Identifying Native Applications with High Assurance *

Hussain M. J. Almohri
almohri@vt.edu

Danfeng (Daphne) Yao
danfeng@cs.vt.edu

Dennis Kafura
kafura@cs.vt.edu

Department of Computer Science
Virginia Tech
Blacksburg, VA, USA

ABSTRACT

Main stream operating system kernels lack a strong and reliable mechanism for identifying the running processes and binding them to the corresponding executable applications. In this paper, we address the identification problem by proposing a novel secure application identification model in which user-level applications are required to present identification proofs at run time to be authenticated to the kernel. In our model, applications are supplied with unique secret keys. The secret key of an application is registered with a trusted kernel at the installation time and is used to uniquely authenticate the application. We present a protocol for the secure authentication of applications. Additionally, we develop a system call monitoring architecture that uses our model to verify the identity of applications when making designated system calls. Our system call monitoring can be integrated with existing mandatory access control systems to enforce application-level access rights. 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 low overhead, indicating the feasibility of our approach for cryptographically identifying and authenticating applications in the operating system.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms

Security

Keywords

Operating system, malware, cryptography, application authentication, process identification

*This work has been supported in part by NSF grant CAREER CNS-0953638 and ARO grant STIR-450080.

CODASPY'12, February 7–9, 2012, San Antonio, Texas, USA.

1. INTRODUCTION

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, 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 [17], grsecurity [1], and AppArmor [8] enable the user (or the system administrator) to express detailed and powerful policies. These solutions are often implemented using the Linux Security Modules [26] to monitor access to selected system resources, and apply the specified policies to the corresponding processes.

In this paper, we point out that the existing MAC-based based approaches to application authorization alone are not sufficient for defeating modern malware. That is, the kernel must have secure mechanisms for authenticating and identifying processes, beyond the simple and easy-to-forge process ID (PID) or process name based identification. Thus, a critical problem in detecting malicious activities in the user-space is the ability to strongly identify processes at runtime and bind them to appropriate application identities. The identification of processes is a necessary step to prevent malicious processes to achieve their goals by benefiting from the access rights of legitimate processes.

Although providing useful security solutions, existing MAC systems do not explicitly address the problem of application identification. In [25], the authors present an extension to the Singularity operating system to define applications as first-class entities. The extension provides a language for the specification of application-level access rights. However, the proposed method combines the identity of the application (using application name) with the user's access rights and does not provide an explicit application identification model. Another approach is to identify malicious processes through the use of behavioral analysis. One may attempt to identify malicious processes through an anomaly-detection based approach by conducting analysis on process behaviors. However, this approach suffers from advanced

and newly discovered attacks that are capable of bypassing the detection scope [21].

Our goal is to securely identify processes at runtime and distinguish legitimate processes from undesired (and perhaps malicious) ones. Such a secure identification must provide high assurance in detecting and preventing the execution of malicious processes.

We present a novel identification model in which applications are identified and authenticated with high assurance. A privileged legitimate application is associated with a strong identity used to authenticate itself to the kernel. Using our identification model, we achieve the following:

- **Application identification.** Applications with registered identities, can authenticate to the trusted kernel in order to provide proofs of identity. The kernel can prove the identity of legitimate applications, relying on the uniqueness of application identities and a secure in-host authentication protocol.
- **Application monitoring.** Our identification model enables the design and implementation of a sophisticated system call monitoring architecture that is used to enforce *application-level* access rights.

Contributions. We present our secure application identification model using which the kernel can identify and authenticate the running processes. We present the design of a modified challenge-response protocol to securely authenticate applications. Moreover, we design and implement a system call monitoring architecture to monitor the execution of the processes through their interactions with monitored system calls. We use standard benchmarks to evaluate the performance of our implementation and show that our system is feasible to implement without a significant performance penalty.

2. THE AUTHENTICATED APPLICATION FRAMEWORK

A major 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. 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.

In the diagram of Figure 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.

We introduce the design and implementation of the *Authenticated Application* (A2) framework to address the four steps required to provide strong identification of applications. The core idea of A2 is a novel identification model that is based on sharing unique symmetric keys between every installed application (that runs in the form of stand-alone 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. The design of A2 is driven by the following security goals:

1. Providing unforgeable identities that can be used by mandatory access control systems verify the legitimacy of the processes.
2. Enabling effective application-level access rights at the system call usage level.

To achieve our security goals, we define an application key as follows:

DEFINITION 1. *An application key is a string of characters s of length l generated by a cryptographic key generation function $f : l \rightarrow s$ such that with an appropriate length l, s is always unique and is computationally hard to guess.*

Each application key is generated using a trusted kernel helper (discussed in Section 2.4) and is used for a single application. Every process created by an application must be able to use the key to authenticate itself to the kernel. Application keys can be generated using secure symmetric encryption key generation functions such as AES [20].

2.1 Threat Model

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. We assume that legitimate applications may be vulnerable and thus allow downloading malicious code. However, legitimate applications may not contain malicious code and a malware cannot misuse a legitimate application without running as a stand-alone process. Specifically, malicious code running within the boundary of a legitimate process (such as

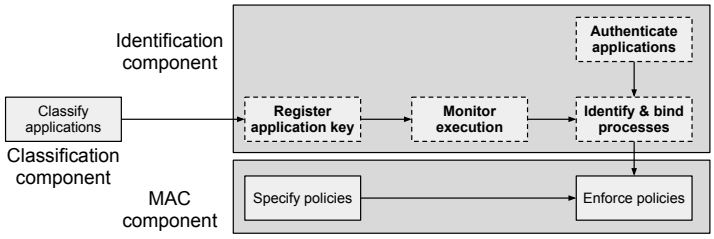

Figure 1: Conceptual process of protection against malware.

a malicious browser script or extension) is out of the scope of our work. Moreover, we assume the hardware, the installed firmware and standard operating system APIs are trusted. We also assume that an attacker has no physical access to the machine and does not know the user's credentials.

A2 is used to fundamentally distinguish legitimate processes from malicious ones. Although the identification model itself is not a malware detection method, it can be used to boost the security guarantees of MAC systems as well as application sandboxing solutions. Later in Section 2.5 we discuss the integration of our application identities with an existing a MAC system.

2.2 Design Overview

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 [4] of the cryptographic hash functions.

In the identification model of A2, applications are recognized as individual principals. Keyed applications 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.

The A2 framework consists of three main components: *Trusted Key Registrar*, *Authenticator* and *Service Access Monitor (SAM)* depicted in Figure 2. We implement the Authenticator and SAM as Linux kernel modules without modifying the kernel (see Section 3). 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 2.3.1) based on a token generation protocol.

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.

2.3 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 [13]. 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 3.

2.3.1 Token Generation Protocol

Our authentication protocol is used to generate identity

[1] A piece of executable code that runs in at least one standalone process.

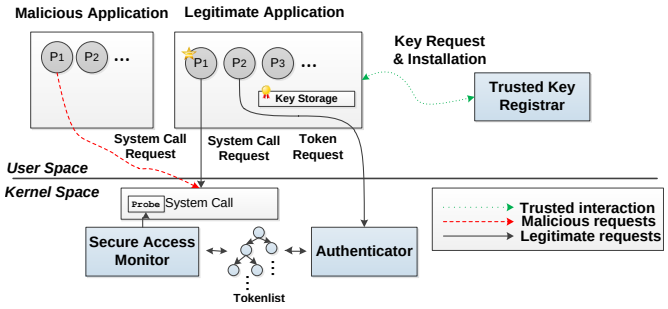

Figure 2: The access to selected system calls is monitored by A2. P_i denotes an application process.

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 3.2).

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 1 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 2. *A registered application is a piece of executable code that runs in the form of one or more stand-alone processes and is issued a secret key by the kernel.*

Definition 3. *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 4. *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 Verifies if the requesting application has a registered key. Otherwise, `malicious(p)`.
   - 2.2 Verifies if `p` has already established a token. If so, `malicious(p)`.
   - 2.3 `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.
   - 2.4 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) $h = \text{HMAC}(s,\texttt{p.pid},k)$ (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 $h' = \text{HMAC}(s,\texttt{p.pid},k)$. 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 `tlist` that is only readable by the verification module (i.e. SAM).

TGP prevents replay attacks since $h = \text{HMAC}(s,\texttt{p.pid}, k)$ 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.

2.3.2 Tokens Storage

The `tlist` is a data structure that is maintained by the Authenticator. Authenticator can only allow read access to `tlist` to be used by a verification module inside the kernel. In systems with heavily use of various types of software (especially multi-process software), `tlist` may grow relatively large. It is not efficient to store the `tlist` in a sequential list. That is because, the `tlist` will be frequently searched for tokens at the time of system call monitoring by the verification module. Therefore, it is beneficial to make use of binary search trees, which on average reduce the time of searching to O(log n) where n is the number of tokens in the list. Binary search trees take longer time for the insert operation compared to a normal sequential list (O(log n) as opposed to O(1)). However, our insert operation is not as critical as the search since the insert is less frequent than the search. Linux kernel include an API for red-black trees [3] (a balanced binary search tree), that can be used for maintaining the `tlist` . Red-black trees are used for the organization of virtual memory but are available to other Linux kernel functions or modules. These trees provide a search as efficient as that of the binary search trees.

Using the `tlist` data structure to store the tokens is necessary to avoid modifying the stock Linux kernel. While the performance penalty is modest (see Section 3.2), it is possible to modify the kernel to store the tokens in the process control block. Using the latter design choice, both search and insertion time remain constant. We further discuss this in Section 3.

2.4 Application Key Registration

Prior to performing any authentication, it is necessary for the kernel to generate and register the secret keys for legitimate applications. In this section, we present the key registration and revocation steps.

The secret key must be registered by the kernel at the installation time and must be stored in the application's code. 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.

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 exchanges the key information with the application in a secure system state. 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, the trusted registrar generates k and sends it back to the application. Otherwise, the application is removed and reported as malicious.
4. The application accepts k and stores it in its executable code.

As depicted in Figure 1, the original identity of the application is determined as part of a binary classification and certification process. The purpose of this certification is to verify the legitimacy of the application at the first place. To allow the installation of the application, the trusted registrar decision is based on the user's permission as well the result of the certification process. If either give a negative answer, the trusted registrar would not issue a key for the application. Existing binary analysis and certification solutions such as BitBlaze [23] can be utilized for this purpose.

2.5 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.

2.6 Security Analysis

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. This sandboxing 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. Al-

though 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.

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.

Scope of A2. 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.

We further discuss a detailed analysis of our security model in our technical report [2].

3. PROTOTYPE

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). Due to limited space, the details on the implementation of the trusted registrar is left as a future work.

3.1 Implementation

The Authenticator module uses the Linux kernel Cryptographic API [6] 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 2.3.1).

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 currently, implement the list of tokens as a sequential list. In our future implementations we are providing two options for storing the tokens. One is through the use of a red-black tree discussed in Section 2.3.2. Alternatively, Authenticator can store the token in the Process Control Block (PCB). The latter design would eliminate the search overhead but requires a modification to the kernel.

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 3.2).

3.2 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 [19]. 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. In addition to answering the questions mentioned earlier, we experiment with two open-source keyloggers and our key stealer malware to test A2's functionality against undesired software.

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 iteration. 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 `tlist`). In the final experiment, SAM verifies the `tlist` with a total of 300 stored tokens. The results of our experiments are shown in Figure 3. 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 `probe`s in the kernel functions. That is because, the average extra latency caused by the verification of the `tlist` (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 [19]. 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 4, 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%.

Our results show efficient system call performance without a significant penalty due to our monitoring architecture.

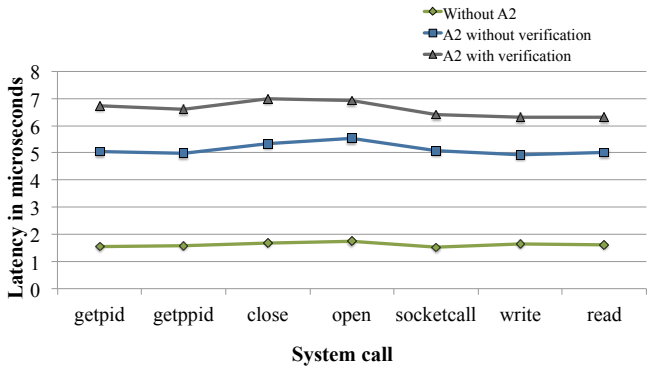

Figure 3: System call overhead measured in three experiments: No A2 modules running, A2 without any verification, and A2 running and performing verification on a list of 300 tokens.

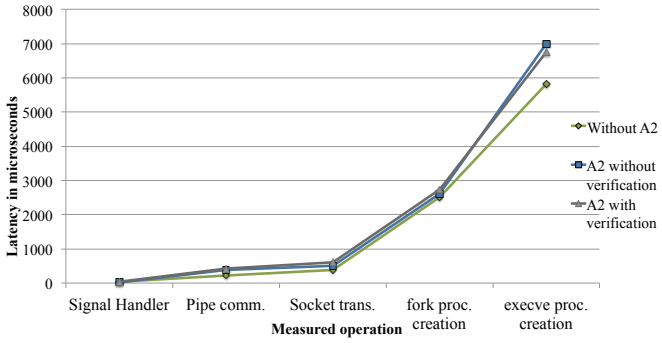

Figure 4: The latency caused by A2 modules for UNIX socket transactions and process creations.

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.

4. RELATED WORK

Techniques for protection against malware include the use of mandatory access control (MAC) systems, virtual machine monitors (VMM) to isolate untrusted software, application sandboxing, and a variety of other approaches such as hardware-based protection. In the following, we describe how A2 is related and compared to the mentioned approaches.

SELinux is a policy-based MAC system [17], which does not have a strong application identification mechanism that is independent of a particular user identity and does not rely on dynamic features such as a process ID. Grsecurity [1] is a policy specification platform similar to SELinux with a simplified specification language that suffers from the same identification problem. A more usable MAC solution is described in [16]. A cryptographic-based MAC system is the authenticated system call work by Rajagopalan et al. [22], which is closed in spirit to our A2 framework. The presented work in the authenticated system call 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. Moreoever, Jaeger et al. did pioneering work in kernel-based control of program behaviors, including regulating downloaded executable content [10] and general-purpose policy enforcement through intercepting inter-process communication [9].

As described in Section 2, the identification model provided as part of the A2 framework is complementary to existing MAC-based solutions. We provide strong and secure identification of running processes, which is a critical step before proper application-level access rights can be enforced. A2 can integrate with application-level policy-based MAC systems as described in Section 2.5.

Isolation using virtual machine monitors and application sandboxing has been addressed in a number of projects. VMM-based solutions such as [15, 12] make use of VMM to provide high-assurance isolation and thus monitoring of untrusted software. 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. Nevertheless, A2 can also be integrated with VMM-based solutions to provide high assurance on the identity of processes within untrusted environments.

Application sandboxing solutions such as Vx32 [7], UserFS [14], and BLADE [18] are used to isolate undesired code from being executed to maintain integrity and confidentiality of the execution environment. The A2's identification mechanism enables strong and clear sandbox of undesired code that runs as stand-alone processes. Techniques similar to the ones described in the cited literature can be used to further automate the decision on legitimacy of executable code before the application registration step (Section 2.4).

Other approaches for protecting system resources against unauthorized applications include signature-based malware detection (proved to be ineffective against zero day attacks) [5], integrity preserving based on information flow such as PRIMA [11], and Trusted Platform Module [24]. These approaches provide valuable security solutions. However, our security model differs in providing provable identity to native applications.

5. CONCLUSIONS AND FUTURE WORK

We presented a novel identification model that provides strong and unforgeable application identities and binds the processes to their corresponding applications at runtime. Our identification 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 inside a kernel. In the A2 framework, malicious processes are completely isolated to prevent them from attacking other processes or achieving any attack goals.

The identification model of A2 is simple to implement and is highly portable. We introduced the idea of an authentication protocol between a user

Our evaluation results indicate the feasibility of using cryptography for the purpose of identifying running pro-

cesses. 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.

6. REFERENCES

[1] grsecurity. http://www.grsecurity.net/.

[2] H. M. J. Almohri, D. Yao, and D. Kafura. Identifying native applications with high assurance. Technical Report, Department of Computer Science, Virginia Tech, 2011.

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

[4] J. A. Buchmann and J. A. Buchmann. Cryptographic Hash Functions. In S. Axler, F. W. Gehring, and K. A. Ribet, editors, *Introduction to Cryptography*, Undergraduate Texts in Mathematics, pages 235–248. Springer New York, 2004.

[5] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, may 2005.

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

[7] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

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

[9] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[10] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.

[11] 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*, SACMAT '06, pages 19–28, New York, NY, USA, 2006. ACM.

[12] 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*, 13:12:1–12:28, March 2010.

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

[14] T. Kim and N. Zeldovich. Making Linux protection mechanisms egalitarian with UserFS. In *Proceedings of the 19th USENIX conference on Security*, pages 13–27, Berkeley, CA, USA, 2010. USENIX Association.

[15] 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*, ICPADS '10, pages 706–711, Washington, DC, USA, 2010. IEEE Computer Society.

[16] N. Li, Z. Mao, and H. Chen. Usable mandatory integrity protection for operating systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 164–178, may 2007.

[17] 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, 2001. USENIX Association.

[18] 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*, CCS '10, pages 440–450, New York, NY, USA, 2010. ACM.

[19] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[20] NIST. Announcing the advanced encryption standard (AES). Federal Information Processing Standards Publication 197, November 2006.

[21] C. Parampalli, R. Sekar, and R. Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 156–167, New York, NY, USA, 2008. ACM.

[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*, 3:216–229, July 2006.

[23] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, N. James, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS'08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[25] T. Wobber, A. Yumerefendi, M. Abadi, A. Birrell, and D. R. Simon. Authorizing applications in singularity. In *Proceedings of the 2nd European Conference on Computer Systems*, EuroSys '07, pages 355–368, New York, NY, USA, 2007. ACM.

[26] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security module framework. In *Proceedings of the 11th Ottawa Linux Symposium*, 2002.