Process Authentication for High System Assurance

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

Abstract—This paper points out 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; PIDs or process names are identifiers for processes in an OS environment. However, the information such as process names or executable paths that is conventionally used by OS to identify a process is not reliable. As a result, malware may impersonate other processes, thus violating system assurance. We propose a lightweight secure application authentication framework in which user-level applications are required to present proofs at run time to be authenticated to the kernel. 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 processes before completing the requested system calls. We implement and evaluate a prototype of our monitoring architecture in Linux. The results from our extensive performance evaluation shows that our prototype incurs reasonably low overhead, indicating the feasibility of our approach for cryptographically authenticating applications and their processes in the operating system.

Index Terms—Operating system security, process authentication, secret application credential, system call monitoring

1 Introduction

Typical operating system 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, usually can freely execute privileges provided to the user account running the process.

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 [2], grsecurity [3], and AppArmor [4] enable the user (or the system administrator) to express detailed and powerful policies. These solutions are often implemented using the Linux Security Modules [5] to monitor access to selected system resources, and apply the specified policies to the corresponding processes.

The above security solutions belong to the category of authorization. However, authorization mechanisms alone are not sufficient for achieving system assurance. Our thesis in this paper is to argue and demonstrate that the 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 reported in the literature.

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. Typically, 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. 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. Without secure process authentication, malware may impersonate legitimate applications and abuse system resources, thus violating system assurance.

- *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 [1].*
- *This work has been supported in part by NSF grant CAREER CNS-0953638 and ONR grant N00014-13-1-0016.*

This paper 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 between process authentication and 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 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 [2] 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. Our 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 legacy 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.

Because of the complexity of the operating system's tasks in managing a large number of diverse applications, ensuring the authenticity of the basic operating data is becoming increasingly important. With modern attack models, system information whose security is usually taken for granted needs to be re-examined and re-evaluated. The cryptographic provenance verification work in [6] points out the need for the kernel to ensure the authenticity of origins of data flows that are consumed by the system. For example, the data may be user inputs or traffic flows. Recent assured digital signing work in [7] describes methods for the integrity protection and authenticity verification of a signing agent on a host for creating digital signatures. It points out the differences between a human signer and a program signer and the system challenges associated with realizing a trustworthy program signer. Their solution extends the attestation service of the hardware trust platform module (TPM). Our work demonstrates another case of hardening the system by re-examining the fundamental process identification mechanism.

The contribution of our work is not only the specific A2 solution presented, but also the systematic discussion on the requirements and challenges of process authentication in OS environments. Even though user authentication of various flavors is very well understood in the network security literature, process authentication requires careful system security design and engineering as a process is less autonomous compared to human.

2 Model and Overview

We give the models and definitions used in our work. We discuss the design choices and general requirements for the authentication of applications and their processes in the operating system.

2.1 Motivations

We motivate our work through discussing and distinguishing four pairs of concepts related to authentication below.

Process identification vs. process authentication A process identifier may be the process ID, process name, etc. In the context of our A2 work, we define process identification as a naming convention to describe a process. Process authentication, on the other hand, is for a process to prove its identity to the operating system. It needs to prevent identity spoofing. There is no process authentication mechanism in the systems security literature, even though almost all access control solutions for OS make access decisions based on who the processes are. In these systems, installation paths may be used to distinguish among processes. However, such a simple mechanism is weak. We summarize their differences from our A2 solution in Table 1, and further explain them in Section 2.2.

Process authentication vs. user authentication Unlike the conventional user authentication (e.g., password authentication) in a client-server architecture, process authentication imposes new and unique system and security challenges.

- *Storage of the secret* The user's secret (e.g., password or passphrase) can be memorized by the user. A process' secret is stored on the host and needs to be kept confidential with the help of the kernel to prevent it from being accessed by other processes.
- *Compatibility* An authentication mechanism needs to be compatible with legacy applications without requiring customization.

Application authentication vs. process authentication The authentication of applications is realized through the authentication of processes of that application. By process authentication, we refer to that the process of an application needs to prove the application's identity. (We choose to perform this authentication at the start of the process in our prototype.) We use the terms application authentication and process authentication interchangeably in this paper.

One-time authentication vs. runtime authentication status One-time authentication refers to the authentication of a process, which can be done at its creation. The authentication status of a process needs to be recorded and maintained by the system. At runtime, when the process makes requests to access system resources (e.g., system call requests), the authentication status can be queried and used for deciding whether or not to grant the request. Our A2 system supports both mechanisms.

2.2 Types of Credentials

An important part of our contribution is the systematization of the security requirements and components of a general process authentication framework. The discussion on these issues is fragmented in the current literature. With our general framework, capabilities of various credential types can be compared. There are various approaches to instantiate an application credential, as explained next.

- Labels are given to binaries in SELinux as their extended attributes [1]. These labels may serve as application credentials. Yet, additional extension to SELinux needs to be made to ensure the uniqueness and integrity of labels (e.g., to prevent unauthorized relabeling or label inheritance).
- Keyed hash such as HMAC is another way to instantiate application credential. Hash-based approaches have been used for code integrity in the context of trusted computing [8] or buffer overflow prevention [9]. For example, in runtime execution monitor (REM) [9], the HMAC hash value is computed for a small block of instructions. The collection of HMAC values are appended to the program. The hash values need to be recomputed at the time of verification. Execution of malicious code without the proper HMAC is flagged.
- In Table 1, application signing refers to the following approach. The kernel is given a public and private key pair. It uses the private key to sign the hash value of the code of a valid application, which generates a credential for the app containing the digital signature from the kernel. At verification, the hash value of the code is recomputed first, and then the signature is verified against the public key by the kernel. The advantage of this approach compared to A2 is that the credentials are not secret and are easy to manage. On the other hand, the private key of the kernel needs to be kept secret. Similar to the keyed hash above, the hash values need to be recomputed at the time of verification.

A complete application authentication framework needs three components: *credential generation*, *process authentication*, and *runtime monitoring*. Our contribution on the runtime monitoring (in Section 3.4), including efficiently managing the status of authenticated, is useful independent of the specific credential type.

2.3 Security Models

Security goals and assumptions Our security goal is to ensure the system assurance, which is to verify that a system enforces a desired set of security goals [10], more specifically, to ensure that the operating system correctly authenticates processes of applications at the runtime and malware cannot impersonate the identities of legitimate processes. Malware may abuse system calls for conducting malicious network activities, e.g., botnet command and control [11], data exfiltration [12], fetching executables [13], and spyware eavesdropping [14].

Our basic trusted components are the kernel code, kernel data structure (e.g., PIDs), and kernel's memory region. The kernel's code is trustworthy and does not contain any malicious code. The confidentiality and integrity of the kernel's memory are preserved. (Such a trust can be partly established using existing techniques such as the Trusted Platform Module (TPM) [15], [16] at boot time, assuming the exclusion of hardware attacks.)

Attack model Stealthy malicious code on the system may attempt to run itself as a stand-alone user-level process. Malware may be downloaded to the victim computer through a crafted malicious web page (e.g., drive-by download). Malware stores and attempts to execute at the user space. Malware may attempt to impersonate other (legitimate) applications, e.g., by spoofing the names of other processes. Thus, process names alone are not reliable for distinguishing processes. Malicious code running within the boundary of a legitimate process (through code injection, or a malicious browser script or extension) is out of the scope of our attack model (See also the discussion in Section 4).

2.4 Application Credential and Its Requirements

We define *secret application credential* next and explain the requirements for realizing a specific credential scheme in the operating system environment.

Definition 1. *A secret application credential (SAC) is a unique secret information issued to a trustworthy application by the operating system. SAC is used for processes of*

[1] Labeling is based on path names (which could be forged).

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.
Hashes on files or code blocks	Have to recompute hashes at authentication	Storing hashes as registered app identities	Hash values may be used as A2 credentials (See also 4.3).
Code hash signed by kernel	Kernel's private key should be kept secret. Need to recompute hashes for verifying the signature	Storing hashes as registered app identities	Need kernel-key protection, similar to A2's credential protection. Hash values may be used as A2 credentials (See also 4.3).
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
A comparison between the security in conventional process identification mechanisms and our application authentication solution. See also Section 2.2.

the application to prove their identities to the OS kernel during the authentication procedure.

There are various approaches to instantiate secret application credentials, but they need to satisfy the following requirements. Some requirements are common in other credential systems, whereas some are new and unique to the process authentication problem.

- *Uniqueness* For one executable, there is no more than one secret application credential (SAC). If the executable is reinstalled on the file system, its SAC is updated.
- *Secrecy* SACs shall not be available to unauthorized userland processes.
- *Unforgeability* Valid SAC cannot be forged.
- *Anti-replay* Replaying a legitimate SAC to the kernel will be caught.
- *Binding* A SAC needs to uniquely binds to all the processes of a single executable.
- *Status checking* The authentication status of a process (i.e., whether a running process has been successfully authenticated or not) needs to be recorded. This status information can be queried before granting the runtime access rights of a process.

2.5 Operations For Process Authentication

The authentication operation requires userland processes to demonstrate the possession and knowledge of kernel-issued application credentials. Processes without valid credentials are restricted from accessing system resources (e.g., making system calls) and considered potentially malicious. This mechanism provides a secure sandbox that isolates malware from system resources. We describe the general operations needed for process authentication solutions, including Credential generation, Process authentication, and Runtime monitoring.

1) Credential generation This is a one-time operation run by the kernel to issue the secret application credential to a (trusted) application. This operation may be performed at the time of application installation.
Determining whether an application should be given a credential is a classification procedure, which is independent of our focus on process authentication. A classifier analyzing the trustworthiness of the executable code can be deployed for this purpose to complement A2, e.g., using the static programming analysis tools described in [17], [18].
2) Process authentication This is a protocol run by the kernel and a process for the process to authenticate itself to the kernel. The authentication is through the process proving the possession of the required SAC value.
3) Runtime monitoring The kernel monitors the execution of processes so that processes that have not been properly authenticated are caught. A system administrator may also choose to enforce fine-grained access control policies at the process level (e.g., specifying what system calls

can be performed by applications), which can be integrated with this operation.

A process is not allowed to inherit its authentication status from its parent process in our model. Next, we describe details of our design for a system that supports the authentication of applications.

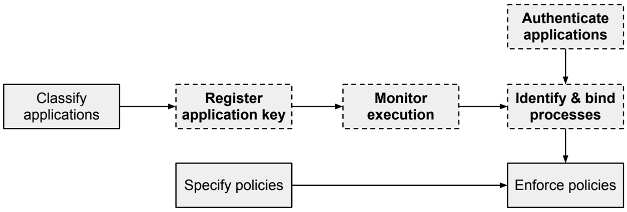


Fig. 2.1. A diagram showing how A2 components/operations (in dashed line blocks) work with each other and how they may be integrated with complementary security elements.

3 Authenticated Application (A2) System

Our Authenticated Application (A2) design enables the authentication of applications. It consists of three main components: *Credential Registrar*, *Authenticator* and *Service Access Monitor (SAM)*. (We implement the Authenticator and SAM as Linux kernel modules without modifying the kernel). We describe the functions of our components in the following, and describe in details how each of three operations Credential generation, Process authentication, and Runtime monitoring are realized.

Credential Registrar is for generating a credential for the application and registering the application with the kernel.

Authenticator is for authenticating a process when it first starts.

Service Access Monitor (SAM) is for verifying the authentication status of a process at runtime, i.e., whether the process has been successfully authenticated by the Authenticator.

There are two important lists in A2, the *credential list* and the *status list*. The credential list is the registrar's copy of the all the current valid credentials generated for registered applications. The status list is the Authenticator's record of the currently running processes that have been successfully authenticated.

3.1 Credential Generation and Storage

The kernel, more specifically the credential registrar, generates the secret credentials for legitimate applications. In A2, the registration operation for trustworthy applications can be done any time between the time of installation and the time of first execution. The registrar and the application each maintain a copy of the credential. The credential is no longer valid if the application is removed, reinstalled, or modified; and a new credential needs to be issued. There are many algorithms to instantiate the credential value. A simple method is to use a secret value of sufficient length as the SAC value that is generated by a pseudorandom number generator controlled by the kernel. The random values should be hard to guess.

A key problem in credential storage is how to protect the secrecy of application credentials that is stored by the application. (Kernel side of credential storage is assumed to be secure in our model.) To address that problem, we introduce a protection mechanism referred to as the *code capsule*. The application's copy of the secret credential is stored along with the application's code capsule (e.g., appending to the end of the executable). We define the code capsule as follows:

Definition 2. *A code capsule is a piece of executable code along with a secret application credential that is unique and verifiable by the kernel. A code capsule is not read or write accessible by any user process except by necessary kernel helper processes.*

Code capsules serve two major purposes. One purpose is to protect the secret application credential from being revealed to unauthorized processes through the file system. The other purpose is to bind a credential with the executable 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, namely the credential registrar. When the application is executed, the credential is not loaded into the memory.

The Registrar runs a registration function as defined below.

Definition 3. *The registration function* $\rho : E, s \rightarrow C_s$ *where* E *is a string containing the executable code and* s *is the secret application credential generated by the credential Registrar. The function* ρ *produces the string* $C_s = E||s$*, which is a code capsule protected by the kernel.* $||$ *represents string concatenation.*

We describe the steps for a credential Registrar to generate an application credential next. Applications with the credentials issued by the Registrar are referred to by us as the registered applications. For malicious applications that bypass this registration phase, they cannot succeed in the authentication next due to the lack of valid registered credentials. Denote the current credential list maintained by the Registrar by `L`. The list consists of (`name`, `credential`) pairs of registered applications. The list needs to be kept confidential with restricted read and write access.

1) The Registrar runs an external checking mechanism (e.g., a classification method [17]) to verify that the application with name `app.name` is trustworthy.
2) If the external verification fails (indicating that the application may be malicious), reports it

and halts. Otherwise, the Registrar generates a random value of required length n as the new credential s, and generates a code capsule C_s using the function $\rho(E, s)$. The Registrar writes C_s to the file system.

3) The Registrar appends the tuple (`app.name`, s) to the credential list `L`.
4) When the application is uninstalled, the Registrar is notified and deletes the entry (`p.name`, s) from the credential list `L`.

The credential generation operation is fully compatible with legacy applications and does not require any customization. Large applications may consist of several executable files. We register each executable file that may create at least one independent process with a unique credential. 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. The registrar may also need to ensure that the application has not been previously issued a credential, e.g., by checking whether a credential already exists at the end of its executable.

We point it out that the trusted registrar itself can be given a credential (e.g., manually installed by the system administrator). Then it can engage in the authentication procedure with the kernel as a regular process once it starts every time. The detailed process authentication protocol via challenge and response is described next.

Our work on protecting secret credentials on the disk may bear some resemblance to existing code integrity work (e.g., [8], [9]) or rootkit detection solutions (e.g., [19]). For example, the solution in [19] controls the access to specific regions of the disk under the assumption of an already compromised system. As we aim for the protection of a short secret application credential (SAC) as opposed to general kernel and user level code or data, the solution that we adopt is much more specific and thus lightweight. Because the goals of these pieces of work significantly differ, none of the existing solutions provides a satisfying solution for the application authentication problem as A2 does.

3.2 Process Authentication

The process authentication protocol is to authenticate individual processes based on the credentials of the corresponding applications. We first discuss several design choices for realizing process authentication, and justify our approach next. One simple design choice is that the kernel directly accesses the application's credential and verifies its identity provided that the credential 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 credential can be either defined in memory or the file system. Defining the credential in the memory imposes additional risk to stealing the credential as well as causing complexity of maintaining the credential location. An alternative design is to isolate all credentials in a restricted storage. The kernel retrieves the credential of corresponding process at the authentication time. However, this design is clearly inadequate because it does not bind a running process to the corresponding credential file at the runtime.

In order for a process to prove its identity to the kernel using the application's secret credential, our approach is for the process and the kernel to engage in a challenge-and-response protocol. The challenge-and-response concept is common in network security. We tailor it for the operating system environment. The main steps are summarized below and the details are presented in Section 3.3. *i)* The kernel sends a random nonce to the application process. *ii)* The process produces the hash-based message authentication code (HMAC) with the nonce and the secret credential and returns the hash value to the kernel. *iii)* The kernel recomputes the HMAC and compares it to the value submitted by the process. We describe the three technical challenges and our approaches for addressing them next.

- One technical challenge is the implementation of an efficient and reliable communication channel between the process and the kernel. Our authentication protocol is executed on a socket file between the process and the kernel. This method is realized using a memory-based socket or shared memory, e.g., `/proc` file system [20]. The advantage of using the shared memory 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.
  Throughout A2 design, this communication method via shared memory with restrictions is used for all the communications between the userspace and kernel space processes. The userspace processes may be applications or A2 components.
- Another technical challenge is that because the authentication protocol requires additional operations by processes, one needs to avoid having to modify and customize existing applications. We design and implement a piece of middleware that assists processes with the authentication operations. As a result, A2 is completely compatible with existing applications for process authentication. More details are given in Section 4.1.
- The third technical issue is how to minimize the authentication overhead while ensuring the runtime system assurance (e.g., at the system call level of granularity). Requiring the process to authenticate itself at every request of system call

incurs excessive runtime overhead. We choose to perform the authentication at the time of process creation. We have a lightweight mechanism to maintain the *authenticated status* of a process during all subsequent requests, which is conceptually similar to session IDs in the web.

3.3 Authentication Protocol

The authentication protocol is run between the Authenticator and a process at the time of process creation. The description below requires the application to be customized to follow our protocol. We eliminate this requirement in Section 4.1 for compatibility. The goal of A2 authentication protocol is two-fold: *i)* to securely authenticate running processes and *ii)* to record and maintain the authentication status of processes. The authentication status information is stored as a kernel data structure by the Authenticator, which is referred to as the *status list*. It is shared with the Service Access Monitor (SAM) at runtime for SAM to determine the legitimacy of processes submitting system requests in Section 3.4.

Let `A` represent the Authenticator module. We denote the `A`'s credential list by `L`. It is a list of (`app-name`, `app-cred`) pairs, where `app-name` is an application name, and `app-cred` is its corresponding application credential generated by `A`. The Authenticator (and no one else) can submit queries to the Registrar in the form of `query(app-name)`, which takes as the input an application name and returns its corresponding credential on `L`. The Authenticator `A` maintains a status list `T` consisting of process IDs of successfully authenticated running processes. The list `T` is made readable by the Service Access module (SAM). Let `p` be a user process; `p.pid` is `p`'s process identification; and `p.name` is `p`'s application name. We denote `p`'s copy of its secret credential (stored in the code capsule) by `p.cred`. Let `auth-request(p.app)` is the function used by `p` to send an authentication request to `A`. Let HMAC be a secure hash-based message authentication code function.

1) `p`: Sends `auth-request(p.name)` to `A`, i.e., the application claiming to be `p.name` requests to be authenticated.
2) `A`: Does the following.
   a) `p.cred'` ← `query(p.name)`, that is, queries the Registrar with `p.name` to retrieve its credential `p.cred'` on the credential list `L`. If the returned `p.cred' is null` i.e., the application does not have a registered credential, reports `p` as suspicious.
   b) Generates a random nonce and sends it to `p`. `A` also sets a timer t for the string to expire if there is no future response from `p`. The time frame to expire t is short (e.g., in milliseconds).
3) `p`: Computes h ← HMAC(`nonce`, `p.pid` `p.cred`), where `p.cred` is `p`'s secret credential obtained from its code capsule. h is sent to the Authenticator `A`.
4) `A`: If the delay associated with the received HMAC exceeds the required threshold t, the authentication request is discarded and the authentication of `p` fails.
   `A` computes h' ← HMAC(`nonce`, `p.pid` `p.cred'`). If $h == h'$, then the authentication is successful. Otherwise, it fails.
   `A` checks to see whether `p.pid` ∈ `T`. If yes, reports `p` as suspicious. Otherwise, `A` appends `p.pid` to the list `T`.
5) When `p` terminates, `A` is notified, deletes `p.pid` from the status list `T`.

The ability for an application to succeed in the authentication protocol depends on its knowledge of the required secret application credential. E.g., if a process claims to be the Mozilla Firefox browser (i.e., with `p.name` being Mozilla Firefox), then it needs to prove its knowledge of the registered credential value of that application. The security of our protocol is thoroughly analyzed in Section 4. Our security guarantee is hinged on the confidentiality of the application credentials, both the copy on the credential list and the application's copy in the code capsule. The PID that is used as an identifier for querying the status list belongs to kernel data, which is assumed to be trustworthy and unforgeable in our security model (in Section 2).

3.4 Runtime Verification of Authentication Status

At runtime, whenever a process makes a request for accessing system resources through system calls, the Secure Access Monitor (SAM) intercepts the request and verifies the authentication status of the process, i.e., whether the process has successfully passed the authentication. This verification is accomplished by SAM through looking up the process's PID on the status list of the Authenticator, and verifies the PID's existence on the list. Our experiments show that this runtime verification of authentication status of processes is lightweight. PID is used an identifier for looking up the status list. Because of our assumption on kernel's code and data integrity, PID values used for this runtime verification are trustworthy, specifically unforgeable.

Our authentication system can be conveniently integrated with existing policy based access control systems for strong system assurance. SAM can also be integrated with a policy specification language to benefit from existing work in policy specification such as [21] that uses an abstracted logical language to specify SELinux policies and Polymer [22], a runtime policy specification framework.

4 Discussion

In this section, we first present our solution for solving the compatibility issue in the authentication protocol. Then, we analyze in details the security guarantees that A2 provides, as well as justify the integrity of A2 components themselves.

4.1 Compatibility Mode for Legacy Applications

Our process authentication protocol described as in Section 3.3 requires the modification of legacy applications to support the interaction with the Authenticator, raising a compatibility issue. We deign a middleware to perform the authentication on behalf of the application. The credential generation operation is unchanged. We authenticate legacy applications using a helper program referred to as the *Verifier*. The Verifier has the read access to the credential list `L` maintained by the registrar, but not the write access.

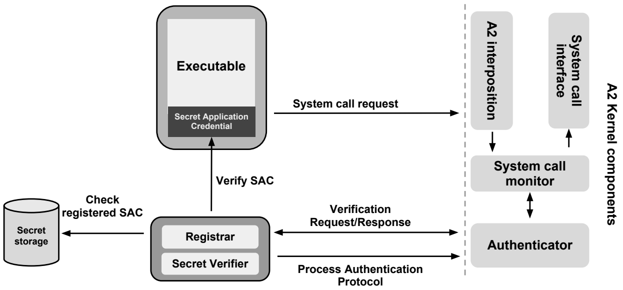


Fig. 4.1. Workflow of A2 in the compatibility mode with the verifier.

1) To authenticate a newly started process `p` with process name `p.name`, process ID `p.pid`, and the path to the code capsule `p.path` (all obtained from the kernel), the Authenticator checks if the process has already been verified by looking its `p.pid` up in the status list `T`. If `p.pid` $\notin$ `T`, the Authenticator sends to the Verifier (`p.path`, `p.name`).
2) The Verifier reads `p.path` to retrieve the application's copy of the credential at the end of its code capsule. This credential is denoted by `p.cred`. It throws an error if the credential cannot be found.
3) The Verifier looks up the credential list `T` by `p.name` to retrieve the corresponding credential, which is denoted by `p.cred'`. It throws an error if `p.cred'` is null.
4) The Verifier checks if `p.cred' == p.cred` [2]. If yes, then the authentication succeeds. Otherwise, fails. The Verifier notifies the Authenticator with the authentication result.
5) The Authenticator updates the status list with `p.pid`.

The Verifier's main task is to access the code capsule of the application on behalf of the application. The security guarantee of the authentication protocol using the Verifier is equivalent to the one without it (discussed next). In our A2 prototype, the Verifier is implemented as a userspace application. It has a shared memory region with the Authenticator to exchange verification messages.

The Verifier is equipped with a manually installed credential, so that itself can be authenticated as a bootstrapping procedure. When the Verifier's process starts, the Authenticator authenticates its identity to prevent identity spoofing.

4.2 Security Guarantees

The security of A2 relies on the confidentiality of the application credentials. Thus, we analyze our security guarantees by discussing the confidentiality of the application credentials and the integrity of A2 components.

Unforgeability credentials

Forging existing secret credentials (that appear on the credential list) by attackers is computationally hard, as long as a strong pseudorandom number generator is used to generate the credential. Besides existential forgery, a malware process using a self-generated arbitrary value as its credential cannot successfully pass the authentication, because that self-generated credential is not registered with the kernel and does not appear on the credential list.

Confidentiality of code capsules and credential list

To protect the secret credential from being revealed to other applications, A2 restricts the read access to applications' binaries, namely code capsules (where the application's copy of credential is stored). Malware may attempt to steal a credential from application's or A2 components' memory at runtime, which is prevented by the standard process memory isolation mechanism of the system.

Similarly, A2 restricts the access to the credential list (owned by registrar) by other processes, thus ensuring its confidentiality. More specifically, only the registrar and the verifier have the direct access, and the Authenticator can (indirectly) query the list.

Resistance to replay attacks

Malware may attempt to intercept the challenge-and-response communication between the Authenticator and a process during the authentication protocol. Because the `proc` file is only readable by A2 components, messages exchanged cannot be intercepted preventing replay. (This read restriction is enforced in kernel by A2.)

Integrity of A2 components and data

[2] Instead of the direct comparison of the two copies of credentials, an equivalent-yet-less-straightforward approach is for the Verifier to engage in the authentication protocol of Section 3.3 with the Authenticator (on behalf of the application process).

A2 components span both the kernel space and the user space. The kernel-level components include the Authenticator and Service Access Module. Because of our assumption on the kernel integrity, these two components are trustworthy. In contrast, there is no assumption on the integrity of the user-level Registrar and Verifier, which may be targets of malware tampering and spoofing.

For anti-spoofing, A2 requires these two programs (namely the Registrar and Verifier) to authenticate to the Authenticator through our A2's authentication protocol as they start. The authentication procedure slightly differs from the one described in Section 3.3 in that *i)* their credentials are manually generated, and *ii)* the kernel's copies of their credentials are hardcoded in the Authenticator, as opposed to be stored on the credential list. For anti-tampering, A2 forbids the write access to the code capsules of the Registrar and Verifier by other userland processes.

A2 data includes the credential list, status list, as well as the intermediate communication messages in the authentication protocols via (`proc` file based) shared memory. We have discussed the confidentiality of the credential list, which is a user-space file. In contrast, the status list is a kernel data structure in the memory. Thus, its integrity and confidentiality are guaranteed as a result of our kernel integrity assumption.

The shared memory approach is used for all the communication between the kernel modules and user-level processes, including: *i)* the authentication protocol messages between the Authenticator and the requesting process, *ii)* credential queries between the Authenticator and the Registrar, and *iii)* authentication status update between the Authenticator and the Verifier. A2 secures the confidentiality and integrity of the shared memory based communication channels by preventing the read and write access to the shared memory by other non-related user-level processes.

In summary, the A2 solution guarantees that the operating system correctly authenticates processes of applications at the runtime and malware cannot impersonate the identities of legitimate processes, thus satisfying our security goal specified in Section 2.

4.3 Extensions and Limitations

Updates. Handling updates and new installation in A2 is convenient and requires limited user effort. Our policy considers each installation of an application as a new entity that needs a new credential. An upgrade in Linux desktop systems may be upgrading a resource file, patching the application with additional libraries or modification of existing libraries, and providing a modified binary. Updates may trigger A2 to generate separate credentials for the resources, or regenerate a new credential for the main application. The user needs to authorize credential updates. In recent Linux systems, software updates already require user permissions. These permission can also serve as the user authorization for credential updates.

Dynamic linked libraries (DLL). Validating dynamically loaded libraries is not performed during the authentication of the main application. When invoked, the library itself can be separately authenticated in A2, which requires the library to acquire a credential of its own. The library's credential is examined before it is loaded.

Linux shared libraries (such as libc.so) are loaded by the dynamic linker (ld.so). For authenticating libraries, there are two design choices. The first is that prior to loading a specific library, the dynamic linker itself interacts with the verifier to request for authentication. The second design choice is to monitor the dynamic loader and enforce authentication on all files opened and loaded by the dynamic linker. The second design choice maintains the compatibility with the current dynamic loader and can integrate as an additional functionality in the process monitor. For the two approaches described above, one needs to perform the library authentication at the time of loading the library into the memory. Thus, there is no additional overhead per system call.

Limitations. A2 is capable of identifying interpreted programs running as stand-alone processes. For instance, a Java executable runs as a separate process. The program can be given a unique credential at registration. Each program can authenticate itself independently using our framework. Other interpreted languages such as JavaScript, Adobe Action Script, and Word document macros are out of our security model, because the program runs in a container process as opposed to a separate process ID. For a similar reason, the detection of malicious code injected into (vulnerable) authenticated processes – as opposed to running as a stand-alone process – is out of the scope of A2's attack model.

The security of A2 partly depends on the accuracy of the classification analysis. Classifying the trustworthiness of a program is challenging, and its inaccuracy may allow malware to obtain a legitimate secret application credential. In practice, multiple complementary static and dynamic analysis and monitoring tools have to be used to improve the accuracy of classification.

Extensions. Although attacks that cause control flow hijacking are out of our model, they may be detected by extending the current methods.

- Direct detection. There exist control flow integrity tools and methods (e.g., StackGuard [23], address space layout randomization) to prevent the control flow hijacking exploits. A recent good survey can be found [24]. These solutions can be used together with A2 on a system.
- Detection of effects. Control flow hijacking is used by attackers to reach their goals.
  1) In the context of A2, the goal may be to ac-

cess or modify credential information without proper authorization. Boot-time integrity checking (e.g., TPM-based solutions [25]) can be integrated with A2 to detect unexpected modification in the credential data on disks.

2) The goal may also be to launch new processes to complete attack tasks, e.g., eavesdropping, command and control, or sending attack traffic. Together with the boot-time integrity checking, A2 can detect the creation of such unauthorized new processes, as they would not have the proper credentials.

5 IMPLEMENTATION

We have developed a prototype of the A2 framework in C in Linux (3.2.0-36), including the implementation of the credential registrar, the Authenticator, the Service Access Module, and the verifier. (The verifier is introduced in Section 4.1 to improve the compatibility of A2.)

- The Authenticator and SAM are kernel device drivers (modules). These two modules are loaded at boot time.
- The credential registrar and the verifier are implemented as kernel helper programs that run in the user mode for efficiency and compatibility considerations. We avoid the unnecessary context switches to register an application, and can use standard user-level libraries.

  As a bootstraping procedure, we require the registrar and the verifier to authenticate themselves to the kernel (namely the Authenticator) with their respective secret credentials at the time of the process creation.

Credential List

Each secret credential is a 128-bit random value generated by the AES key generation algorithm during the CREDENTIAL GENERATION operation by the registrar. We describe how the credential list can be accessed by each of the A2 components.

- The registrar can read and write to the credential list, which is a file. The verifier can read the file as well. No other processes can access that file. This restriction is realized as monitoring the `open` system call requests to the file. This system call monitoring is performed by SAM.
- Because Linux kernel components do not access file system directly, in our prototype the Authenticator and SAM do not have the direct read access to the file storing the credential list. They need to communicate with the registrar to retrieve a credential via the `proc` file mechanism.

As a bootstraping procedure, the registrar authenticates itself to the kernel Authenticator with its own credential. The registrar's credential is appended to its executable file, as specified in A2. In our prototype, this credential is hard-coded into A2 kernel modules.

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

Authenticator

To carry out the authentication protocol, the Authenticator communicates with the user space applications 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. We support 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-level process. The Authenticator module uses the Linux kernel Cryptographic API [26] to perform the HMAC operations using a number of supported hashing algorithms. Our implementation of the Authenticator can accept multiple requests from multiple processes using the same `/proc` protocol file. For each process, only one request is served at a time.

Status List and Secure Access Monitor

Secure Access Monitor (SAM) and the Authenticator communicate via a shared data structure in the memory that holds the status list, i.e., a list of PIDs of successfully authenticated running processes. This data structure is maintained by the Authenticator and visible to SAM (but no other processes). To verify a process' identity, SAM searches through the status list.

We implement the status list as a sequential dynamic array. In our experiments, under a normal use, the number of running processes was under 100. As it is shown in the evaluations, searching the status list did not have a significant overhead in a normal usage. However, in order to improve the overhead, one can implement the list as a red-black tree (a special type of balanced binary search tree [27]) that has a search complexity of O(log n) where n is the size of the status list in the 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 applications' functionality, limited by an average of 3 times more overhead (see Section 6).

To provide a more efficient alternative realization 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 require additional computational and management overheads in the operating system. In order to assess the efficiency of our framework, we answer the following questions in our experiments:

- How does A2 impact the overall system performance?
- What is the performance penalty caused by A2 on various I/O and system call functions?
- What is the process creation performance under A2?

Overview and Setup. We conducted extensive performance evaluations using various system benchmark suites. An overall measurement of A2's performance is calculated using UnixBench[3], and, Phoronix test suite[4]. To measure the performance of critical system functions, such as I/O and system calls, we use lmbench [28] and UnixBench system benchmark suites.

Our experiments reveal efficient performance of A2 in various system operations. The highest performance downgrade is in `open` system calls (with A2 being about two times slower than the generic stock Linux). Our overall performance measurements, process creation and general I/O operations show reasonably fast performance of A2.

Our experiments executed on directly on a physical Intel Core Duo 2 machine with two cores of 2.99 Mhz speed and 3 GB of ram. The kernel version on the testing machine was Linux 3.2.0-36. At the time of each test, there was no user interaction with the machine except for execution of the benchmark programs. All benchmark results show an average of several iterations.

We tested the system's performance twice for each experiment. First, we experimented with the stock generic Linux kernel (referred to as generic in Figures 6.1, 6.2, 6.3, 6.4 and Table 2) installed on the machine. Second, we tested the system with all A2 modules loaded. Also, we developed a daemon to simulate the authentication, by sending authentication requests to the kernel and performing the authentication protocol, in the intervals of 60 seconds during the course of each experiment.

Benchmark	A2	Generic	Decrease
PHP compile time (seconds)	75.43	74.75	0.91%
Apache throughput (requests/sec)	12324.62	12531.00	1.65%

TABLE 2
Comparison of PHP compile time and Apache throughput (using Phoronix test suite) in A2 vs the generic Linux.

Overall system performance. For an overall measurement of A2's performance, we used UnixBench and two benchmarks from Phoronix test suite, that is PHP compile, and Apache throughput. UnixBench performs various system level tests and calculates an overall performance index relative to a base score, referred to as the BYTE index. A higher value of the BYTE index indicates a relatively better performance.

Figure 6.1 shows the results form UnixBench tests. The overall performance penalty with a single processor is 0.977%. For two parallel executions, the performance downgrade is 1.534%. Our tests with PHP compile and Apache throughput show an efficient overall performance of A2. As described in Table 2, PHP compile time had a downgrade of 0.91% under A2, and Apache's requests per second downgraded 1.65% when using A2.

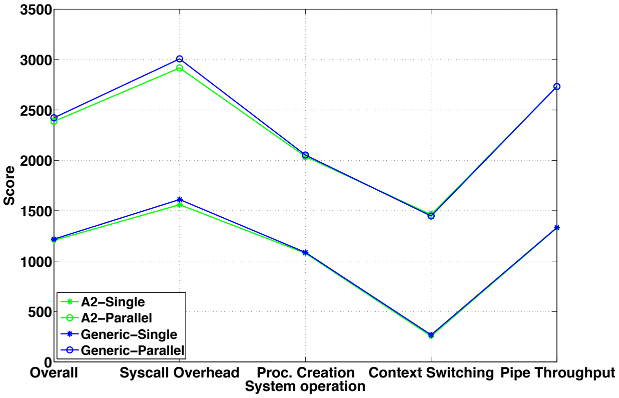


Fig. 6.1. UnixBench benchmark operations and results. The values are calculated according to a base score method. For each kernel (A2 and the generic Linux kernel) two sets of experiments are performed: single and two parallel copies, one for each core.

I/O and system call performance. UnixBench and lmbench perform extensive I/O and system call performance measurements. The benchmarks involve continuous calls to a system function and measuring the total processing time. For UnixBench results (in Figure 6.2), we show the benchmark scores, whereas

[3] http://code.google.com/p/byte-unixbench/
[4] http://www.phoronix-test-suite.com/

for lmbench results (in Figure 6.3) we show the actual processing time.

The file copy operations in UnixBench had an average decrease of 1.55% in performance with maximum decrease of 2.161%. lmbench measures calls to `open` followed by a `close`, as depicted in Figure 6.3. These calls had the most downgrade of about 235% in A2. This major downgrade is mainly due to various checks that we perform at the `open` system call to make sure proper access rights on protected regions that contain application credentials (e.g., checking access to code capsules). Other I/O operations such as select on file descriptors, select on TCP file descriptors, and sockets demonstrated statistical ties between A2 and the generic kernel.

In our prototype, the system call checking has two levels: level-I checking to see if the call is monitored and level-II checking to see if the authentication is needed. Every system call invocation is intercepted for the level-I checking. For the purpose of system assurance, only select system calls performing security critical functions go through the level-II checking. The support for additional critical system calls such as `sys_socketcall` may be included in the future.

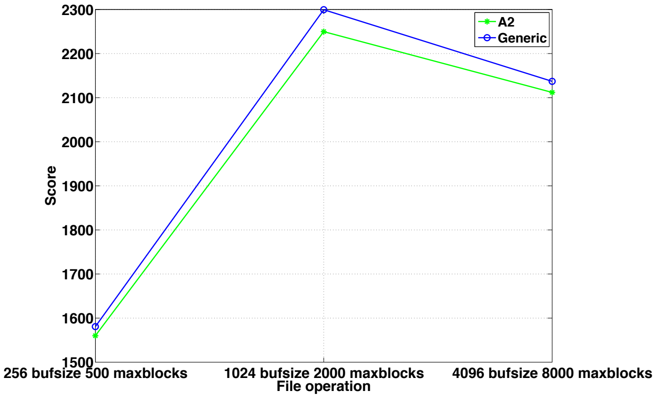


Fig. 6.2. UnixBench benchmark file copy with various sizes. The file copy is slightly affected by our permissions checking in A2 with an average downgrade of 1.55%.

Process creation performance. We directly monitor calls to `fork` and `execve` for monitoring process creations and activities by all processes to enforce our mandatory authentication protocol. Our monitoring involves a check to see if the process requires authentication. Thus, expect modest performance penalties by A2. To measure process creation, we used lmbench (Figure 6.4) and UnixBench (Figure 6.1) process creation benchmarks.

The results from lmbench show an average performance downgrade of 2.1627% when using A2 modules and a maximum downgrade of 2.949% for the fork, execve and shell execution results. UnixBench measures process creation with fork with a performance downgrade of 0.686% in the single execution experiment and a performance downgrade of 0.701% in the parallel execution experiment. The corresponding experiment in lmbench (calls only to fork) has a similar performance downgrade of about 1%.

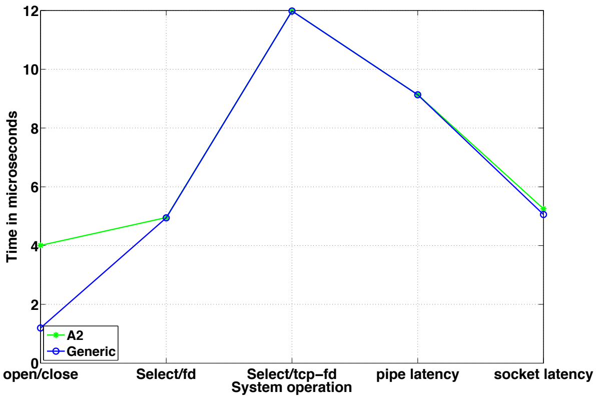


Fig. 6.3. lmbench benchmark operations and results. The values show time to execute the operation in microseconds. Most operations perform efficiently and do not suffer major performance downgrades. The performance downgrade in open/close is due to our permission checking before granting a file descriptor in the open system call.

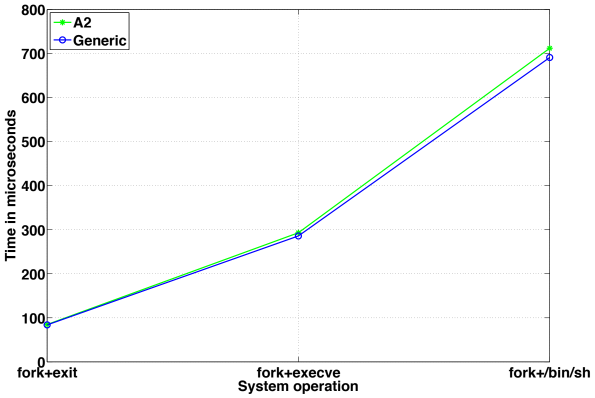


Fig. 6.4. lmbench process creation results. The values show time to execute the operation in microseconds. The results show process creation using fork without running external code, process creation with a call to execve, and process creation with running shell scripts.

7 Related Work

Existing work on protecting system's integrity is studied in the form of program integrity measurement techniques [29], [30], [31], information flow integrity [32], mandatory access control [2], [4], virtual machine monitors [33], [34], and application sandbox [35], [36], [37].

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 [31]. 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 [30] a similar approach is taken to apply IMA on mobile operating systems.

The work in [31] is enhanced by PRIMA [32] 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 [29]. 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.

In contrast with remote program attestation methods, A2 does not require a remote verification for program and system integrity. However, remote verification can be used in conjunction with A2, for example when verifying the integrity of a system running in an untrusted cloud.

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 [2]. 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, grsecurity [38] provides sophisticated memory protection mechanisms such as enforcing read-only memory pages.

Policy-based systems such as SELinux are found to be difficult to use by end users [39], and lack a general application authentication mechanism. In A2, we provide the first process authentication mechanism. It is independent of a particular user identity and does not rely on dynamic features such as a process ID, yet (in its core functionality) does not depend on complex policy specification systems.

In [40], [41], 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 [42] mostly focused on the use of virtual machine monitors (VMM) to monitor system calls [33], [34]. 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. Furthermore, a VMM may be used in A2 to ensure the integrity of the kernel itself.

Application sandbox is a mechanism to allow execution of untrusted code on protected hosts. Recent sandbox proposals include Vx32 [35], UserFS [36], and BLADE [37]. 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 and classification before registering the application as a legitimate application.

8 Conclusions and Future Work

Our work is the first to formally design application and process authentication in the operating environments. We have demonstrated its feasibility by presenting our architecture, implementation, and evaluation of a prototype Linux system supporting process authentication. We explained how process authentication can isolate malicious processes and thus prevent them from abusing and accessing system resources. The authentication model of A2 is highly portable and can be made compatible with legacy applications without any customization. Our evaluation results indicate that the overhead of performing process authentication at the system call level is acceptable.

A secure system needs multiple layers and components of protections. Achieving strong software assurance (i.e., ensuring the trustworthiness of code) is difficult if not impossible. Thus, we believe that our work on preventing untrusted code from running is equally valuable. Future work of ours will be focused on porting our A2 design to Android operating system for mobile devices in order to support the authentication of apps. Such a solution will significantly improve the system assurance of Android devices that may be targets of malicious and stealthy apps. We also plan to make A2 compatible with SELinux by extending and generalizing SELinux modules.

References

[1] H. M. J. Almohri, D. Yao, and D. Kafura, "Identifying native applications with high assurance," in *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY)*, February 2012.

[2] 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.

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

[4] 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.

[5] 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.

[6] K. Xu, H. Xiong, D. Stefan, C. Wu, and D. Yao, "Data-provenance verification for secure hosts," *IEEE Transaction on Dependable and Secure Computing (TDSC)*, vol. 9(2), pp. 173 – 183, March/April 2012.

[7] W. Dai, T. P. Parker, H. Jin, and S. Xu, "Enhancing data trustworthiness via assured digital signing," *IEEE Transactions on Dependable and Secure Computing*, vol. 9 (6), pp. 838 – 851, 2012.

[8] G. Xu, C. Borcea, and L. Iftode, "Satem: Trusted service code execution across transactions," in *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 321–336.

[9] A. M. Fiskiran and R. B. Lee, "Runtime execution monitoring (REM) to detect and prevent malicious code execution," in *ICCD*. IEEE Computer Society, 2004, pp. 452–457.

[10] T. Jaeger and R. Sandhu, *Operating System Security*. Morgan & Claypool, 2008.

[11] K. Xu, P. Butler, S. Saha, and D. Yao, "DNS for massive-scale command and control," *IEEE Transactions of Dependable and Secure Computing (TDSC)*, vol. 10, no. 3, pp. 143–153, May/June 2013.

[12] X. Shu and D. Yao, "Data-leak detection as a service," in *Proceedings of the 8th International Conference on Security and Privacy in Communication Networks (SECURECOMM)*, September 2012.

[13] K. Xu, D. Yao, Q. Ma, and A. Crowell, "Detecting infection onset with behavior-based policies," in *Proceedings of the Fifth International Conference on Network and System Security (NSS)*, September 2011.

[14] H. Zhang, W. Banick, D. Yao, and N. Ramakrishnan, "User intention-based traffic dependence analysis for anomaly detection," in *Proceedings of Workshop on Semantics and Security (WSCS)*, May 2012.

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

[16] 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.

[17] K. O. Elish, D. Yao, and B. G. Ryder, "User-centric dependence analysis for identifying malicious mobile apps," in *Proceedings of the Workshop on Mobile Security Technologies (MoST)*, May 2012, in conjunction with the IEEE Symposium on Security and Privacy.

[18] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2011.

[19] K. R. Butler, S. McLaughlin, and P. D. McDaniel, "Rootkit-resistant disks," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 403–416. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455821

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

[21] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel, "A logical specification and analysis for SELinux MLS policy," *ACM Transactions on Information and Systems Security*, vol. 13, no. 3, pp. 26:1–26:31, Jul. 2010.

[22] L. Bauer, J. Ligatti, and D. Walker, "Composing expressive runtime security policies," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 3, pp. 9:1–9:43, Jun. 2009.

[23] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *In Proceedings of the 7th USENIX Security Symposium*, 1998, pp. 63–78.

[24] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of IEEE Symposium on Security and Privacy*, May 2013.

[25] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *USENIX Security Symposium*. USENIX, 2004, pp. 223–238.

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

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

[28] 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

[29] 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.

[30] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger, "Measuring integrity on mobile phone systems," in *Proceedings of the 13^{th} 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

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

[32] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture," in *Proceedings of the 11^{th} 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

[33] 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 16^{th} 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

[34] 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

[35] 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

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

[37] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "BLADE: an attack-agnostic approach for preventing drive-by malware infections," in *Proceedings of the 17^{th} 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

[38] M. Fox, J. Giordano, L. Stotler, and A. Thomas, "SELinux and

grsecurity: A Case Study Comparing Linux Security Kernel Enhancements," 2003.

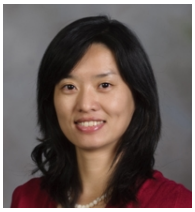
[39] Z. C. Schreuders, T. McGill, and C. Payne, "Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 2, pp. 19:1–19:28, Sep. 2011.

[40] 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.

[41] 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

[42] 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

Hussain M. J. Almohri received his B.S. degree from Kuwait University and M.S. degree from Kansas State University. He recently received his Ph.D. degree in computer science from Virginia Tech (2013). He starts as an assistant professor of computer science at Kuwait University. He is a member of ACM and IEEE computer society. His research focuses on systems security, mobile systems security, and quantitative security measurements.



Danfeng (Daphne) Yao is an assistant professor in Department of Computer Science at Virginia Tech. She received the PhD degree in computer science from Brown University. Her research interests are in system and network assurance and anomaly detection. She received the National Science Foundation CAREER Award in 2010 for her work on human-behavior-driven malware detection. She received best paper awards from ICNP '12, CollaborateCom '10, and ICICS '06. She received the Outstanding New Assistant Professor Award from VT. She is a member of the IEEE and the IEEE Computer Society.

Dennis Kafura is a professor in Department of Computer Science at Virginia Tech. He received his Ph.D. and M.S. degrees in Computer Science from Purdue University in 1974 and 1972, respectively. He joined the faculty at Virginia Tech in 1982, serving as Head of the department from 1998 to 2008. His research interests are broadly in systems and software engineering. He is the author of over 50 refereed journal and conference publications and the author of two books on object-oriented programming. He is a member of the ACM, the IEEE, and the IEEE Computer Society.