Identifying Native Applications with High Assurance

ABSTRACT

The work described in this paper investigates the problem of identifying and deterring stealthy malicious processes on a host. We point out the lack of strong application identification in main stream operating systems. We solve the application identification problem by proposing a novel identification model in which user-level applications are required to present identification proofs at run time to be authenticated by the kernel using an embedded secret key. The secret key of an application is registered with a trusted kernel using a key registrar and is used to uniquely authenticate and authorize the application. We present a protocol for secure authentication of applications. Additionally, we develop a system call monitoring architecture that uses our model to verify the identity of applications when making critical system calls. Our system call monitoring can be integrated with existing policy specification frameworks to enforce application-level access rights. We implement and evaluate a prototype of our monitoring architecture in Linux as device drivers with nearly no modification of the kernel. The results from our extensive performance evaluation shows that our prototype incurs low overhead, indicating the feasibility of our model.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Operating system, malware, cryptography

1. INTRODUCTION

Security of personal computers mainly depends on the reliability of the running processes and the way that the operating system kernel controls and manages the execution. Operating system kernels enforce minimal restrictions on the applications permitted to execute, resulting in the ability of malicious programs to abuse system resources. For example, a spyware process can freely make use of networking system calls to transmit malicious traffic. Thus, once a malicious process is created on a victim machine, it can successfully use kernel services to reach its goals. To effectively prevent infection by malicious processes, the kernel must be able to strongly identify the applications, distinguish the benign process from the malicious ones, and prevent the misuse of kernel services.

In this work, we address the problem of sandboxing malicious processes under the assumption of a trusted kernel. We point out the lack of proper identification of the running processes that is needed to detect and prevent the execution of undesired processes at runtime. In mainstream operating system kernels (such as the Linux kernel), applications are identified based on executable file names, the installation path or the process identification (`pid`). However, none of these identification methods strongly bind a running process to the corresponding executable code as they are subject to forge and change. Existing security solutions based on the Linux Security Modules such as AppArmor [12] use the application's installation path as an identification based on which access rights are enforced. The installation path is a weak identification, as the malware may try to relocate a legitimate application and assume its path.

Existing work in the area of protecting the system from malicious activities spans a wide range of approaches including policy-based access control, system call inspection, isolation using virtual machine monitors (VMM), and application sandboxing. Fine-grained policy-based access controls using system call monitoring have been proposed by the Security Enhanced Linux (SELinux) [22]. System call monitoring techniques using virtual machine monitors have also been studied [11]. In addition, application sandboxing provides a mechanism to run untrusted code in a protected environment [10].

Despite providing useful security solutions, existing approaches do not explicitly address the problem of application identification. In [38], 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 applica-

tion (using application name) with the user's access rights and does not provide an explicit application identification mechanism that can prove its origins.

In this paper, 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 at runtime. Using our identification model, we achieve the following:

- **Application identification.** Applications with registered identities, are able to authenticate themselves 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 authentication protocol. Undesired applications are unable to authenticate themselves to the kernel due to the lack of known identities. Consequently, these applications are sandboxed and restricted from performing sensitive operations such as the use of the kernel's networking subsystem.

- **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 as well as sandboxing undesired applications.

Contributions. We present the idea of using unique secret keys to identify application processes at runtime. To use this strong identity, we design a secure authentication protocol for an application to authenticate itself to the kernel. Moreover, we develop a system call monitoring architecture that monitors system calls made by the running processes and verifies the corresponding application identities prior to deciding on the access rights. The system call monitoring instantiates our novel application identification model to identify application processes. The implementation prototype consists of two Linux kernel modules to securely authenticate applications and to verify their identities at the time of using system calls. Our implementation requires minimal modifications to commodity applications with nearly no modification to the kernel. Our evaluation results indicate the feasibility of our system call monitoring approach without a significant performance penalty.

Outline. We present the design of *Authenticated Application* framework, discuss secure authentication and authorization of applications, and provide an in-depth discussion of achieved security properties and guarantees in Section 2. Section 3 discusses our implementation. In Section 4 we describe our experimental results including measuring the overhead caused by the system call monitoring, scalability tests, and verifying the ability of our system call monitoring architecture in limiting the activities of spyware. In Section 5 we discuss the related work. We conclude in Section 6 and discuss our future work.

2. THE AUTHENTICATED APPLICATION FRAMEWORK

Stealthy malware running as a stand-alone process, once installed, can freely execute benefiting from the privileges provided to the user account running the process. Traditional operating system kernels are not designed to detect malicious behavior, or identify malicious processes at runtime. An existing technique to guard against malicious processes is through the use of behavioral analysis. This approach suffers from advanced (and newly discovered) attacks that are capable of bypassing the detection scope [30]. An orthogonal strategy to protect against malicious processes is to be able to identify malicious processes without relying on the behavior of these processes.

Our security goal is to design an application identification model with the following properties:

1. **Unforgeable identities.** Uniquely identifies the applications by generating secure credentials for the running processes.

2. **Application isolation.** Isolates detected undesired applications by strictly limiting their activities at runtime.

3. **Application-level access rights.** Enables effective application-level access right enforcement using generated process credentials.

We achieve our goals by using our proposed identification model discussed in the rest of this section.

2.1 Design Overview

We design the *authenticated application (A2)* framework, which is capable of providing secure application identification at runtime. 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 [6] of the cryptographic hash functions [1]. The proved identity of the process is later used in our system call monitoring architecture to decide on *application-level* access rights.

In our identification model, applications are recognized as individual principals. Keyed applications are the most privileged applications while keyless 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 by monitoring a set of critical system calls such as the `open` system call.

Our identification model is based on the assumption that the kernel's code and memory are trusted. User-level processes, unless authenticated, are possibly malicious. Using A2, we preserve the security of kernel by restricting the activities of unauthorized processes.

The A2 framework consists of three main components: *Trusted Key Registrar*, *Authenticator* and *Service Access Monitor (SAM)* depicted in Figure 1. We implement the

[1]Given a secure cryptographic hash function, it is easy to compute the hash of any message. It is infeasible to (i) generate the original message given its hash, (ii) to find a common hash for two different messages, and (iii) to modify a message without modifying its hash.

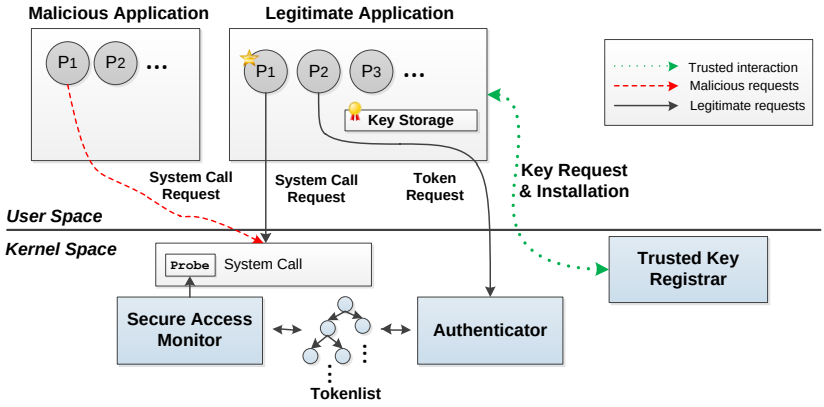

Figure 1: The access to selected system calls is monitored by A2. P_i denotes an application process.

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 the component responsible for installing a key for the application and registering the application with the kernel. The application interacts with the trusted key registrar in a trusted system state to receive a secret key. The trusted key registrar stores the same key and register 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 needs to ensure the identity of the process and prevent malware from spoofing a legitimate process. The Authenticator implements our token generation protocol to authenticate applications.

Tokens are statements that are certified by the Authenticator, to identify a process when interacting with the kernel. A token is valid until its process is terminated. The tokens are maintained by the Authenticator and can be accessed and verified by the Service Access Monitor.

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.

2.2 Secure Authentication of Applications

In order to identify the application associated with a running process, the process must be able to prove its identity to the trusted kernel. This proof of identity needs to be based on the secret key that is embedded in the application's code and is known to the kernel. Thus, it is necessary for a newly created process to securely authenticate itself using the application's secret key. Our authentication mechanism is based on three generic steps. First, the kernel needs to send a random nonce to the application process. The process hashes the nonce with the secret key and returns the nonce back to the kernel. The kernel regenerates the hash of the string and compares the result with the hash claimed 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 for the kernel can directly access the application's key and verify 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 design an authentication protocol that can be executed on a secure socket between the process and the kernel. This method can be realized using a memory-based socket known as the `/proc` file system [17]. The advantage of using the `/proc` file system is that it is easily 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.

2.2.1 *Token Generation Protocol*

Our authentication protocol is used to generate identity tokens for legitimate applications. The identity tokens are later used to identify the applications 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 for the purpose of identifying the application. On the other hand, the verification of identity is needed when the process is accessing restricted system resources. This separation improves the system call monitoring overhead (see Section 4). Thus, it is important to have identity tokens stored for the running processes to be used whenever an authorization is needed. We design a token generation protocol (TGP) that is used to authenticate individual processes based on the keys of the corresponding applications.

We first define a *registered application*, an *identity token* and the *Authenticator* as follows:

DEFINITION 1. *A registered application is a piece of executable code that has been issued a secret key by the kernel in a trusted system state.*

DEFINITION 2. *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`.

Properties of the identity tokens are as follows:

- Generated when the process starts through the token generation protocol.
- Unique for each process and valid until the its termination.
- Can be generated only by the Authenticator (defined below).
- Readable only by the Secure Access Monitor.

DEFINITION 3. *The Authenticator is a kernel service that implements the token generation protocol and is responsible for creating and maintaining identity tokens for registered applications.*

The steps of the token generation protocol are as follows:

1. Process p sends a request to the Authenticator A for an identity token. The process must specify the application name that needs to be authenticated.
2. Upon receiving the request:
   - 2.1 A verifies if the requesting application has a registered key. Otherwise, $malicious(p)$ (p is reported as a malicious process and the protocol is terminated).
   - 2.2 A checks whether 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 checks 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 policy.
   - 2.4 A generates a random nonce n and sends it to p. g. 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 produces the message authentication code (MAC) using a secure hashing function $h = hmac(n, p.pid, k)$ where $p.pid$ is the process ID of p and k is its secret key. h is sent to A.
4. If t has expired, A discards the authentication. If p is still executing, it will be terminated to prevent a race condition.
5. A verifies h by re-computing the MAC $h' = hmac(n, p.pid, k)$. If $h = h'$, then a token $tk = (pid, p.app)$ (where p.app is the name of the application) is generated. tk is valid until the termination of p. Otherwise, $malicious(p)$.
6. tk is stored in a shared data structure *tokenlist* that is only readable by the verification module in the trusted kernel.

The protocol can be executed on a transparent socket between the Authenticator and the application. This does not decrease the security level as the protocol uses HMAC to hide the key and is not vulnerable to man-in-the-middle (and replay) attacks. Moreover, we restrict the number of authentication (or key agreement) requests from an application to prevent denial-of-service attacks.

2.2.2 Tokens Storage

The *tokenlist* is a data structure that is maintained by the Authenticator. Authenticator can only allow read access to *tokenlist* to be used by a verification module inside the kernel. In systems with heavily use of various types of software (especially multi-process software), *tokenlist* may grow relatively large. It is not efficient to store the *tokenlist* in a sequential list. That is because, the *tokenlist* 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 $O(log_n)$. 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 provides a special kind of tree API, denoted as red-black trees, that can be used for the purpose of maintaining the *tokenlist* [5]. 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.

2.2.3 Key Management

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.

Key registration. 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 (hereafter referred to as the registrar) that is used to register applications' keys in the kernel and the application. The registrar exchanges the key information with the application in a secure system state. The steps taken by the registrar are as follows:

1. The application is started for the first time and requests a key from the registrar.
2. The registrar verifies if the application was previously issued a key and if the application is designated legit-

imate either by the user or after an application certification process.

3. If verification passed, the 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.

The original identity of the application is determined as part of a binary certification process. The purpose of this certification is to verify the legitimacy of the application. To allow the installation of the application, the registrar decision is based on the user's permission as well the result of the certification process. If either give a negative answer, the registrar would not issue a key for the application. Existing binary analysis and certification solutions such as BitBlaze [34] can be utilized for this purpose.

Key revocation. In the event that the system administrator decides an application should no longer be registered as a keyed application, the registrar can be used to revoke the key from the kernel. In order to make sure that the key is completely unusable, we flag the key as revoked in the key storage to indicate that an application with this key cannot be registered.

Key protection. To protect the secret key from being revealed to other applications, A2 restricts read access to identified application binaries. Further, a file system encryption can be used to encrypt the binary code of an application using a master key known to the kernel. This can be done by implementing a kernel module, which will be responsible for encryption and decryption of the application binaries without the need to modify the kernel. We leave this issue for our future work.

2.3 Secure Authorization of Applications

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:

1. Monitor designated system calls.

2. Verify the identity of the process making the system call.

3. Enforce application-level access rights according to policy file.

4. Log malicious activities.

The Secure Access Monitor can be integrated and with a policy specification language to benefit from existing scholarly work in this area [2, 27, 37, 38]. A policy file must specify application categories, system call names, and optionally permitted system call usage frequencies. A sample policy file is depicted in Table 1. SAM uses the binary decision for allowing or disallowing the use of a system call. In this file, permitted frequency is omitted indicating that the corresponding system call usage frequency is unlimited.

The categories of applications are determined and specified as part of the application specification process when an application is being registered. We restrict the behavior of application categories by limiting the way they make use of system calls. For example, in the sample policy file in Table 1, we forbid all categories of applications from completing an `open` system call on executable binary files. This is a necessary policy, which we enforce to maintain the privacy of the secret keys.

A more detailed discussion on the integration of SAM with a suitable policy specification language and the challenges associated with enforcing a rich policy in the kernel environment, is left for our future work.

2.4 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. Although a legitimate application such as a web browser may allow malware to be downloaded, A2 prevents the downloaded malicious code to reach its ultimate goal.

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

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. As a result, A2 can have the extra benefit of distinguishing Java programs running under the same process name. 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

App category / System call	open (exec)	open (non-exec)	socket	execve	fork	ipc	kill
Web browser	0	1	1	1	1	1	1
Social networking	0	1	1	1	1	0	0
Text editor	0	1	0	0	1	0	0
Miscellaneous	0	1	0	0	1	1	0
Unidentified	0	1	0	0	0	0	0

Table 1: A sample policy file. SAM uses 1 or 0 to allow or disallow the use of a system call respectively.

using a shell terminal. It is the responsibility of the process to perform the authentication and identify itself.

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

3. IMPLEMENTATION

We realize a prototype of the A2 framework in the Linux Operating System (Debian 2.6.32). We have implemented SAM and the Authenticator as Linux kernel modules. These two modules can be loaded using `root` privileges as needed. SAM depends on the Authenticator in order to verify the tokens. In the following we describe the implementation of our prototype.

3.1 Implementation of Authenticator

The Authenticator module relies heavily on the Linux kernel Cryptographic API [9]. The API supports various hashing algorithms and provides a number of ways to perform encryption and decryption. It works directly on memory pages and uses a special data structure called `scatterlist` to hold the required data.

We have a complete implementation of the TGP using the Cryptographic API. To implement TGP, we use the HMAC [4] algorithm to hash the data with the secret key k serving as a secure signature of the application to be presented to the kernel.

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.

We define two functions for reading and writing operations to the protocol file in `/proc` file system. The `read_protocol_file` function is executed when the user reads the file. For writing to the challenge file, we define the function `write_protocol_file`. In this function, our module reads the data that is written by the user. The Authenticator only responds to one request, which is the generation of a token by a running process. The request verifications described in Section 2.2 is performed before the protocol can proceed.

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

3.2 Implementation of SAM

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. Each time SAM needs to verify a process' identity, it needs to search through a list of currently valid tokens that are maintained by the Authenticator. We currently, implement the list of tokens as a sequential list. However, in our future implementations we are providing two options for storing the tokens. One is through the use of a red-black tree is discussed in Section 2.2.2 and the other is by storing the token in the Process Control Block (PCB). The latter design would eliminate the search overhead but requires a modification to the kernel. We are implementing both options to increase the flexibility of using A2.

SAM uses `kprobe` to hook into existing system calls and redirect the access to the SAM code. The `kprobe` is a built-in API in Linux kernel that allows kernel modules to monitor kernel functions by placing probes in desired functions. When a probe is triggered, `kprobe` causes an interrupt and transfers the control to SAM. 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 4).

SAM can be designated to monitor any function in the kernel. In our experiments, we run SAM with at least 5 monitored functions. When the module is loaded, it requests the installation of probes for each monitored function. A universal handler receives the control when a probe is triggered. We verify the system call being monitored by checking the `eax` register and start searching for a token for the process. We allow or deny access to the monitored system call depending on the token and the access control table described in Section 2.3. Our current policies are simple for the purpose of performing runtime overhead experiments. As described earlier, we are discovering the integration of advanced policy specification languages to allow sophisticated policies into the A2 framework.

4. EXPERIMENTAL 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?
- What are the most frequently used system calls in a normal load?
- How does A2 perform, when the system is overloaded with a large number of tokens?

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 [24]. 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.

4.1 System Call Monitoring Overhead

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 `tokenlist`). In the final experiment, SAM verifies the `tokenlist` with a total of 300 stored tokens. The results of our experiments are shown in Figure 2. 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 `tokenlist` (that already contains a total of 300 tokens) is 29.03%. It is expected that by modifying the kernel and manual installation of the hooks into the kernel function, we can achieve a considerable reduction of the overhead. However, the advantage of using our current kernel modules is the simplicity and protability of the solution.

In addition, we recorded the number of system calls invoked by running processes to our selected subset of critical system calls in a short 10-minute period. Based on our results, `open` and `socketcall` (the interface to all networking

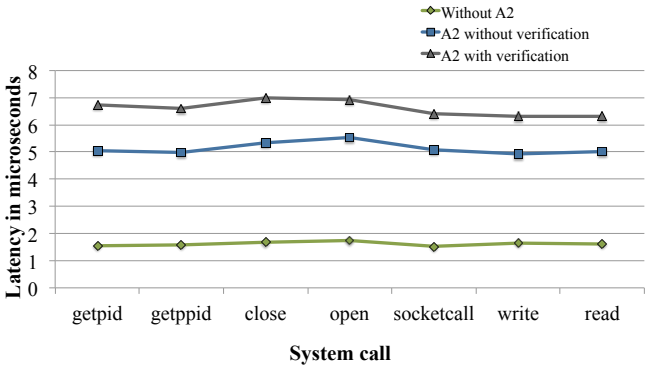

Figure 2: System call overhead measured in three experiments: No A2 modules running, A2 without any verification, and A2 running and performing verification on a tokenlist of 300 tokens.

system services such as `sendmsg`) have received most system call requests. Our results also suggest that `open` and `socketcall` are used by most applications running in our experiment. A total of 30 applications were measure that attempted to access the system calls. In this experiment, 78% of calls to `socketcall` were made by a cloud-based file system software and 12% of the calls belonged to the web browser for a total of 90%.

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 [24]. 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 3, 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%.

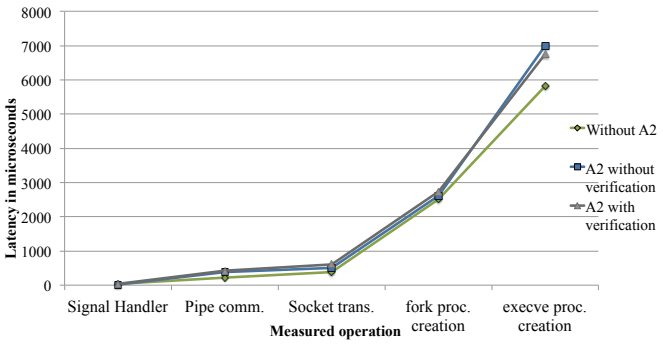

Figure 3: The latency caused by A2 modules for UNIX socket transactions and process creations.

4.2 Scalability Evaluation

Figure 4(a) shows an experiment to measure the scala-

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

Figure 4(b) shows an experiment for measuring the total overhead caused in a 5-minute-long trace for searching for a token based on a global monitoring frequency f. With $f = 50$, an inspection is performed for every 50 calls to `sys_socketcall`. The corresponding average overhead associated with our access verification is quite small and efficient, which is under 5 seconds over a five-minute period. Higher monitoring frequencies incur higher latency. A higher monitoring frequency provides a stronger secure guarantee for the overall system. (If $f = 1$, SAM monitors every single system call request for monitored system calls.) Therefore, the malware behavior can be detected immediately from the first call made. In contrast if $f > 1$, SAM skips f number of system calls before it verifies the `tokenlist` to increase the performance. In case of restricting access to highly critical system calls, the malware may be able to manage to fall into the interval after the last `tokenlist` verification and before the next verification. f must be as low as possible, especially for those system calls that are least frequent such as `sys_kill`. The f value may be slightly increased for highly frequent system calls like `sys_open`. On the other hand, even with a $f = 1$ for less frequent system calls, the system has a negligible system performance downgrade.

4.3 Spyware Detection.

We use two open source Linux keyloggers to test SAM's monitoring: Logkeys Linux keylogger (Logkeys) [2] and LKL Linux KeyLogger (LKL) [3]. The keyloggers may be dropped through drive-by-download exploits or dropped by other malware. We started these keyloggers while SAM was running to intercept the critical system calls. In this experiment, SAM was in alert mode allowing the execution of the program and generating alerts when necessary. Both keyloggers were immediately detected by SAM and alerts were reported. Logkeys caused SAM to generate alerts for `sys_unlink` and `sys_kill` as they were monitored by SAM. The alerts generated for LKL were due to the use of `sys_socketcall`.

This experiment further verfiies that SAM is able to detect any malware that needs to run as a user process and initiate a system call that SAM is set to monitor at the time the malware runs. We can detect those types of malware that try to modify binary codes to change the behavior of the program or to steal the key. For instance, we wrote a sample malware that tried to steal the key from our customized program by reading its binary code and inspecting the strings. Since we hard-coded the key in the program without encryption, the sample malware easily extracted the key. However, SAM detected the sample malware while it was in the process of finding the key.

[2]http://code.google.com/p/logkeys
[3]http://sourceforge.net/projects/lkl

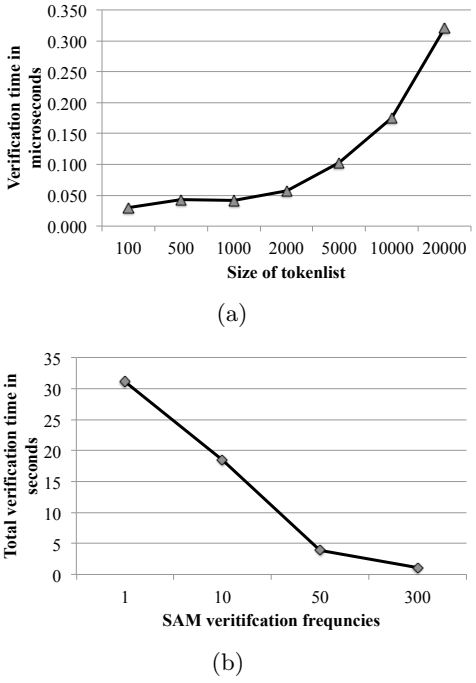

Figure 4: (a) Verification time associated with one request in milliseconds increases with the size of token list. (b) Total verification time in seconds in a 5-minute-long trace.

4.4 Summary of Experiments

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

5. RELATED WORK

The work that is the closest to ours in spirit is the authenticated system call work by Rajagopalan et al. [31, 32]. The authors propose the use of message authentication code (MAC) in monitoring system calls, and present an automated method of rewriting the application binary to include the MAC in the arguments of a system call. The kernel has a secret key (which is not known to the application) to recompute and verify MAC. The presented work is limited to providing identities (the MAC) to individual function calls to system calls in an application. Thus, it does not provide an identity to the application itself.

Policy-based security models are the subject of a number of research projects including policy specification languages [19, 28, 21]. Jaeger et al. did pioneering work in

operating-system security and security kernel architecture for OS-level control of program behaviors, including regulating downloaded executable content [14] and general-purpose policy enforcement through intercepting inter-process communication in OS [13]. SELinux is another policy-based mandatory access control system [22]. It enforces access control via monitoring system calls and enforcing a set of policy rules. The complexity of managing the policies is well documented [26]. SELinux differs with our proposed model in that it relies on standard Linux user identities to decide on the access rights. We instead, provide a provable application-level identification mechanism that is independent of a particular Linux user. Grsecurity [1] is a policy specification platform similar to SELinux with a simplified specification language that suffers from the same identification problem.

System call monitoring is an ongoing research direction in the area of protection against malware [11]. Some of the research has focused on the use of virtual machine monitors (VMM) to monitor system calls [3, 20, 16, 29]. Using system call mining has been explored in [7, 25, 36].

Application sandbox is a mechanism to allow execution of untrusted code on protected hosts. Recent sandbox proposals include Vx32 [10], UserFS [18], and BLADE [23]. Application sandboxing is a promising approach that can be combined with our application identity model to produce a comprehensive protection solution.

There is a variety of other approaches for protecting system resources against unauthorized applications. These approaches include signature-based malware detection (proved to be ineffective against zero day attacks) [8], integrity preserving based on information flow such as PRIMA [15], and Trusted Platform Module [33, 35]. These approaches provide valuable security solutions. However, our security model differs in providing provable identity to native applications.

6. CONCLUSIONS AND FUTURE WORK

We presented a novel application identification model that provides strong and unforgeable application identities and binds process to their corresponding applications at runtime. Our identification model is combined with a new system call monitoring architecture that verifies process identities. 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 sandboxed to prevent them from attacking other processes or achieving any attack goals.

Our identification model is simple to implement and is highly portable. We introduced the idea of an authentication protocol between a user application and the kernel. The advantage of taking this approach is eliminating the need for kernel modification and re-build. On the application side, such a protocol is simple to implement and requires minimal modification to the application. We developed a client application that implements the TGP, which contains a total of 196 lines of code dedicated to the protocol.

Our evaluation results indicate the feasibility of using cryptography for the purpose of identifying running processes. We achieve this result by separating the authentication from the monitoring. Therefore, there is virtually no performance penalty due to the use of cryptographic functions.

In current implementation of A2, we use simple policies for our the secure access monitor. In future, we plan to provide an interface for the secure access monitor to integrate with existing policy specification languages and frameworks. In addition, we are investigating the challenges with regard to interpreting and effectively enforcing specified policies.

7. REFERENCES

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

[2] C. Ardagna, E. Damiani, S. D. C. di Vimercati, and P. Samarati. XML-based access control languages. *Information Security Technical Report*, 9(3):35 – 46, 2004.

[3] F. Baiardi, D. Maggiari, D. Sgandurra, and F. Tamberi. Transparent process monitoring in a virtual environment. *Electronic Notes on Theoretical Computer Science*, 236:85–100, April 2009.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Advances in Cryptology*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 1996.

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

[6] 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.

[7] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the the 6^{th} joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, ESEC-FSE '07, pages 5–14, New York, NY, USA, 2007. ACM.

[8] 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.

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

[10] 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.

[11] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 418–430, Washington, DC, USA, 2008. IEEE Computer Society.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] S. Lachmund. Auto-generating access control policies for applications by static analysis with user input recognition. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 8–14, New York, NY, USA, 2010. ACM.

[20] 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.

[21] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin. Access control policy combining: theory meets practice. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, pages 135–144, New York, NY, USA, 2009. ACM.

[22] 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.

[23] 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.

[24] 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.

[25] K. Morik, F. Jungermann, N. Piatkowski, and M. Engel. Enhancing ubiquitous systems through system call mining. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 1338–1345, Washington, DC, USA, 2010. IEEE Computer Society.

[26] Y. Nakamura and Y. Sameshima. SEEdit: SELinux Security Policy configuration system with higher level language. In *Proceedings of the 2009 Large Installation System Administration Conference (LISA)*, 2009.

[27] Q. Ni and E. Bertino. xfACL: an extensible functional language for access control. In *Proceedings of the 16th ACM Symposium on Access control Models and Technologies*, SACMAT '11, pages 61–72, New York, NY, USA, 2011. ACM.

[28] Q. Ni, E. Bertino, and J. Lobo. D-algebra for composing access control policy decisions. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 298–309, New York, NY, USA, 2009. ACM.

[29] K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 2116–1221, New York, NY, USA, 2008. ACM.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] X. Tian, X. Cheng, M. Duan, R. Liao, H. Chen, and X. Chen. Network intrusion detection based on system calls and data mining. *Frontiers of Computer Science in China*, 4:522–528, December 2010.

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

[38] 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.