Process Authentication For High System Assurance

Hussain M.J. Almohri, *Member, IEEE*, Danfeng (Daphne) Yao, *Member, IEEE* and Dennis Kafura

Abstract—This paper addresses the need in modern operating system kernels for a process authentication mechanism, where a process of a user-level application proves its identity to the kernel. Process authentication is different from process identification. Identification is a way to describe a principal, and process IDs and names are identifiers for processes in an OS environment. However, forging and spoofing process identifiers by malware are easy to do. As a result, malware may abuse system resources violating system assurance. Yet, the problem of process or application authentication has not been systematically investigated in the literature.

We explain how process authentication differs from user authentication, its unique requirements, and technical challenges. We propose a lightweight secure application authentication model in which user-level applications are required to present proofs at run time to be authenticated to the kernel. We present a kernel protocol for the secure authentication of applications. To demonstrate the application of process authentication, we develop a system call monitoring framework for preventing unauthorized use or access of system resources. It verifies the identity of applications before completing the requested system calls. We implement and evaluate a prototype of our monitoring architecture in Linux as device drivers with no modification of the kernel. The results from our extensive performance evaluation shows that our prototype incurs very low overhead, indicating the feasibility of our approach for cryptographically authenticating applications in the operating system.

Index Terms—Operating system security, process authentication, applied cryptography, identification, authorization

1 Introduction

Operating system (OS) kernels often enforce minimal restrictions on the applications permitted to execute, resulting in the ability of malicious programs to abuse system resources. Stealthy malware running as stand-alone processes, once installed, can freely execute enjoying the privileges provided to the user account running the process. However, kernels are not designed to detect malicious behaviors, or identify malicious processes at runtime.

A well-known approach to protecting systems from malicious activities is through the deployment of mandatory access control (MAC) systems. Such systems often provide the kernel with access monitoring mechanisms as well as policy specification platforms. The user decides on the policies and the various access rights on system resources. Existing MAC systems such as SELinux [1], grsecurity [2], and AppArmor [3] enable the user (or the system administrator) to express detailed and powerful policies. These solutions are often implemented using the Linux Security Modules [4] to monitor access to selected system resources, and apply the specified policies to the corresponding processes.

- *H. M. J. Almohri, D. Yao, and D. Kafura are with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24060. Email: {almohri,danfeng,kafura}@cs.vt.edu.*

- *A preliminary version of the work appeared in the* Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY). *San Antonio, TX, USA. Feb. 2012.*

The above security solutions belong to the category of authorization. However, authorization mechanisms alone are not sufficient for achieving system assurance. Our thesis of this paper is to argue and demonstrate that the modern kernel must also have secure mechanisms for authenticating processes where the identity of a process can be proved. User authentication through techniques such as password or public-key cryptosystem is common in multi-user system or network environments. Many user authentication techniques exist in the literature. Yet, process authentication, i.e., how to prove a process is indeed what it claims to be, has never been systematically studied.

Process authentication is different and independent from process identification and requires stronger properties, for example unforgeability and anti-replay. In contrast, identification is a way to describe a principal. Process IDs and process names are identifiers for processes in an OS environment. Usually these process identifiers are generated by the system after examining the executable file names and installation paths of processes. This examination of executable file names and installation paths is the simplest form of process authentication. These simple authentication procedures are insecure against existential forgery attacks by malicious software, which we explain in details in Table 1. For example, AppArmor (based on the Linux Security Modules) recognizes processes through the application's installation path, based on which access rights are enforced. However, process authentication based on the installation path is weak and is subject to forgery attack as explained in Section 3. Without secure and robust process authentication, malware may claim to be a privileged or trustworthy process. As a result, advanced malware may abuse system resources violating system assurance.

Our work described in this paper examines the vulnerabilities in existing process identification, and points out (secure) process authentication as the missing link in achieving system security. Our work addresses how to authenticate processes at runtime and bind them to appropriate application identities. We aim to demonstrate that process authentication is a crucial step to prevent malicious processes from accessing and abusing system resources. In our solution, which is referred to by us as Authenticated Application (A2), applications with registered credentials can authenticate to the (trusted) kernel. The kernel can cryptographically verify the identity of applications. We point out that the differences in settings between process authentication and the conventional user authentication. Our description of the unique security and system engineering requirements for designing a process authentication solution is general. It is useful beyond our specific secret key based mechanism proposed. We present the design of a modified challenge-response protocol to securely authenticate applications by the kernel. Such an application authentication mechanism complements to the aforementioned process authorization solutions.

Our process authentication mechanism has important applications in protecting system resources and integrity, and preventing system access from being abused by malware. It can be either used alone in the OS as shown in this paper, or integrates with existing system authorization solutions such as SELinux [1] to support fine-grained process-level access control. In this paper, we demonstrate its practical application in preventing unauthorized system calls. We design and implement a system call monitoring tool in Linux that intercepts system calls made by the running processes and verifies application identities prior to granting the requests. The implementation prototype consists of two Linux kernel modules to securely authenticate applications and to verify their identities at the time of their requests for system call execution. Our implementation requires minimal modifications to commodity applications with nearly no modification to the kernel. Our evaluation results indicate the feasibility of our system call monitoring approach without a significant performance penalty.

Outline. Our model and overview are in the next section. We present the design of *Authenticated Application* framework in Section 3. In Section 4 we discuss our security guarantees and properties of the framework. Sections 5 and 6 discuss our implementation and experimental results. In Section 7 we present the related work and conclude in Section 8.

2 Model and Overview

An important step in enforcing application-level access rights on the running processes is to identify a process and properly bind it to the corresponding application. Existing mandatory access control (MAC) systems such as AppArmor and SELinux use installation paths and process names to identify processes and enforce appropriate access rights. However, such an identification mechanism is weak as summarized in Table 1. That is because the installation path and the process name are dynamic concepts and are subject to change by the user or by an attacker. Thus, to guarantee secure access control enforcement, a MAC system needs to rely on a strong identification model.

2.1 Security Model and Goals

Our basic trusted components are the kernel code and kernel's memory region. We assume that kernel does not contain any malicious code. Further, we assume that confidentiality and integrity of the kernel's memory is preserved. Such a trust can be established using existing techniques such as the Trusted Platform Module (TPM) [5], [6] at boot time, assuming exclusion of hardware attacks. Also, legitimate applications may be vulnerable and thus allow downloading malicious code. However, a malware cannot misuse a legitimate application without executing as a stand-alone process. Specifically, malicious code running within the boundary of a legitimate process (such as a malicious browser script or extension) is out of the scope of our work.

To establish a trust model, we categorize applications into two groups: legitimate and illegitimate. Legitimate applications are the ones that are authorized (and desired) to run in the system, while the illegitimate applications are not authorized. Therefore, we consider a binary level of trust for the applications. Legitimate applications that are registered by A2 have the *higher* level of trust, whereas any other application has the *lower* level of trust.

As described earlier, we do not assume the legitimate applications to be bug-free, however, our framework does not protect legitimate applications against attacks that do not result in running stand-alone malicious processes. We assume that legitimate applications' vulnerabilities may be leveraged by remote attackers (e.g., through a crafted malicious web page) to download malicious code that needs to run as a stand-alone process.

Within the scope of our security model, our research is guided by the following goals:

1) Detecting malicious (or unauthorized) code execution.
2) Preserving system's integrity by controlling the execution of untrusted applications (i.e., those applications that do not have registered keys).

Process Authentication	Property/Weakness	Possible Fix	Comparison With A2
Kernel process ID and executable path	Executable may be modified or replaced when freely allowing access to the file system.	Fixed file paths and locked installation directories.	A2 does not trust file path but uses file path to verify process claims to be legitimate by checking A2 registered credentials on the file path.
SELinux Labels	Based on executable names, can be reused by malicious processes, subject to replay attack and policy misconfiguration	Fixed file paths and locked installation directories.	A2 keeps track of registered apps and verifies their legitimacy by authenticating processes. SELinux does not authenticate processes.
File hashes	Need to be recomputed for authentication	None	Hashes may be used as A2 keys.
Developer-signed applications	May be forged using fake certificates, e.g., Flame and Flashback malware, also expensive to implement in kernel.	None	A2 implements a kernel level certification system that does not rely on certificate signing authorities, yet provides the desired level of security.
Authenticated system calls	Only capable of verifying system call usage integrity, but not a general application authentication.	Authenticating the whole application	A2 authenticates the whole process. The benefit is that an authenticated processes can prove its interactions with other processes, e.g. for authentication of interprocess communications.

TABLE 1
Comparison of the security in conventional process identification mechanisms to A2.

3) Preserving system's performance by developing a lightweight authentication model with the desired level of assurance.
4) Developing an authentication model with least usability penalties and minimal modifications to commodity operating systems.

Given our trust model, A2 aims to fundamentally distinguish legitimate processes from malicious ones, thus boosting the security guarantees of MAC systems. Later in Section 3.4 we discuss the integration of our application identities with an existing mandatory access control system.

2.2 Overview of Our Approach

We introduce the design and implementation of the *Authenticated Application* (A2) framework to provide strong authentication of applications. The core idea of A2 is a novel authentication model that is based on sharing unique symmetric keys between every installed application (that runs in the form of standalone processes) and the kernel. The strengths of this model is based on the cryptographic properties of symmetric keys generated by the key generators for standard encryption functions.

To authenticate and protect an application (a single executable program, also referred to as an executable), we sign it with a *secret application identity*, which is defined as follows:

Definition 1. *A secret application identity (SPI) is a string s of length n that is generated by a secure cryptographic function $f : n \rightarrow s$ such that s is random and computationally hard to regenerate given n. SPI is also referred to as application key in this paper.*

A secret application identity s has the following properties:

- Using s, the kernel establishes a provable identity of the executable such that this identity may not be spoofed, e.g. by replaying a legitimate identity to the kernel.
- s is unique and identifies the processes of a single executable. The executable identified by s is called the *owner* of s.
- For one executable, there is no more than one identity s.
- s shall not be available to unauthorized processes.
- s is given to the executable at the registration time (Section 3.2). If the executable is reinstalled on the file system, it's given an identity s' such that $s' \neq s$.

The requirements of the function f in Definition 1 ensure that the secret s is computationally hard to guess. The secret application identity must be unforgeable to provide high assurance about the identity of a running process and strongly bind it to its executable code. The properties of the secret application identity, guarantee that the owner of s is unforgeable and resistant to various attacks such as stealing the secret from the file system.

To protect application identities from stealing attacks, we introduce a protection mechanism referred to as the *code capsule* that is capable of securely protecting an application's SPI from file system attacks. We define a code capsule as follows:

Definition 2. *A code capsule C_s is a piece of executable code appended with a secret application identity s that is unique and is verifiable by the kernel. A code capsule is not accessible by any user process except a single kernel helper process whose identity is securely verifiable by the kernel.*

Code capsules serve two major purposes. One is to protect the secret application identity s from being accessed, and thus revealed to unauthorized processes through the file system. The other is to bind s with the executable code file, which is later used to verify the identities of the running processes by the kernel. Code capsules are accessible and maintained by a kernel helper processes described next.

2.3 Overview of our framework

In the diagram of Figure 2.1, we present a high level conceptual process of protection against malware that can effectively identify legitimate processes and enforce application-level access rights with the assistance of a mandatory access control system. In this process, three major components need to participate. First, a classification component decides on the legitimacy of the executable code. Next, an identification component must register legitimate applications identified in the first step and prove their identities to the following component. Finally, access policies are specified and used to monitor the execution of a process and enforce access rights within a mandatory access control system. While the classification of applications and policy specification are critical steps, we realize the lack of a proper identification and binding mechanism that can complete the protection process. Hence, throughout this paper we present our solution for the identification component in the form of the following steps:

1) **Application key registration.** We generate a unique secret key for each legitimate application.
2) **Application authentication.** We use the provided application key to securely authenticate the application code which contains the key and produce identity tokens.
3) **Execution monitoring.** We monitor the execution of the processes to limit the activities of unauthenticated ones.
4) **Identification and binding.** We identify a registered application process using generated identity tokens and bind them to the corresponding application access rights.

In the key registration step, we provide legitimate applications with appropriate application keys. In the application authentication, we use the provided application key to securely authenticate the application code which contains the key and produce identity tokens. In the final step, we identify process using generated identity tokens and bind them to the corresponding application access rights.

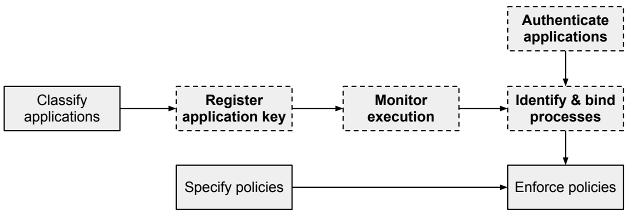

Fig. 2.1. A high level conceptual process of protection against malware.

In A2, each legitimate application is supplied with a secret key that is only accessible by the application code and the kernel. At the time of creating a process, the application's secret key is used by the process to authenticate itself to the kernel. Once the process is securely authenticated, the kernel can assure its identity relying on the strong properties [7] of the cryptographic hash functions.

In the authentication model of A2, applications are recognized as individual principals. Legitimate applications (that are assigned a key) are the most privileged applications while unregistered applications (that are unable to identify themselves) are restricted and considered potentially malicious. This identification mechanism provides a secure sandbox for the potentially malicious processes and isolates them from authenticated processes. It is necessary to allow the creation of any process regardless of its identity. This is to enable any application to authenticate itself at runtime in order to provide proof of identity. In addition, this strategy results in uncovering stealthy malware as soon as it interacts with the kernel through a monitored system call.

3 Authenticated Application Framework

Our Authenticated Application (A2) framework enables the authentication of applications with high assurance. It consists of three main components: *Trusted Key Registrar*, *Authenticator* and *Service Access Monitor (SAM)* depicted in Figure 2.2. We implement the Authenticator and SAM as Linux kernel modules without modifying the kernel (see Section 5). We describe the functions of our components in the following.

Trusted Key Registrar is a kernel helper responsible for installing a key for the application and registering the application with the kernel. The application interacts with the trusted key registrar to receive a secret key. The trusted key registrar stores the same key and registers it for the corresponding application within a secure storage to be used for the authentication of the processes at runtime.

Authenticator is responsible for authenticating a process when it first loads. The Authenticator generates identity tokens (defined in Section 3.1.1) based on a token generation protocol.

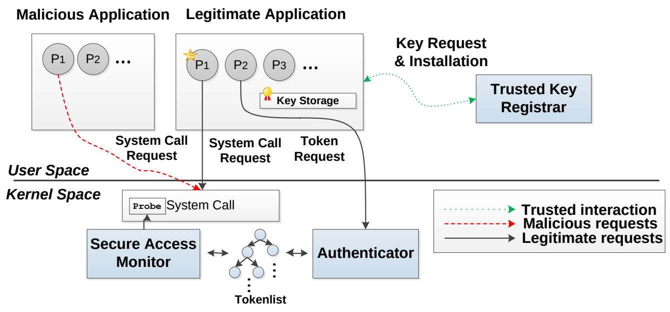

Fig. 2.2. The access to selected system calls is monitored by A2. P_i denotes an application process.

Service Access Monitor (SAM) is responsible for verifying the tokens at runtime and enforce application-level access rights. Since the tokens are maintained by the Authenticator, SAM realizes its task by coordinating with the Authenticator through a shared data structure. SAM enforces application-level access rights based on a user-specified application policy.

3.1 Secure Authentication of Applications

In order to identify a running process and bind it to the corresponding application [1], the process must be able to prove its identity to the trusted kernel using the application's secret key. The authentication is summarized in three generic steps. First, the kernel needs to send a random nonce to the application process. The process produces the hash-based message authentication code (HMAC) using the nonce and the secret key and returns the nonce back to the kernel. The kernel regenerates the HMAC and compares it to the value returned by the application.

Implementing the authentication protocol in kernel is not trivial. A technical challenge is how to support the secure communication between an application and the kernel in an efficient way. The first design choice is that the kernel directly accesses the application's key and verifies its identity provided that the key is stored in a predefined location. However, this method does not provide the security level that is needed in order to establish a strong identification. The location of the key can be either defined in memory or the file system. Defining the key in the memory imposes additional risk to stealing the key as well as causing complexity of maintaining the key location. The alternative design would be separating the key in a restricted key storage to be used by the kernel at the authentication time. This design choice is not adequate since it is not possible to securely bind a running process to the correct key file at runtime. Therefore, we use an authentication protocol that can be executed on a socket file between the process and the kernel. This method can be realized using a memory-based socket (or a shared memory region) such as the `/proc` file system [8]. The advantage of using the `/proc` file system is that it is conveniently accessible by kernel device drivers and is under the complete control of the kernel. More details on the implementation can be found in Section 5.

1. A piece of executable code that runs in at least one stand-alone process.

3.1.1 *Token Generation Protocol*

Our authentication protocol is used to generate identity tokens for legitimate applications. The identity tokens are later used to identify the processes when interacting with the kernel through the system calls. The identity tokens are needed since the authentication and verification of identity are separated in A2. That is, the authentication is only performed at the time of creating a new process. When the created process accesses a monitored system call, the identity verification takes places by searching for an identity token. Beside providing the needed security, the separation of authentication and identity verification improves the system call monitoring overhead (see Section 6).

Our token generation protocol (TGP) is used to authenticate individual processes based on the keys of the corresponding applications. TGP follows a standard challenge-response authentication protocol used in secure data communications over a network. We modify the standard protocol to include necessary kernel-side verifications as well as including token generation steps.

TGP is used to prove the identity of a process to the kernel. TGP must assure that no process can impersonate other processes using a replay attack provided that the application keys remains secret. Further, a registered application process must be able to successfully authenticate itself, if it was not previously authenticated. TGP also assures that no process can launch a denial of service attack on the kernel. Thus, we say that the identity of a process is proved if the process has a key with compliance to Definition **??** and successfully executes TGP such that a token is generated.

In the following, we formally define a *registered application*, an *identity token* and the *Authenticator*:

Definition 3. *A registered application is a piece of executable code that runs in the form of one or more standalone processes and is issued a secret key by the kernel.*

Definition 4. *An identity token is a tuple* `(app, pid)` *where* `app` *is the name of a registered application and* `pid` *is the kernel process ID of the process created by* `app`.

The identity token is unique and binds to a single process. It is valid until the termination of the process and is generated by the Authenticator but it is readable by the Secure Access Monitor.

Definition 5. *The Authenticator is a kernel module that implements the token generation protocol and is responsible for creating and maintaining identity tokens for registered applications.*

Let `A` denote the Authenticator module and `p` be a user process where `p.pid` is `p`'s process identification and `p.app` is `p`'s application name. The function `malicious(p)` would log `p` as malicious and may take any necessary action depending on the security policies. Additionally, `tgenerate(p)` is a function to generate a token `tk` for the process `p`. Finally, `arequest(p.app)` is a function used by `p` to send an authentication request to `A`. The steps of TGP are as follows. In each step the actions of (if there is any) `A` and `p` are specified.

Token Generation Protocol:

1) `p`: Sends `arequest(p.app)` to `A`.
2) `A`: Receives and verifies the request: 2.1
   a) Verifies if the requesting application has a registered key. Otherwise, `malicious(p)`.
   b) Verifies if `p` has already established a token. If so, `malicious(p)`.
   c) `A` Limits the authentication requests in order to prohibit the applications to flood the kernel intentionally or due to an unintended software bug. Thus, `A` verifies if count(`p`) < limit(`p`). If the limit check was failed, `malicious(p)`. Each application has a specified limit of simultaneous requests. This is set as part of `A`'s verification policy.
   d) Generates a random nonce s and sends it to `p`. Additionally, `A` sets a timer t for the string to expire if there was no response from `p`. The time frame to expire t needs to be very short as this authentication is performed without networking inaccuracies. We only need the timer for the case that the process crashed or was killed and did not continue the authentication.
3) `p`: Generates the hash-based message authentication code (HMAC) h = HMAC(s,`p.pid`, k) (where k is `p`'s secret key) and sends it to `A`.
4) `A`: If t has expired, the authentication request is discarded. If `p` is still executing, it will be terminated to prevent a race condition.
5) `A`: Computes h' = HMAC(s,`p.pid`, k). If $h = h'$, then `tk = tgenerate(p)`. `tk` is valid until the termination of `p`. Otherwise, `malicious(p)`.
6) `A`: Stores `tk` in a data structure `tokenlist` that is only readable by the verification module (i.e. SAM).

TGP prevents replay attacks since h = HMAC(s,`p.pid`, k) may not be accepted if received from any process other than `p`. Denial of service attacks are also prevented since TGP limits the number of requests that may be received from a single process in Step 2.3. Further, code injection attacks are avoided in TGP. An attacker can write a malicious code to the shared memory socket. The code may be read either by `p` or `A`. However, since the communication between `p` and `A` has a definite length (either the length of the request, or the nonce or the HMAC), exploiting a buffer overflow is avoided by verifying the string length.

3.2 Application Registration

Prior to performing any authentication, it is necessary for the kernel to generate and register the secret keys for legitimate applications. As shown in Figure 3.2, the secret key must be registered by the kernel and must be stored in the application's code capsule (Section 2.2). To protect the key from being stolen by static analysis of the executable code, A2 restricts read access to executable codes by any application. Further, the installed key is only associated with one installation instance and is not valid once the application is re-installed.

The trusted registrar uses the registration function as defined below.

Definition 6. *The registration function $\rho : E, s \rightarrow C_s$ where E is a string containing the executable code and s is the secret application identity generated by a secure key generation function f. The function ρ produces the string $C_s = E||s$, which is a code capsule protected by the kernel.*

To register the key, we design a trusted key registrar that is used to register applications' keys in the kernel and the application. The trusted registrar itself is authenticated and identified through the TGP protocol with a special key installed manually. The steps taken by the trusted registrar are as follows:

1) The application is started for the first time and requests a key from the trusted registrar.
2) The trusted registrar verifies if the application was previously issued a key and if the application is designated legitimate either by the user or after an application certification process.
3) If verification passed, generates a key s (in compliance with Definition 1) and runs the function

ρ given the application's executable E to produce a the corresponding code capsule C_s. Large applications may consist of several executable files. We register each executable file that may create at least one independent process with a unique s. The purpose of having a unique s for each executable is to be able to correctly identify each running process and bind it to its executable code.

4) Lastly, s is stored in a secured key storage that is not accessible by any other process. This accessibility restriction is enforced in the kernel.

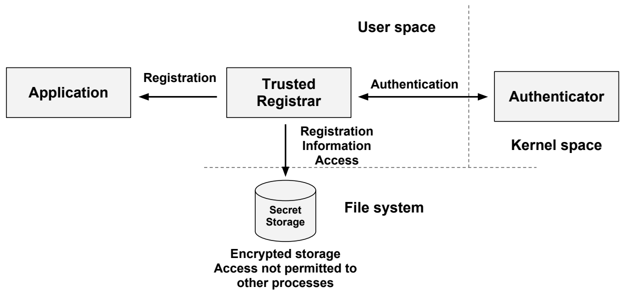

Fig. 3.1. The trusted registrar interacts with applications to register their secrets. It authenticates itself before performing any task.

Notice that application keys are securely stored in the file system. The keys may not be reused by attackers (in a key replay scenario), as the read access to code capsules is restricted to the application that owns the key.

3.3 Authentication of Commodity Applications

The authentication protocol (described in Section 3.1.1) can establish a proof of identity for any user process provided that the process executes the protocol before performing any task. This approach would require modification of a large number of commodity applications to implement the protocol and poses maintainability and consistency difficulties. To overcome the difficulties associated with using the protocol for commodity applications, we deign a middleware software to perform the authentication on behalf of the application. By performing the authentication on behalf of the application, the key is no more available to the application itself. As a benefit of this design, the problem of leaking the key from the application is resolved.

We authenticate commodity applications using a kernel helper referred to as *secret verifier (SV)* that is trusted by the kernel (see Figure 3.3). The kernel establishes the trust with SV similar to the trusted registrar through the authentication protocol. The helper shares a memory region with the kernel to exchange verification messages.

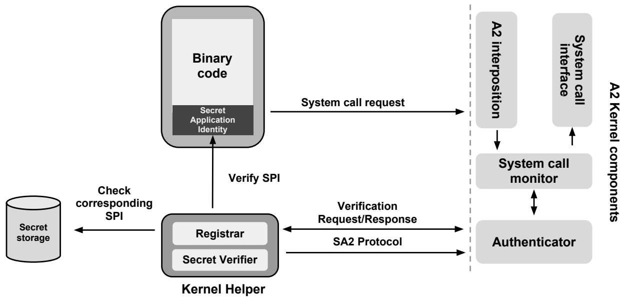

Fig. 3.2. Kernel monitors system calls and if needed asks for verification of the identify of a process.

To verify the identity of a running process, the kernel checks if the process has already been verified by searching the list of valid tokens. If the token is not found, the kernel sends a message to SV containing the path to its executable. SV verifies if the application is registered with a secret. If a corresponding secret was found, SV reads the executable code indicated by the path passed by the kernel. If the executable is a code capsule with the same registered secret, then the verification succeeds and a message is passed back to the kernel. The kernel completes the authentication by adding the token in the list of valid tokens and exempts the process from future authentications.

3.4 Verification of Identities

Our token generation protocol is used to securely authenticate running processes and generate identity tokens. These identity tokens are used by the Secure Access Monitor to validate application access rights at runtime and authorize the use of system calls accordingly. SAM's main functionality is to monitor designated system calls and verify the identity of processes for other cooperating kernel components such as a MAC system. The system call monitoring mechanism assists a MAC system to enforce its specified policies using the provided identity verification. Our monitoring mechanism is general enough to monitor any desired kernel function.

Our identity tokens are integrated with existing MAC systems. For instance, we modified AppArmor to make use of the identity tokens generated by the Authenticator and verified by SAM. Using the identity tokens, AppArmor can strongly bind a process to the application profile and enforce appropriate access rights. Further details on the integration of a MAC system with our framework are left for future work.

The Secure Access Monitor can be integrated and with a policy specification language to benefit from existing scholarly work in this area [9], [10], [11], [12]. A policy file must specify application categories, system call names, and optionally permitted system call usage frequencies. SAM uses the binary decision for allowing or disallowing the use of a system call.

In this file, permitted frequency is omitted indicating that the corresponding system call usage frequency is unlimited.

4 Discussion

In this section we present the properties of the A2 framework. We discuss in detail the security guarantees that are achieved using our identification model.

Strong application identities. Our presented application identification model is strong since it uses cryptographic keys that are kept secret and protected by the A2 framework. The secret key of an application is unforgeable as it is computationally hard for a malware to find the key. Moreover, the token generation protocol enables transparent and secure communication between applications and the kernels relying on the properties of the cryptographic hash functions.

Application isolation and access rights. In the A2 framework, we fully sandbox undesired processes. Our sandbox relies on the fact that malicious applications fail to authenticate to the kernel and thus are prevented from using most critical system calls. Moreover, such processes are exposed to the kernel when trying to interact with it without the presence of a valid token. This makes A2 a powerful tool to find malware that was dropped by other applications by various means such as through drive-by-download. Although a legitimate application such as a web browser may allow malware to be downloaded, A2 prevents the downloaded malicious code to reach its ultimate goal.

Effective application-level access right enforcement is another advantage of A2. When access rights are simply enforced based on application names, installation paths or solely according to the user's access rights, it is difficult to securely control the activities of various processes. The strong binding between a process and its executable code (i.e. the application) enables the kernel to treat applications as principals and enforce appropriate access rights. In this case, when a simple text editor application does not need to use the networking system calls, A2 prevents it from doing so. Such policies help in deciding desired and undesired behavior of an applications to prevent intentional or unintentional misbehavior.

Key protection. Application binaries provide secure storages for application keys. To protect the secret key from being revealed to other applications, A2 restricts read access to registered application binaries. In principle, malware can also steal a key from application's memory at runtime, using a buffer overflow. However, as we restrict the activities of unidentified processes, such an attack cannot be achieved. Further, a file system encryption can be used to encrypt the binary code of an application using a master key known to the kernel. This can be done by implementing a kernel module, which will be responsible for encryption and decryption of the application binaries without the need to modify the kernel. We leave this issue for our future work.

A2 is capable of identifying interpreted programs running as stand-alone processes. For instance, a Java executable runs as a separate process named Java. In this case, the program can have a unique key and be registered in the installation time. Each program can authenticate itself independently using our framework. Other interpreted languages such as Adobe Action Script and Word document macros are out of the scope of our model.

Programs that are executed as part of other programs (for example using `execve`) are also identifiable using A2. In our model, a process does not inherit its access rights from its parent process or the application that was responsible for ordering the execution. For example, programs often run using a shell terminal. It is the responsibility of the process to perform the authentication and identify itself.

Extensible applications such as Internet browsers frequently run other code such as JavaScript programs. In a browser, extensions and add-ons may not have the same trust level or access rights of the browser. The goal of A2 is to fundamentally distinguish a legitimate browser from a malicious one, which is well achieved. Thus, detection of malicious code running in a browser is out of the scope of A2.

5 Implementation

We realize a prototype of the A2 framework in the Linux Operating System (Debian 2.6.32). The two main components of A2 (Authenticator and SAM) are developed as kernel device drivers (modules). We have implemented SAM and the Authenticator as Linux kernel modules. These two modules can be loaded using `root` privileges as needed. SAM depends on the Authenticator in order to verify the tokens. The trusted key registrar and the secret verifier are implemented as kernel helpers that run in user mode. All the components are written in standard C. In the following we describe the implementation of our prototype.

5.1 Trusted Key Registrar and Secret Verifier

The trusted key registrar is implemented as a user process, for several reasons. First, such a functionality should not be included as part of the kernel code to preserve the maintainability of kernel code. Second, it is more convenient to implement the registrar using standard user-level libraries, which are not provided for kernel code. Third, by implementing the registrar as a user process, we avoid unnecessary context switches to register an application.

Provided that, the registrar runs as a user process, the kernel must ensure that the secret information is not communicated with unauthorized user processes.

To achieve this, we hard code a secret application identity for the registrar in its code. The trusted registrar uses the secret to authenticate itself to the kernel. This authentication is through the use of our *Secure Application Authentication* protocol (Section 3.1.1), which is executed between the kernel and a user process.

Application registration is performed by the trusted registrar upon user's request. As depicted in Figure 3.2, the trusted registrar is a kernel helper application that interacts with the Authenticator to prove its identity prior to performing its task.

The trusted registrar interacts with an application to provide it with a secret and stores the secret in a secure storage on the file system. The secure storage of the keys is a file with restricted access to the trusted key registrar process only. This restriction is implemented as preventing the `open` system call on the secret storage file unless the process is authenticated as the trusted key registrar. The kernel can provably identify the authorized process since the trusted key registrar is issued a special key at the development time, which is also known to the authenticator module.

The secret verified process is similarly authenticated. Both the trusted key registrar and the secret verifier are stored in code capsules that are protected by the kernel. The trusted key registrar has access to application executables to respond to the kernel's requests. These requests are sent to the secret verifier process through the `/proc` file system. We also secure the communication channel by restricting the `open` system call to all other processes.

5.2 Authenticator

The Authenticator module uses the Linux kernel Cryptographic API [13] to perform the HMAC operations using a number of supported hashing algorithms. The Authenticator communicates with the user space using the `/proc` file system, which is a memory-based file system controlled by the kernel. A protocol file is created by the Authenticator in the `/proc` file system and is made accessible to all running processes. The protocol remains secure since the communication is performed using the HMAC (Section 3.1.1).

We define two functions for reading and writing operations to the protocol file in `/proc` file system. The `read_protocol_file` function is executed when the user reads the file. For writing to the challenge file, we define the function `write_protocol_file`. In this function, our module reads the data that is written by the user. The Authenticator only responds to one request, which is the generation of a token by a running process. The request verifications described in Section 3.1 is performed before the protocol can proceed.

Our implementation of the Authenticator is able to accept multiple requests from multiple processes using the same `/proc` protocol file. For each process, only one request is served at a time. We define an `tokenlist` data structure that stores the secret keys of all authorized applications. When the Authenticator receives a request, it verifies the availability of a key for the requesting application by searching the `tokenlist`. If a key was found, the Authenticator continues the rest of the verification process and then sends a random nonce back to the process. At this time we set a flag that an authentication process for the requesting application is ongoing. When the hash of the nonce arrives, the Authenticator verifies that hash and in case the hash was correct, a token for the currently communicating application is generated and is kept in the `tokenlist`.

5.3 Secure Access Monitor

Secure Access Monitor (SAM) and the Authenticator communicate via a shared data structure in the memory that holds the valid tokens. This data structure is only visible to SAM and the Authenticator. To verify a process' identity, SAM searches through a list of valid tokens that are maintained by the Authenticator.

We implement the list of tokens as a sequential dynamic array. In our experiments, under a normal use, the number of running processes was under 100 process. As it is shown in the evaluations, searching the list of tokens did not have a significant overhead in a normal usage. However, in order to improve the overhead, one can implement the list of tokens as a red-black tree (a special type of balanced binary search tree [14]) that has a search complexity of O(log n) where n is the number of tokens in memory.

To avoid the need to modify the kernel, SAM uses the `kprobe` API to hook into system calls and monitor process activities. Although the probes introduce extra overhead, the produced overhead does not cause considerable latencies to application's functionality, limited by an upper-bound of 3 times more overhead (see Section 6).

To provide a more efficient implementation of SAM, we modified the Linux kernel to implement a faster system call tracing method. In this implementation, we modify kernel's entry assembly code to perform the verification of identities before the system call takes place. As the kernel prepares for jumping to the address of the requested system call, we place a jump to the address of our kernel function that implements SAM. We store all the necessary information before the jump and send the system call number and the process information to SAM's kernel function. The system call may be allowed or disallowed according to the value returned from our kernel function. After the return, we check the return value and either jump to the desired system call function or execute an exit code to user mode.

6 Performance Evaluation

The strong security guarantees provided by our A2 framework incur computational and management overhead in the operating system. In order to assess the efficiency of our framework, we answer the following questions in our experiments:

- What is the system call overhead caused by A2 as a result of verifying application's identity at the time of making system calls?
- How does A2 impact the overall system performance?

In our evaluation, we design a micro-benchmark to assess the system call overhead. In order to assess the overall system performance penalty due to A2, we use the lmbench micro-benchmark [15]. For our analysis we used a VirtualBox virtual machine (VM) with Ubuntu 10.04 (32-bit) installed on it. We allowed the VM to use up to 1 GB of memory. At the time of our analysis a normal load of user programs were launched.

To measure the overhead caused by SAM on handling the system calls we designed a set of programs to make extensive use of a collection of system calls. We let SAM to monitor a collection of seven system calls containing frequently used system calls such as `read` and less frequently used system calls such as `getpid`.

Each of our benchmarking programs are given a system call and a number of iterations. We set each program to make calls to the specified system calls for 150,000 iterations. The programs do not perform any other tasks. We measure the time spent in kernel for the system calls made by each program in three experimental settings. First, we measure the system without running any of our kernel modules. Next, we run A2 modules, without performing any verifications by SAM (i.e. searching the `tokenlist`). In the final experiment, SAM verifies the `tokenlist` with a total of 300 stored tokens. The results of our experiments are shown in Figure 6.1. On average, the system call overhead is 3 times more than the baseline latency.

Based on our experimental results, the major latency is caused by the installed `probes` in the kernel functions. That is because, the average extra latency caused by the verification of the `tokenlist` (that already contains a total of 300 tokens) is 29.03%.

We measured the overall system performance downgrade due to A2, in another set of three experiments. For these experiments, we used the lmbench micro-benchmark [15]. This benchmark provides performance analysis for various system functions such as networking and file system. We include the results for signal handler, pipe communications, UNIX socket transactions, process creation and termination using `fork` and `exit`, and process creation using `execve`. As shown in Figure 6.2, the extra latencies caused by A2 modules are not significant. On average, there is an increase of 26.76% in processing time and the maximum latency is for UNIX socket transactions for an overhead of 54.65%.

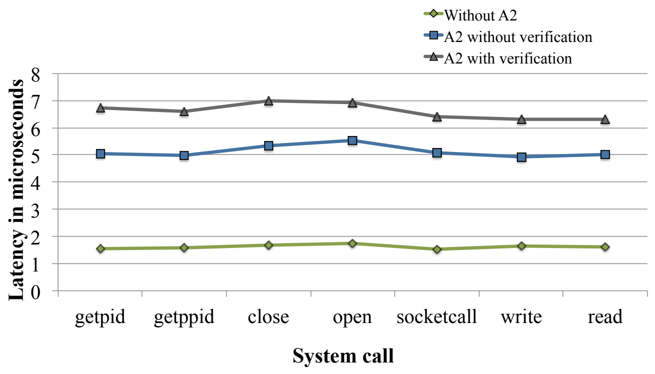

Fig. 6.1. System call overhead measured in three experiments: No A2 modules running, A2 without any verification, and A2 running and performing verification on a list of 300 tokens.

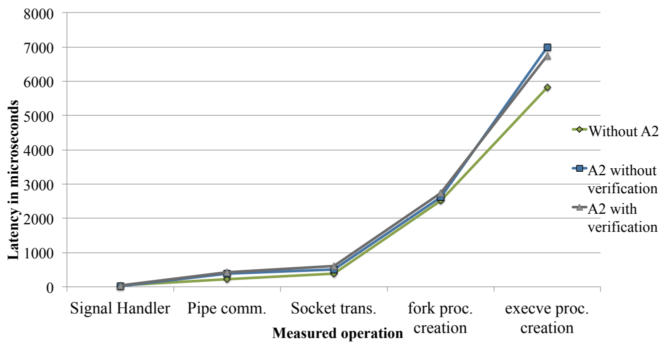

Fig. 6.2. The latency caused by A2 modules for UNIX socket transactions and process creations.

6.1 Scalability Evaluation

Figure 6.3 shows an experiment to measure the scalability of A2 modules. We examined the average overhead associated with verifying the `tokenlist` with various numbers of tokens in the list. With under 1000 tokens generated, the latency caused is a fraction of a second (50 microseconds), which is a negligible overhead. Our token lifetime is equal to the lifetime of the process it belongs to. Given this lifetime model, having 1,000 tokens means 1,000 different processes authenticated in the current session. We do not expect a normal load would have more than 1,000 processes running simultaneously. However, even with having 20,000 tokens the latency reaches up to 350 microseconds.

Figure 6.4 shows an experiment for measuring the total overhead caused in a 5-minute-long trace for searching for a token based on a global monitoring frequency f. With $f = 50$, an inspection is performed for every 50 calls to `sys_socketcall`. The corresponding average overhead associated with our access verification is quite small and efficient, which is under 5 seconds over a five-minute period. Higher monitoring frequencies incur higher latency. A higher monitoring frequency provides a stronger secure guarantee for the overall system. (If $f = 1$, SAM monitors every single system call request for monitored system calls.) Therefore, the malware behavior can be detected immediately from the first call made. In contrast if $f > 1$, SAM skips f number of system calls before it verifies the `tokenlist` to increase the performance. In case of restricting access to highly critical system calls, the malware may be able to manage to fall into the interval after the last `tokenlist` verification and before the next verification. f must be as low as possible, especially for those system calls that are least frequent such as `sys_kill`. The f value may be slightly increased for highly frequent system calls like `sys_open`. On the other hand, even with a $f = 1$ for less frequent system calls, the system has a negligible system performance downgrade.

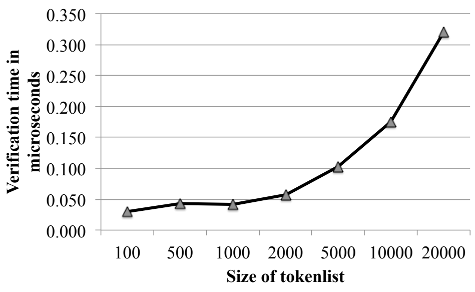

Fig. 6.3. Verification time associated with one request in milliseconds increases with the size of token list.

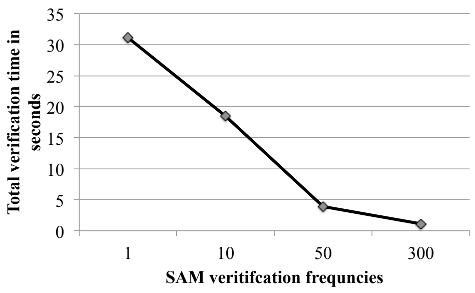

Fig. 6.4. Total latency for a five minute experiment for various values of f.

Our results show efficient system call performance without a significant penalty due to our monitoring architecture. While performing the experiments inside a virtual machine with limited resources, we did not notice the imposed latencies as end users. Moreover, the token generation protocol does not impose further performance penalties as it is not part of the monitoring process. This protocol is only executed once at the time of creation of a process and the generated identification token can be subsequently used in the system call monitoring process.

7 Related Work

Existing work on protecting system's integrity is studied in the form of program integrity measurement techniques [16], [17], [18], information flow integrity [19], and mandatory access control [3], [1].

The Integrity Measurement Architecture (IMA) is a mechanism to provide attestations about the integrity of the kernel and the running programs for a trusted remote verifier [18]. In this architecture, the kernel maintains an aggregation of user programs' and files' checksums (i.e., the hash of the file's contents) in the memory. The integrity of the list in the kernel's memory is maintained using TPM. The checksums of user programs' are communicated to the remote party to perform the necessary verification. In [17] a similar approach is taken to apply IMA on mobile operating systems.

The work in [18] is enhanced by PRIMA [19] to take advantage of information flow integrity for verifying and controlling user programs' inputs. Specifically, PRIMA forces the flow from a low integrity program to a higher integrity program to pass through a filter.

ReDAS approaches the problem by providing attestation of dynamic program features to remote parties [16]. In the proposed methods, the integrity of the kernel is assumed to be established based on TPM. Then, the kernel keeps track of dynamic program features by a static analysis of the program binary. For instance, ReDAS makes sure that the return address of a function points to the instruction following the `call` instruction.

Mandatory access control (MAC) systems specify fine-grained policies for the installed applications. These policies are typically administered by a power user (such as the `root` user in UNIX-based systems) to control the behavior of the applications. A well-known MAC system is SELinux [1]. SELinux assigns applications to domains and tags executable files with their appropriate domain information. At runtime, SELinux monitors the access by all processes and enforces the predefined access policies by binding the process to an appropriate domain and deciding on the right policies. An alternative to SELinux, gr-security [20] provides sophisticated memory protection mechanisms such as enforcing read-only memory pages.

Existing MAC systems differ with our proposed model in that they rely on standard Linux user identities to decide on the access rights. We instead, provide a strong application identification mechanism (Section 3.1.1) that is independent of a particular user identity and does not rely on dynamic features such as a process ID.

In [21], [22], the authors propose the use of message authentication code in monitoring system calls. By using an automated method binary rewriting, all the system calls functions calls are modified to include a message authentication code as extra arguments. The message authentication code is generated using a key that is available to the kernel. At runtime, the kernel uses the key to verify the code against the actual system call made by the application in order to detect possible modifications to the application's behavior. The presented work is limited to providing identities (the HMAC) to individual function calls to system calls in an application. Thus, it does not provide an identity to the application itself.

System call monitoring is an ongoing research towards protection against malware [23] mostly focused on the use of virtual machine monitors (VMM) to monitor system calls [24], [25]. We do not implement the components of A2 within a VMM to avoid the semantic gap introduced. This semantic gap prevents A2 from close monitoring of the process activities as well as proper identification of the processes. Futher, VMM does not provide additional security guarantees since we preserve the security of the kernel and the authenticated processes by limiting the activities of malicious processes.

Application sandbox is a mechanism to allow execution of untrusted code on protected hosts. Recent sandbox proposals include Vx32 [26], UserFS [27], and BLADE [28]. Application sandbox methods are useful for our A2 framework. Systems such as UserFS that allow temporary secure execution of an untrusted code can be coupled with A2 to perform the necessary application checking before registering the application as a legitimate application. In a future direction, we will examine the integration of A2 with an application sandbox to provide a more comprehensive solution.

8 Conclusions

We presented a novel authentication model that provides strong and unforgeable application identities and binds the processes to their corresponding applications at runtime. Our model is combined with our system call monitoring architecture that verifies identities of the processes. This model resolves the problem of detecting the identity and the origins of running processes using a trusted kernel. In the A2 framework, malicious processes are completely isolated to prevent them from attacking other processes or achieving any attack goals. The authentication model of A2 is simple to implement and is highly portable. We introduced the idea of an authentication protocol between a user application and the kernel. To protect commodity applications, we introduced a method to authenticate applications without the need to modify the applications. Our middleware contributes to the portability and usability of our framework. Our evaluation results indicate the feasibility of using cryptography for the purpose of identifying running processes. We achieve this result by separating the authentication from the monitoring. Therefore, there is virtually no performance penalty due to the use of cryptographic functions.

References

[1] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the Linux operating system," in *Proceedings of the 2001 USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, 2001.

[2] "grsecurity," http://www.grsecurity.net/.

[3] Z. M. Hong Chen, Ninghui Li, "Analyzing and comparing the protection quality of security enhanced operating systems," in *Proceedings of the 16^{th} Annual Network & Distributed System Security Symposium*, 2009.

[4] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security module framework," in *Proceedings of the 11^{th} Ottawa Linux Symposium*, 2002.

[5] S. W. Smith, *Trusted Computing Platforms: Design and Applications*. Secaucus, NJ, USA: Springer-Verlag, 2004.

[6] D. Stefan, C. Wu, D. Yao, and G. Xu, "Knowing where your input is from: Kernel-level provenance verification," in *Proceedings of the 8^{th} International Conference on Applied Cryptography and Network Security (ACNS)*. Springer-Verlag, 2010, pp. 71–87.

[7] J. A. Buchmann and J. A. Buchmann, "Cryptographic Hash Functions," in *Introduction to Cryptography*, ser. Undergraduate Texts in Mathematics, S. Axler, F. W. Gehring, and K. A. Ribet, Eds. Springer New York, 2004, pp. 235–248.

[8] M. T. Jones, "Access the Linux kernel using the `/proc` filesystem," 2006, *http://www.ibm.com/developerworks/linux/library/l-proc.html*.

[9] C. Ardagna, E. Damiani, S. D. C. di Vimercati, and P. Samarati, "XML-based access control languages," *Information Security Technical Report*, vol. 9, no. 3, pp. 35 – 46, 2004.

[10] Q. Ni and E. Bertino, "xfACL: an extensible functional language for access control," in *Proceedings of the 16^{th} ACM Symposium on Access control Models and Technologies*, ser. SACMAT '11. New York, NY, USA: ACM, 2011, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1998441.1998451

[11] S. D. C. D. Vimercati, S. Foresti, P. Samarati, and S. Jajodia, "Access control policies and languages," *International Journal of Computational Science and Engineering*, vol. 3, pp. 94–102, November 2007.

[12] T. Wobber, A. Yumerefendi, M. Abadi, A. Birrell, and D. R. Simon, "Authorizing applications in singularity," in *Proceedings of the 2^{nd} European Conference on Computer Systems*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 355–368.

[13] J.-L. Cooke and D. Bryson, "Strong cryptography in the Linux kernel," in *Proceedings of the 2003 Linux Symposium*, 2003, pp. 139–144.

[14] D. P. Bovet and M. Cesati, *Understanding the linux kernel*. O'Reilly, 2006.

[15] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23. [Online]. Available: http://portal.acm.org/citation.cfm?id=1268299.1268322

[16] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *Dependable Systems and Networks*, 2009, pp. 115–124.

[17] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger, "Measuring integrity on mobile phone systems," in *Proceedings of the 13th ACM Symposium on Access control Models and Technologies*, ser. SACMAT '08. New York, NY, USA: ACM, 2008, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/1377836.1377862

[18] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the 13th USENIX Security Symposium*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16.

[19] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture," in *Proceedings of the 11th ACM symposium on Access control models and technologies*, ser. SACMAT '06. New York, NY, USA: ACM, 2006, pp. 19–28. [Online]. Available: http://doi.acm.org/10.1145/1133058.1133063

[20] M. Fox, J. Giordano, L. Stotler, and A. Thomas, "SELinux and grsecurity: A Case Study Comparing Linux Security Kernel Enhancements," 2003.

[21] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting, "Authenticated system calls," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, June 2005, pp. 358–367.

[22] M. Rajagopalan, M. A. Hiltunen, T. Jim, and R. D. Schlichting, "System call monitoring using authenticated system calls," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, pp. 216–229, July 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1159168.1159375

[23] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," in *Proceedings of the 2008 Annual Computer Security Applications Conference*, ser. ACSAC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 418–430. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2008.54

[24] B. Li, J. Li, T. Wo, C. Hu, and L. Zhong, "A VMM-based system call interposition framework for program monitoring," in *Proceedings of the 16th IEEE International Conference on Parallel and Distributed Systems*, ser. ICPADS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 706–711. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2010.53

[25] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection and Monitoring Through VMM-based "out-of-the-box" Semantic View Reconstruction," *ACM Transactions on Information Systems Security*, vol. 13, pp. 12:1–12:28, March 2010. [Online]. Available: http://doi.acm.org/10.1145/1698750.1698752

[26] B. Ford and R. Cox, "Vx32: lightweight user-level sandboxing on the x86," in *Proceedings of the 2008 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 293–306. [Online]. Available: http://portal.acm.org/citation.cfm?id=1404014.1404039

[27] T. Kim and N. Zeldovich, "Making Linux protection mechanisms egalitarian with UserFS," in *Proceedings of the 19th USENIX conference on Security*. Berkeley, CA, USA: USENIX Association, 2010, pp. 13–27. [Online]. Available: http://portal.acm.org/citation.cfm?id=1929820.1929823

[28] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "BLADE: an attack-agnostic approach for preventing drive-by malware infections," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 440–450. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866356