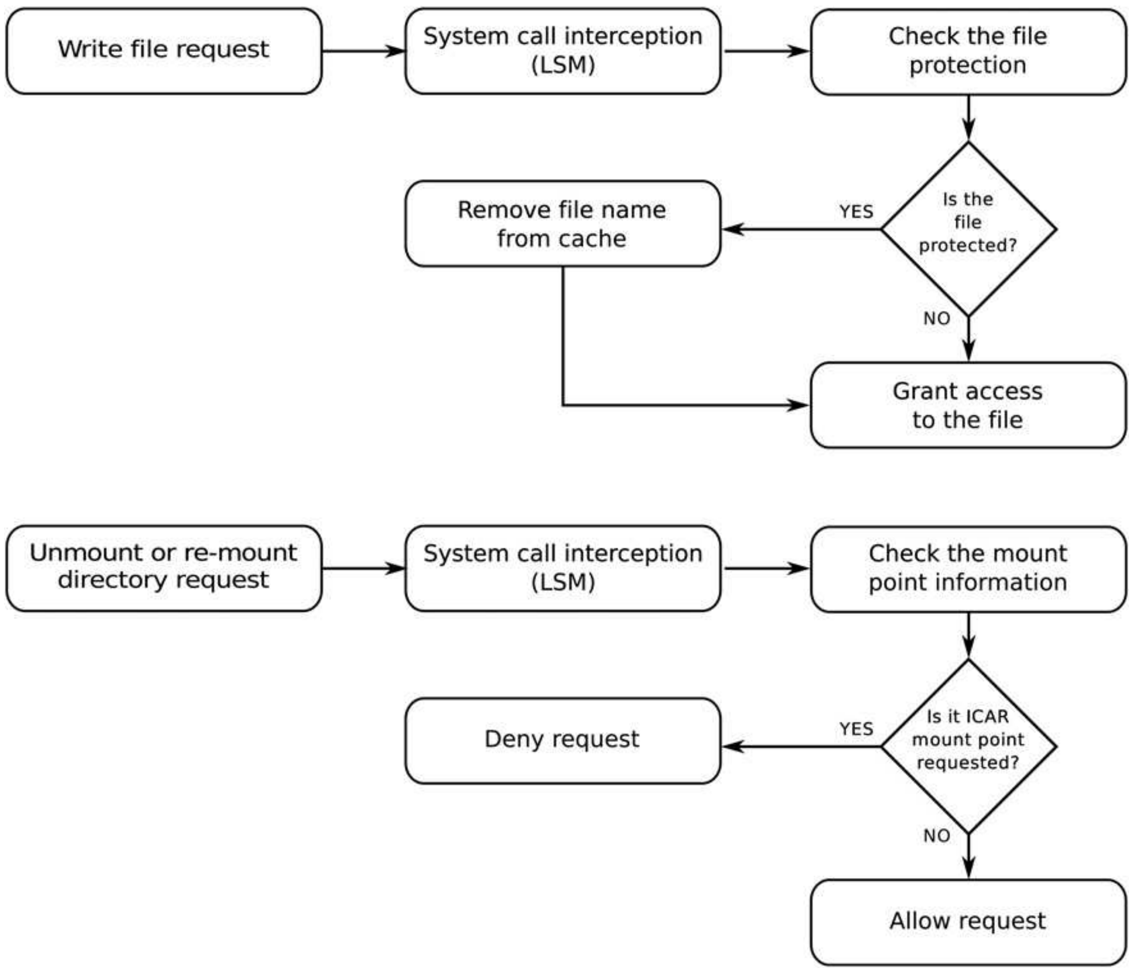
Write file request
System call interception (LSM)
Check the file protection
Is the file protected?
YES
NO
Remove file name from cache
Grant access to the file
Unmount or re-mount directory request
System call interception (LSM)
Check the mount point information
Is it ICAR mount point requested?
YES
NO
Deny request
Allow request

Fig. 4 *File system protection algorithm*

and guarantees the effectiveness of the solution. In practice, a LiveCD Linux distribution or write-protected USB flash memory are most popular mechanisms that can be used in the developed system.

The system assumes that the kernel and the ICAR system are the only modules stored in write-protected storage and other system files may be stored in traditional disks. The bootup process loads system kernel from the write-protected storage and then continues with traditional booting using data from hard disks. If ICAR detects a file modification during integrity checking, it restores the original file contents using the backup copy from the secure storage.

ICAR design assumes that the module is integrated with system kernel, which minimises the threat of its elimination by an unauthorised user. In practice, different solutions are available for such integration in Linux, including system stackable file systems, system calls and LSM. Table 1 shows methods of kernel–module integration and their selected properties.

We decided to use LSM because it is well suited for security actions such as verification and filtering of user operations. LSM supplies a well-defined API, allowing integration of security modules with system kernel. It is also resistant to TOCTTOU attacks [16, 17]. ICAR is implemented as an LSM-hooked module that resides in Linux system kernel. The use of LSM ensures that our module will not be removed from the operating system kernel by an intruder.

An unauthorised modification of cache is a serious threat that must be prevented. Three groups of attacks on cache has been identified: malware modules loaded into the operating system kernel, writing directly to kernel memory and writing directly through partition device file. All of these possible attacks can be blocked by the application of the grsecurity kernel patch.

4 System implementation

The implementation of the ICAR system is an extension for the Linux operating system and aims at verifying the effectiveness of the designed mechanism. The cdlinux.pl Linux distribution was chosen as the runtime environment, although ICAR may be applied in any Linux distribution. The cdlinux.pl is a Debian-based distribution developed by authors since 2001, available from www.cdlinux.pl.

ICAR is implemented in the C language and consists of two main parts: a kernel module and user-level tools. The LSM-based kernel module covers over 1.000 lines of code of kernel module responsible for file protection. The cache mechanism utilises the Linux inode hash function and stores its data in slab allocator.

The implementation supplies user-level tools that simplify setting-up, initialisation and configuration of ICAR. Additionally, a script was prepared that automates the creation of the operating system containing the designed security mechanism. The generated system may be used to create a LiveCD image or USB memory contents.

ICAR sources are available at http://www.cdlinux.pl/icar. The system is available under the GPL license and may be used by computer system users or other researches for further studies.

4.1 *System installation and configuration*

ICAR installation and configuration requires administration steps similar to installation of other integrity checking software. Fig. 5 shows the following main steps of ICAR installation:

- ICAR installation in the operating system, which covers:
  - Patching system kernel with the ICAR module.
  - Installation of ICAR user-level tools.
- Selection of protected files.
- Generation of initial cryptographic hashes.
- Preparation of write-protected media containing: initial cryptographic hashes, protected file copies and operating system kernel with the ICAR module.
- System restart with the new kernel from the write-protected media.

ICAR installation and configuration requires special security means as the operating system may be prone to attacks during reconfiguration of the module. It is recommended that the target machine is clean and isolated (run in single-user mode and disconnected from any network). After taking the security means, the ICAR system may be installed and configured. If the system has already been installed, it should be disabled and reconfigured concerning new file versions or protected file selection. A configured system may be used as long as the contents and selection of protected files do not change. Updated system data is generated and written in a write-protected storage. The operation requires physical access to the computer (i.e. it cannot be performed remotely through a network connection). The restrictions ensure high security of the configuration process.

Table 1 Methods of kernel-module integration

	Stackable filesystem	System call interception	Linux security model
initialisation method	mount the file system	load kernel module	load kernel module
required changes	implementation of file system layer with modified read and write functions	read and write functions implementation and substitution	read and write functions implementation
programming language	abstract high level language (FIST)	C	C
properties	easy to maintain, noticeable performance overhead	inefficient and prone to time-of-check-to-time-of-use race	standard part of the Linux kernel

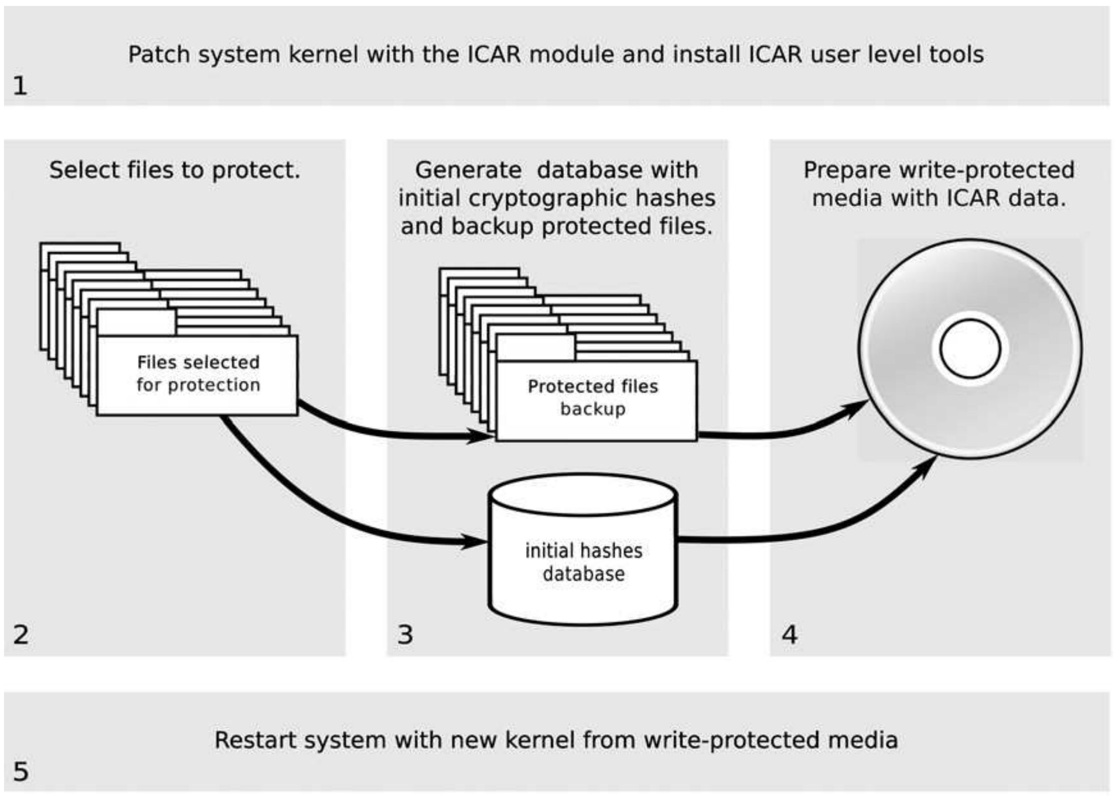
1
Patch system kernel with the ICAR module and install ICAR user level tools
2
Select files to protect.
Files selected for protection
3
Generate database with initial cryptographic hashes and backup protected files.
Protected files backup
initial hashes database
4
Prepare write-protected media with ICAR data.
5
Restart system with new kernel from write-protected media

Fig. 5 *Main steps of security database update process*

4.2 Classification of protected files

It is virtually impossible to protect all of the files in a system because of their size and number. Protection of all files would inevitably lead to significant performance degradation. Therefore, it is necessary to select which files should be protected. The selection results from a security policy that in turn depends on the value of stored information and computer system type, such as a desktop machine or a corporate server.

Significance for security and modification frequency are the main issues that decide whether a file should be integrity protected or not. Files stored on a computer system are classified as operating system files, application files, system configuration files, user data files and others. Protection of operating system and application files is critical for the security of the whole computer system. If an intruder modifies one of those files, he may easily overtake operating system control or install malicious software. The advised security policy assumes that these files should be verified for integrity and cannot be modified without ICAR reconfiguration.

Configuration files are the second security group. These files are less crucial as their unauthorised modification does not immediately result in a security threat. Additionally, relatively frequent modifications of these files may be necessary during regular administrative work. Therefore, the advised security policy assumes that only selected configuration files are protected for integrity. Actually, system administrators decide which configuration files should be protected from modifications depending on system purpose and installed software.

Files from other groups, including user data files and temporary files, are modified frequently and it is not recommended to select them for integrity protection. Although the files are exposed to attacks, their privileges are limited, which poses a reduced threat to the operating system. These files should be backed-up regularly and appropriate policies should be designed to restore the files if an intrusion has been detected or the files have been corrupted.

5 Evaluation and performance

We conducted tests on ICAR to evaluate the performance overhead of the implementation. Two kinds of tests were performed. The first test involved the single calculation of MD5 checksum for files of various sizes, and the second test involved typical file operations (i.e. size modification of graphical files, application compilation and conversion of audio and video files). All tests were performed on a machine with an Intel Core2 Quad Q6600 2.40 GHz processor, 3 GB RAM memory and a Samsung HD322IJ 320 GB 7200 rpm hard disk. We used the Linux 2.6.28 kernel in the experiments. Data-layer of the ICAR system, with the initial cryptographic hashes database and backup of protected files, has been stored on the write-protected USB flash memory. No additional applications were running on the machine during the experiments.

System performance was measured in three different configurations:

- without the ICAR module (NOICAR)
- with ICAR using cache (CACHE)
- with ICAR without cache (NOCACHE)

The abbreviations are used in presented performance graphs to note results for each system configuration.

Table 2 Methods of kernel-module integration

FILE SIZE, MB	NOICAR, s	CACHE, s	NOCACHE, s
0.100	0.13	0.15	0.26
0.315	0.20	0.25	0.61
0.700	0.31	0.42	1.19
1.000	0.39	0.54	2.39
3.150	1.03	1.39	4.88
7.000	2.21	2.84	10.56
10.000	3.11	3.99	15.01
31.500	9.55	12.32	47.30
70.000	21.00	27.16	104.89
100.000	29.96	38.72	149.09
350.000	94.27	122.06	469.85
700.000	208.82	273.02	1042.37
1000.000	298.18	386.90	1487.99

5.1 Performance by the file size

The first experiment verified the dependence between file size and file access time. The experiment concerned the worst-case scenario, in which only protected files were processed. The experiment was executed on binary files of sizes ranging from 100 kB to 1 GB, for which MD5 checksum was calculated. We used the SHA256 algorithm for file protection. The md5sum program was chosen as a representative case, although it may be replaced by any other program that reads the entire file contents. In order to increase calculation accuracy, each file was processed in 10 rounds, with each round consisting of 100 calculations.

Table 2 shows the results for MD5 hash calculation for defined test files. The results were used in regressive analysis to determine if there exists a linear dependency between file size and computation time, which would indicate system scalability.

Considering the high differences in file sizes, two ranges were distinguished during the analysis: file sizes of 100 kB to 7 MB, and file sizes of 7 MB to 1 GB. A detailed analysis of results for small files shows some deviation with Pearson product-moment correlation coefficient of 0.98, as shown in Fig. 6. The deviation is observed regardless of ICAR activation in the operating system. Therefore, it may be stated that the effect results from the operating system, not from ICAR processing. The effect may also result from inaccuracies in measurement of small time intervals. Despite insignificant deviations for small files, it may be stated that a linear dependency exists between file size and MD5 checksum calculation time. Generally, we conclude that ICAR does not change the scalability of file access time in the operating system.

Fig. 7 compares checksum calculation time in three configurations: (i) without ICAR, (ii) with ICAR and the buffering mechanism (CACHE) and (iii) with ICAR without the buffering mechanism (NOCACHE). The analysis of access time shows that the second configuration (CACHE) results in 30% performance degradation compared to a clean system (NOICAR). If the cache mechanism is disabled in ICAR (NOCACHE), file access time increases approximately 400% compared to the time without integrity checking. The speedup in case of buffering results naturally from the fact that the SHA-256 cryptographic hash is calculated once in each test round (consisting of 100 operations). If the buffering is disabled, the hash is calculated for each operation. The achieved results show that the cache mechanism works correctly and increases system performance.

5.2 Performance by the operation type

In the second experiment, we measured representative file operations: source code compilation, image file resolution reduction, audio conversion and video compression. In this experiment, more complex operations were executed as they were not limited to reading the file contents as in the first experiment. Therefore, results supply more representative measures of ICAR system overhead. During the experiments, we measured operation performance in the following configurations: NOICAR, CACHE, NOCACHE, analogous to plain MD5 checksum calculation. The experiments aimed at determining the influence of operation type on processing time.

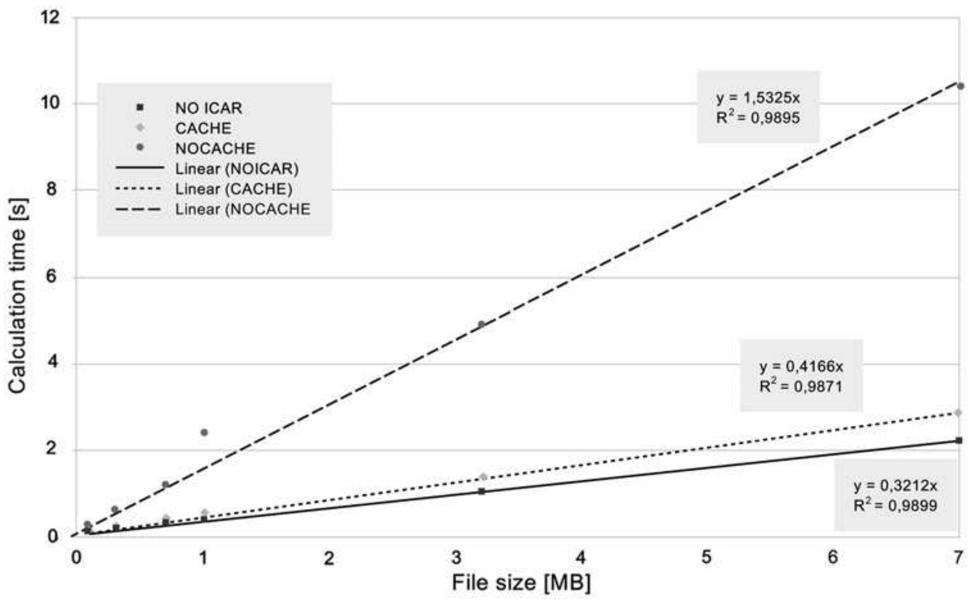

Fig. 6 *File operation time for files smaller than 7 MB*

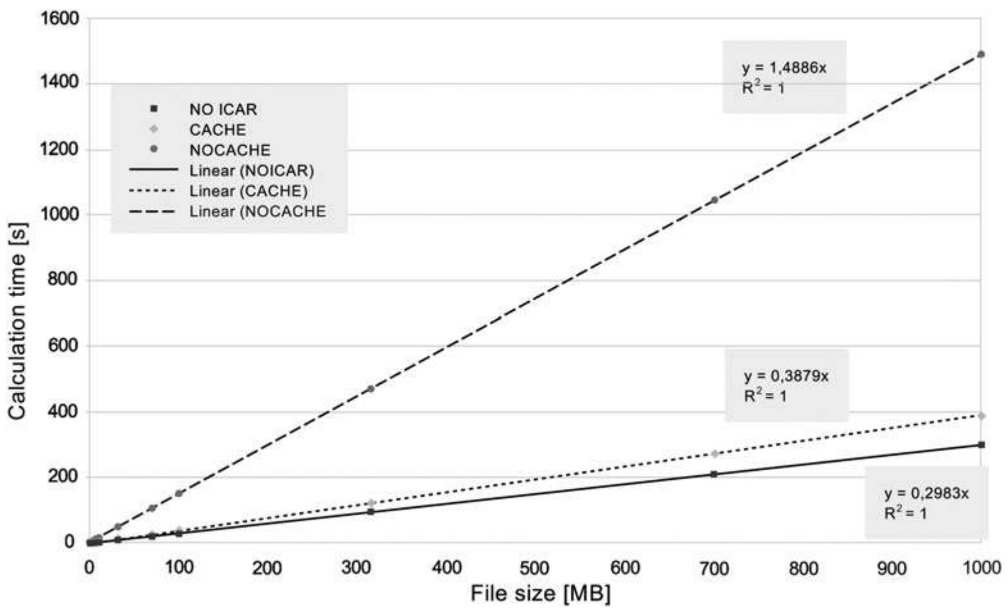

Fig. 7 *File operation time for files smaller than 1 GB*

We selected the following data for performance calculation:

• Application compilation: a C language application consisting of approximately 12 000 files, having 332 MB size in total.
• Image file processing: 20 image files of average size 4 MB, modification of image resolution.
• Audio file processing: 20 audio files of average size 5 MB, file conversion from the MP3 format to the Ogg Vobis format.
• Video file processing: one 1.4 GB video file compression.

Fig. 8 presents time overhead of the operations for an operating system in the two configurations. The bar on the left-hand side compares the processing times for a plain system (NOICAR) and a system with buffered ICAR (CACHE/NOICAR). The bar on the right-hand side compares the processing times between a plain system (NOICAR) and a system with non-buffered ICAR (NOCACHE/NOICAR).

The results indicate the effectiveness of the buffering mechanism, especially in operations that require repeated reading of the same file. The effect is noticeable when comparing results of image and video processing performance. Buffering has a negligible influence in video processing, although the video file is significantly larger than image files. It results from the fact that image files are read several times during processing, while video files are read once only.

The results of compilation tests can be compared with similar tests carried out by the developers of the I^3FS system [12]. With multiple I^3FS policies, ICAR should be compared with the MW policy – whole-file check-summing date. Compiling process overhead for I^3FS is 2.3%, which is slightly better than the results for the ICAR system (2.5%).

The performance increase resulting from buffering depends both on the type of processed files and on implementation details of processing in utility applications. The presented results show that the overall performance degradation of file operations caused by the ICAR system using cache should not be higher than 6% under normal user workloads, and 30% for the worst-case scenario. This indicates that, in practice, ICAR operation will not be noticeable by a regular user.

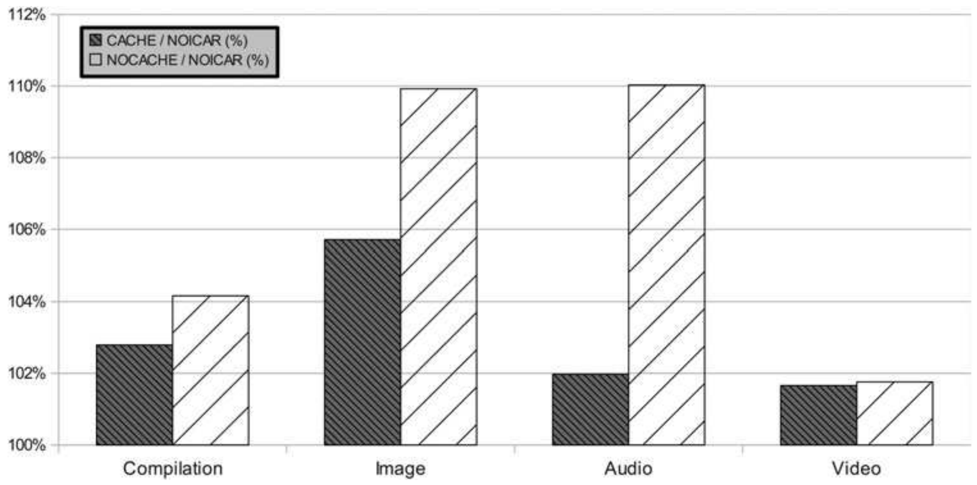

Fig. 8 *Time overhead for different operations*

6 Conclusions and future work

The presented solution increases computer system security by integrity checking of files that are vital for system operation. Compared to existing systems, two main innovative solutions are supplied by ICAR. First, the system supplies a mechanism that automatically restores the original content of the file from backup if an unauthorised modification has been detected. Second, ICAR uses write-protected media to store crucial files of the security system: cryptographic hashes, file backups and security binaries. It is proposed to use commonly available devices, such as CD-ROM disks or write-protected USB memories, as the write-protected storage. The use of relatively cheap hardware enables common application of ICAR with negligible required expenses.

The performance evaluation of the system showed that ICAR introduces performance degradation of approximately 10% for typical operations. The results indicate that ICAR operation should not be noticeable for a regular user. The system has been thoroughly tested to detect and fix potential bugs. Final tests and performance evaluation demonstrated that ICAR is a reliable and stable system.

We plan to improve ICAR administration and configuration as the main scope of future work. What is required now is the modification of ICAR security data each time the contents of protected files change. The operation requires recalculation of cryptographic hashes, copying of file backups and preparation of new CD-ROM disks or unlocking of USB memory write operations. The whole process may include advanced administrative operations and compliance with security restrictions. We plan to simplify ICAR administration and configuration by using the virtualisation technique, which is in the scope of intensive research. Various virtualisation-based security systems, such as HyperSpector, Livewire [18], VMWacher [19] and VMFence [20], have been designed and implemented in recent years.

Virtualisation distinguishes two software layers: virtual machine monitor and operating systems layer. Virtual machine monitor (called also hypervisor) enables access to hardware resources by many operating systems running concurrently. When a guest operating system requests a hardware resource, the request is intercepted and processed by the hypervisor. Operating systems host regular applications and access virtual machine resources in the read-only mode, whereas the hypervisor has full access rights to its resources. Using virtualisation, ICAR critical data may be stored in the hypervisor layer, which guarantees its safety from the operating system point of view and, simultaneously, simplifies configuration from the administrator point of view. Hypervisor enables safe data modification below the operating system, as the later treats the hypervisor as hardware resources.

Two models were designed for ICAR integration in the virtual machine architecture. The first model assumes that the ICAR module resides in system kernel whereas ICAR critical data (cryptographic hashes and file backups) are stored outside the protected operating system. The hypervisor is used to access critical data that is planned to be stored in a separate virtual machine due to design assumptions of the architecture. The solution requires relatively limited extensions of the hypervisor layer. In a more advanced architecture, the whole ICAR system is located inside the hypervisor layer. In this solution, operating system kernel is not modified and treats the security module as an extension of the hardware layer. The security module intercepts disk requests from the operating system and processes them to identify protected files and verify their contents. The main benefit of the solution is that ICAR operation is independent from the protected operating system, which enables the application of ICAR in different operating systems such as Linux, Windows, MacOS or others.

7 Acknowledgments

The work was supported in part by the Polish Ministry of Science and Higher Education under research project NN519-172-337.

8 References

1 Bace, R., Mell, P.: 'NIST special publication on intrusion detection systems'. DTIC Document, 2001

2 Preneel, B.: 'State-of-the-art ciphers for commercial applications', *Comput. Secur.*, 1999, **18**, (1), pp. 67–74

3 DeMara, R.F., Rocke, A.J.: 'Mitigation of network tampering using dynamic dispatch of mobile agents', *Comput. Secur.*, 2004, **23**, (1), pp. 31–42

4 Kaczmarek, J., Wrobel, M.: 'Modern approaches to file system integrity checking'. IEEE First Int. Conf. Information Technology, 2008

5 Kim, G.H., Spafford, E.H.: 'The design and implementation of tripwire: a file system integrity checker'. In: Proc. Second ACM Conf. Computer and Communications Security, ACM, 1994, pp. 18–29

6 Rocke, A.J., DeMara, R.F.: 'CONFIDANT: Collaborative object notification framework for insider defense using autonomous network transactions', *Autonom. Agents Multi-Agent Syst.*, 2006, **12**, (1), pp. 93–114

7 Zadok, E., Iyer, R., Joukov, N., Sivathanu, G., Wright, C.P.: 'On incremental file system development', *ACM Trans. Storage (TOS)*, 2006, **2**, (2), pp. 161–96

8 Borchardt, M., Maziero, C., Jamhour, E.: 'An architecture for on-the-fly file integrity checking'. Dependable Computing, 2003, pp. 117–126

9 Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: 'Linux security modules: General security support for the Linux kernel'. In: Proc. 11th USENIX Security Symposium, San Francisco, CA, 2002, vol. 2, p. 44

10 Schreuders, Z.C., McGill, T., Payne, C.: 'Empowering end users to confine their own applications: the results of a usability study comparing SELinux, AppArmor, and FBAC-LSM', *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 2011, **14**, (2), p. 19

11 da Silveira Serafim, V., Weber, R.F.: 'Restraining and repairing file system damage through file integrity control', *Comput. Secur.*, 2004, **23**, (1), pp. 52–62

12 Patil, S., Kashyap, A., Sivathanu, G., Zadok, E.: 'I3FS: An in-kernel integrity checker and intrusion detection file system'. In: Proc. 18th Annual Large Installation System Administration Conf. (LISA04), 2004

13 Ateniese, G., Burns, R., Curtmola, R., Herring, J., Khan, O., Kissner, L.: 'Remote data checking using provable data possession', *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 2011, **14**, (1), p. 12

14 Pennington, A.G., Griffin, J.L., Bucy, J.S., Strunk, J.D., Ganger, G.R.: 'Storage-based intrusion detection', *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 2010, **13**, (4), p. 30

15 Wang, X., Yu, H.: 'How to break MD5 and other hash functions'. Advances in Cryptology-EUROCRYPT 2005, 2005, pp. 561–561

16 Edwards, A., Jaeger, T., Zhang, X.: 'Maintaining the correctness of the Linux security modules framework'. In: Ottawa Linux Symposium, 2002, p. 223

17 Bishop, M., Dilger, M.: 'Checking for race conditions in file accesses', *Comput. Syst.*, 1996, **2**, (2), pp. 131–152

18 Garfinkel, T., Rosenblum, M.: 'A virtual machine introspection based architecture for intrusion detection'. In: Proc. Network and Distributed Systems Security Symposium, 2003

19 Jiang, X., Wang, X., Xu, D.: 'Stealthy malware detection and monitoring through VMM-based out-of-the-box semantic view reconstruction', *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 2010, **13**, (2), p. 12

20 Jin, H., Xiang, G., Zou, D., Zhao, F., Li, M., Yu, C.: 'A guest-transparent file integrity monitoring method in virtualization environment', *Comput. Math. Appl.*, 2010, **60**, (2), pp. 256–266