An MTM based Watchdog for Malware Famishment in Smartphones

Osman Ugus and Dirk Westhoff

Department of Computer Science, Hamburg University of Applied Sciences (HAW)
Berliner Tor 7, 20099, Hamburg
{osman.ugus, dirk.westhoff}@haw-hamburg.de

Abstract: Due to their various wireless interfaces, a continuously increasing number of fishy Apps, and due to their raising popularity, smartphones are becoming a promising target for attackers. Independently of the point of entrance, malwares are typically attached to an App to perform their malicious activities. However, malware can only do harm on a smartphone if it is executed. We thus propose a malware prevention architecture for smartphones that exploits App signatures, process authentication during their creation, and process verification during their execution and scheduling. The proposed security mechanism will allow a smartphone to run only those Apps which are classified as trusted (i.e., signed Apps) and which are not modified after their installation. The trust derived from the App signatures and a Mobile Trusted Module (MTM) is propagated through the processes until their execution via process authentication. The MTM serves as a trust anchor for our malware famishment in smartphones. This work presents our solution conceptually. We will soon start with a proof of concept implementation and a performance study using a software based MTM emulator.

1 Introduction

Companies are increasingly considering to migrate from conventional mobile phones to the new smartphone generation. This would allow their employees to make phone calls, read company emails, and pursue their works with customized Apps from one device e.g., while being on a business trip. Today's smartphones are equipped with various interfaces such as GSM/UMTS, WLAN, BT, and NFC. Due to possible security exploits via these interfaces, smartphones are much more vulnerable to a bunch of attack vectors than conventional mobile phones [NKZS10]. Malwares are typically attached to an App and perform their malicious activities whenever that program is executed. Once malware is uploaded to a smartphone via one or another channel and may get sufficient access rights, sensitive company information may eventually be revealed. Thus, company administrators will only consider smartphones as a part of the company's network topology, if the use of the above described services on smartphones comes with a moderate level of security risk, if at all.

As it is not always feasible to protect all channels against possible threats, we propose the following approach for protecting a smartphone against incoming malware and malicious Apps[1]: we simply allow malicious code to reside on the smartphone. However, it is our objective to *prevent* the malicious code from execution. The prevention of execution is achieved via App signatures, process authentication during their creation, and process verification during their execution and scheduling during assigning CPU slots to processes. The hash of Apps which are allowed to run on the smartphone are signed by a *Mobile Trusted Module* (MTM). Then, only those Apps which are intact and classified as trusted by the MTM are executed. Furthermore, we allow a creation of a new process only if the process which initiates the creation is trusted. The trust is extended from the first process `init()` whose trust is assured by the MTM via Message Authentication Codes (MAC) and the verification of the signature of Apps being executed. Finally, we modify the scheduler such that it assigns to a process CPU if it was authenticated during its creation. All other running processes are rigorously forbidden to receive CPU slots. Note that such a harsh security policy perfectly fits to the use case in which companies equip their employees with smartphones but want to have a strict control over all software legitimately running on the device. To establish a chain-of-trust we assume that an MTM can be used as a trust anchor. We believe that, although at this point in time no concrete MTM is available, in the near future smartphones will be equipped with such hardware components. For our proof of concept implementation, we plan to use a software based MTM implementation [EK11] based on the TPM emulator [SSM11].

The remainder of this work is organized as follows: Section 2 provides related work on the use of MTM for smartphone security, Section 3 introduces our MTM based watchdog architecture, Section 4 provides a security evaluation, and finally this article is concluded in Section 5.

2 Related Work

The Mobile Trusted Module (MTM) [MTM10] is a security module that provides a root-of-trust for integrity measurement in mobile devices. Integrity measurement is the collection of platform characteristics that affect the trustworthiness of that platform and putting digests of those metrics in the MTM. The integrity measurements are stored in immutable locations called Platform Configuration Registers. Transitive trust is used to extend the trust to the system states and program code that are not in the MTM. For this reason, the integrity measurement starts from a well-know state whose digest is known to the MTM such as a power on self-test. A chain of trust over the system is established by applying the transitive trust between the states by an Integrity Measurement Architecture (IMA) [SZJvD04].

SELinux [SEL09] ensures Mandatory-Access-Control (MAC) as an additional access control to the conventional Discretionary Access Control (DCL) for Linux. SELinux uses policies to allow access between objects and subjects. Policy-reduced integrity measurement architecture (PRIMA) [JSS06] is an IMA Architecture based on SELinux. Other IMA solutions that are not specific to smartphones include [NAZA09, LWPM07].

[1]Security is provided if the vulnerability/attack is already known in advance.

An attestation mechanism using an MTM for smartphones (Android) was recently proposed in [NKZS10]. As there is no hardware implementation of the MTM for smartphones available yet, the authors used in their work a software implementation emulating the MTM. The MTM is used for measuring the integrity of the system and reporting it to a remote attestation entity. The integrity measurements are performed at the operating system level and at the application level which allows to detect malicious Apps and any modification made by a malware on the classes loaded on the platform. The integrity of the system is verified by a remote attestation entity by comparing the signed integrity measurements (i.e., hash values of the Apps and system classes) received from the MTM with the hash values that the system in a trustworthy state must have. Thus, this approach allows a remote entity to detect if a smartphone is infected by a malware and the Apps installed on the smartphone are intact. However, it provides no countermeasure to prevent the execution of malware. Since continuously remote monitoring the integrity of a smartphone is not a realistic option, the attacker has sufficient time to perform malicious activities on the smartphone between two attestation phases and before a preventive action is taken. On the contrary we propose a mechanism that detects any modification in the system and *immediately prevents* the smartphone from running modified code.

3 MTM based Watchdog Architecture

Since our approach is based on processes and scheduling under Linux-kernel on top of which Android is built, we briefly introduce it [BC00]. A process is an instance of a program in execution. Every process is identified by a unique integer called the process ID (`pid`). Scheduling manages switching between processes and selecting a new process to run. A process switch is performed when the time slice (i.e., `quantum`) of the active process expires. The scheduler selects the process with the highest `priority` to run whenever it performs a process switch. The process priority is dynamically updated by the scheduler for a fair CPU share between all processes. All the information related to the scheduling is stored in the *process descriptor* whereas three parameters in a process descriptor are of particular importance for this work: `need_resched`, `priority`, and `counter`. The `need_resched` field is set to invoke the scheduler when the timer interrupt terminates. The `priority` field determines the likelihood of a process for CPU assignment from the scheduler. The `counter` is the CPU time left to the process before its quantum expires. The relevant data structures for the process descriptor are the process tabular (PT) and the various process control blocks (PCB). For each running process, the PT contains a link to its process descriptor which contains the process's relevant information for storing respectively reloading it when the scheduler initiated context switching. When a malicious process is detected, the MTM based watchdog proposed in this work sets the `need_resched field` of the current process and assigns the `priority` and `counter` fields of a malicious process to minimum to block it from execution.

For a smartphone that eventually accommodates malware we propose to control i) the execution of Apps such that only trusted Apps signed by the MTM are executed. Any modification made on an App is detected when it is started, hence, modified Apps are prevented from running on the system. We propose to extend ii) the scheduler such that only authenticated processes (i.e., those created from a trusted process) receive CPU slots. Additionally, and as a pre-requisite to the extension of the scheduler, iii) we authenticate the processes during their creation. The root of trust for authentication is derived from the verification of App signatures and the verification of the platform's integrity provided by the MTM. The proposed means ensure a preventive malware protection at a different granularity and performance level. Verifying the integrity of Apps via digital signatures will ensure that only trusted Apps which are not modified after their installation will be executed on the system. Authenticated processes will ensure that although malware is present it will never receive CPU slots and thus cannot behave maliciously. Our three steps approach requires the establishment of a trust anchor for App signatures and process authentication. Storage of a signing key in an immutable location of the MTM and its availability to only authorized entities will build up the trust anchor for the App signatures. The transitive trust and the sealed key storage offered by the MTM will build up the trust anchor for the process authentication.

3.1 Assumptions

The proposed malware famishment approach relies on the following four assumptions:

1. The programs `fork()`, `execve()`, and the Completely Fair Scheduler (`CFS`) from Linux are extended to be interfaced with the proposed security modules to perform the required verification steps described in Section 3.3. We call them `fork`$^{++}$(), `execve`$^{++}$(), and `CFS`$^{++}$ from now on.

2. The MTM provides the integrity of the system up to the first process `init()` running in the system. This includes particularly the verification of the security modules described in Section 3.3. This can be achieved by storing the hash values of those modules into the MTM and by verifying them during the booting.

3. All processes running on the system are derived from `init()` via `fork`$^{++}$() and `execve`$^{++}$() calls. The only way of assigning CPU cycles to a process is the scheduler `CFS`$^{++}$.

4. The MTM is bootstrapped by its owner administrating it. That is, the keys for encryption and authentication are generated and sealed[2] in the MTM (See Section 3.2). Moreover, the administrator of the smartphone (e.g., the company) let the MTM sign the Trusted App List for the Apps allowed to run on the smartphone (See Section 3.3).

[2]The MTM releases the content of a sealed storage only if the system is in a predefined status. The status can be checked by attesting the integrity of some system files. We note that this approach is used in the TPM based file encryption systems such as BitLocker Drive Encryption [BIT11].

3.2 Key Management

The proposed security architecture relies on the following keys and security primitives:

- *Encryption key*: the encryption key (K_{enc}) is a symmetric key used for encrypting and decrypting the Trusted App List. It is sealed in the MTM and released if the platform passes the integrity check during the booting.
- *Authentication key*: the authentication key (K_{mac}) is a symmetric key and used to compute the Message Authentication Code (MAC) of process IDs (`pid`). It is also sealed in the MTM and released if the platform passes the integrity check during the booting. K_{mac} is shared between the Process Authentication Module (PAM) and the Process Verification Module (PVM). PAM is used by `fork`$^{++}$() to authenticate the `pid` of processes before their creation with MACs. PVM is used by `execve`$^{++}$() and `CFS`$^{++}$() to verify the `pid` of processes before their execution.
- *MTM ownership key*: the MTM ownership key ($K_{mtmauth}$) is a shared secret between the MTM and its owner administrating it. The owner of the MTM stores this key outside of the MTM and uses it to let the MTM perform operations requiring authentication such as signing or generating sealed encryption and authentication keys.
- *MTM signature key*: the MTM signature key is a RSA public key pair. The private part of the key ($Priv_{sig}$) is known only to the MTM. This key is never revealed outside of the MTM and signature generations with it are performed within the MTM. Signing a data with this key requires authentication with the MTM using $K_{mtmauth}$. The public part of the signature key (Pub_{sig}) is stored on the platform. It is used to verify signatures generated by the MTM using $Priv_{sig}$.

3.3 Security Modules

The Android architecture with the additions required for realizing the proposed security mechanism is depicted in Figure 1[3]. The Android runtime is composed of the implementation of the core libraries e.g., for the Java programming language and the Dalvik Virtual Machine (DVM) which executes each App with a separate instance. The DVM uses the Linux kernel for low level functionality such as memory management and process scheduling [AND11]. The following security modules need to be implemented in the kernel space: App Verification module (AVM), Trusted App List (TAPL), Process Authentication Module (PAM), Process Verification Module (PVM), Trusted PID List (TPIDL), and the MTM. Please note that in this work we do not describe how these modules are

[3]We note that the proposed security concept relying on application signatures, process authentication and verification can be employed in any system based on Linux. However, due to interfaces required for communicating with the MTM and computing hash of application files, the implementation would vary depending on the concrete system architecture.

implemented. Instead we purely focus on the functionality that should be provided by them.

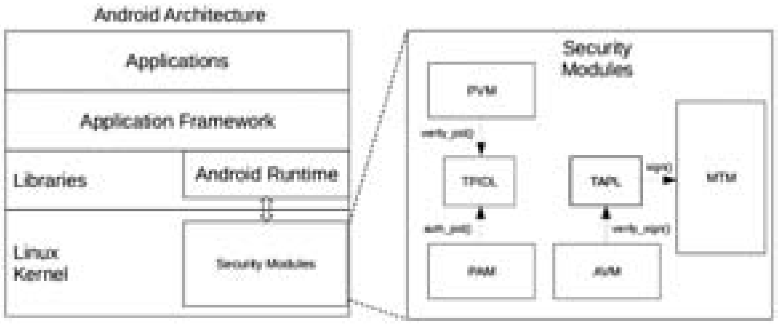

Figure 1: Android architecture with the proposed security extension.

MTM : The main task of the MTM module is to verify the integrity of the platform during the booting and to store encryption, authentication, and the private signature keys (K_{enc}, K_{mac}, and $Priv_{sig}$) in its protected locations. For checking the platform integrity, the MTM compares the hash of the important system modules such as AVM, PAM, and PVM with a snapshot of the last trusted state during the booting. The MTM releases the sealed keys K_{enc} and K_{mac} when the integrity of the verified modules and configuration files are intact.

Trusted App List (TAPL) : TAPL is the list of hash values of the trusted Apps encrypted with using K_{enc} and signed with using $Priv_{sig}$. The hash values are signed individually while the encryption is over the whole TAPL. K_{enc} is revealed to the AVM during the startup when the MTM detects no tampering with the platform's integrity. TAPL is decrypted at system's startup and encrypted again at system's shutdown. Hash values of Apps stored in the TAPL allow to detect any modification made on the Apps after their installation. Signing them allows the administrator to set the list of Apps which are allowed to run on the system. Hence, declaring an App as trusted requires the knowledge of $K_{mtmauth}$ which is needed to let the MTM sign the hash value of the new App. $K_{mtmauth}$ is known only to the administrator of the smartphone and the MTM.

App Verification Module (AVM) : The AVM verifies the trustworthiness of an App before its execution in the Dalvik Virtual Machine (DVM). This is done by checking whether the hash value of the App being run exists in the TAPL. The App is executed if the hash value is in the list, otherwise it is terminated. The DVM incorporates this module to execute only allowed Apps. More specifically, during loading the App, the hash of the App is computed and compared with the signed value in the TAPL. As the hash value of an App changes in case it is altered, any modification on it is detected and the execution of the App is terminated. The AVM decrypts the App Hash List every time it is loaded at system startup and encrypts it while the system is shutting down using K_{enc}. The reason of encryption is to block the system from running any App whenever a tampering with the system is detected. This is true as the App verification would fail always as long as TAPL is encrypted. The decryption key K_{enc} is revealed only if the system's integrity is verified by the MTM.

Trusted PID List (TPIDL) : TPIDL is the list of `pid`s authenticated with K_{mac} which is released upon the integrity verification of the platform by the MTM. TPIDL contains the `pid` and MAC pairs for each process running on the system which are computed by PAM as described below.

Process Authentication Module (PAM) : The PAM is used by `fork`$^{++}$() to authenticate `pid`s during a process cloning. We note that the authentication key K_{mac} is a sealed key in the MTM and released by the MTM only if the system modules are not altered. This is required to achieve protection against malwares e.g., faking the verification of a process authenticity. The `pid` of a new created child process is authenticated by computing the keyed hash value of the `pid` via HMAC using the key K_{mac}. The keyed hash values are stored in the TPIDL.

Process Verification Module (PVM) : The PVM is used by `execve`$^{++}$() and `CFS++` to verify the authenticity of `pid`s. More specifically, `execve`$^{++}$() checks if the `pid` of a process is authenticated and stored in the TPIDL before executing it. Similarly, `CFS++` assigns a process CPU slots if its keyed hash value is available in the TPIDL. Otherwise, the `CFS++` prevents the process from receiving CPU by setting the priority of that process to the minimum.

3.4 Establishing the Trust Anchor

The proposed security architecture relies on the MTM as a trust anchor. More specifically, the trusted Apps are signed with a private key which is known only to the MTM. The security of process authentication and verification used by `fork`$^{++}$(), `execve`$^{++}$(), and `CFS`$^{++}$ is based on the sealed key storage. The authentication key is released and available within the platform only if the platform's integrity is intact.

3.5 Building up the Chain-of-Trust

Every Android App is executed in an instance of the Dalvik virtual machine (DVM) with its own Linux process. The Apps are loaded via the `PathClassLoader` [NKZS10]. The DVM starts a process from the underlying Linux kernel to run the App when any component of the App is executed [AND11]. The first process started in Linux is `init()` whose trustworthiness is ensured by the MTM. The chain-of-trust is extended from `init()` to the App_t to be run. The trust is propagated through the processes via $\texttt{fork}^{++}()$ and $\texttt{execve}^{++}()$ calls which are used to start a new process in Linux. The system call $\texttt{fork}^{++}()$ initiates a child-process from a running process whenever a new process is started. This child process is a clone of the father with a different `pid`. The system call $\texttt{execve}^{++}()$ overwrites the functionality of this new child process (formerly created with $\texttt{fork}^{++}()$) by still holding the old `pid`. This is the moment at which the functionality of `App` is started. To maintain the chain-of-trust we propose the following: each time the `PathClassLoader` loads an App, the signature and the integrity of the App is verified with AVM via App's signed hash value stored in the TAPL. Moreover, each time the system call $\texttt{fork}^{++}()$ is executed, the resulting `pid` is authenticated with PAM and written in the TPIDL. Finally, the pids are verified with PVM when they are processed with $\texttt{execve}^{++}()$ and $\texttt{CFS}^{++}$.

3.5.1 Application Verification

The `PathClassLoader` is utilized to load an App for execution in the DVM. The AVM module can be hooked into the App loader such that the signed hash value of the App stored in the TAPL is checked with the AVM. The execution of the App is continued only if the signature is valid and the hash of the App[4] being run is equal to the signed hash value stored in the TAPL:

```
IF (verify(sig$_{hash_{App}}$, $Pub_{sig}$) == fail OR
TAPL -> hash$_{app}$ != hash(app)) THEN // performed by AVM
terminate ELSE
execute app by continuing with $fork^{++}()$ and $execve^{++}()$ calls
```

This verification with AVM ensures that the platform runs an App if it is classified as trusted in the TAPL and not altered after its installation.

3.5.2 Process Cloning

The system call $\texttt{fork}^{++}()$ is the first method called to create a child process while executing a task in Linux. Let $\texttt{pid}_p$ and $\texttt{pid}_c$ denote the process IDs of the parent process (i.e., the $\texttt{fork}^{++}()$ is called from) and the child process (i.e., the clone of the parent process). The execution is terminated, if the parent $\texttt{pid}_p$ is not authentic (i.e., its keyed hash value is not in the TPIDL). Otherwise a child process with a different pid is created and its keyed hash value is written in the TPIDL:

```
IF(TPIDL -> hmac$_{pid_p}$ != hmac($pid_p$, $K_{mac}$)) THEN // performed by PVM
terminate ELSE
$pid_c$ = $fork^{++}()$ AND TPIDL <- hmac($pid_c$, $K_{mac}$) // performed by PAM
```

[4]The hash value is computed over the `AndroidManifest.xml` file defining the permissions required by the App, and the `.dex` file containing the actual application code stored in the `/system/app` folder.

This operation ensures that a clone of a process is created iff the process being cloned (i.e., initiating the $fork^{++}()$ call) itself is trusted. We note that the authenticity check for the very first process `init()` is skipped, as its authenticity is ensured by the MTM.

3.5.3 Process Execution

The system call `execve`$^{++}$(`app`, ...) is called to fill the content of the child process with the functionality of the App desired to run. Let `pid`$_c$ denote the process ID of the child process (i.e., a clone of the parent process) on which `execve`$^{++}$(`app`, ...) is called. The execution is terminated, if the `pid`$_c$ is not authentic or the verification of this application fails. We note that the system call `execve`$^{++}$(`app`, ...) is invoked only if the App passes the verification step when it gets loaded via `PathClassLoader` as described in Section 3.5.1. If the application verification fails, neither `fork`$^{++}$() nor `execve`$^{++}$(`app`, ...) is invoked and the execution of the App is terminated[5]:

```
IF(TPIDL -> hmac$_{pid_c}$ != hmac($pid_c$, $K_{mac}$)) // performed by PVM
terminate ELSE
execute execve$^{++}$(app, ...)
```

This operation ensures that an App is always run by a trusted process and this is allowed iff the App being run is verified by AVM.

3.5.4 Process Scheduling

Whenever the scheduler `CFS`$^{++}$ is invoked to perform a process switch, it checks if the `pid` of the process, which is to run according to its priority, is authentic (i.e., its keyed hash value is stored in the TPIDL). The keyed hash value of the `pid` existing in the TPIDL ensures that the process belonging to that `pid` is trusted. Thus, the `CFS`$^{++}$ assigns the CPU to the process. Otherwise, the process is not trusted and the `CFS`$^{++}$ sets the priority and the counter of that process to zero to block it from execution:

```
IF(TPIDL -> hmac$_{pid}$ != hmac($pid$, $K_{mac}$)) THEN // performed by PVM
assign CPU to $pid$ ELSE
$pid_{priority}$ = 0 AND $pid_{conter}$ = 0
```

This operation ensures that non-authenticated processes receive no CPU from the scheduler.

3.5.5 Process Termination

A process termination requires deleting all hmac$_{pid}$ values belonging to that process from the TPIDL. Thus, the system functions of Linux for terminating processes such as `kill()` need to be extended with this functionality.

[5]The verification of the App before its execution can also be performed in `execve`$^{++}$() by comparing its hash value with the signed hash value stored in the TAPL similar to the Application Verification. The implementation would require to provide `execve`$^{++}$() with an interface that enables it to access the App files stored in the `/system/app` folder over which the hash value is computed.

4 Security Analysis

In order to evaluate the security of the proposed solution, we need to define an attacker model describing the capabilities of attackers. We focus in this work only those capabilities which would obviously break the security of the proposed approach and analyse how our security solution resists against them. The attacker can execute a malicious application on the proposed secure architecture if she can

1. modify the system functions `fork`$^{++}()$, `execve`$^{++}()$, and the scheduler CFS^{++} without being detected such that security evaluations are skipped;
2. cheat the established chain-of-trust `App`→`AVM`→`process`→`PAM`→`pid`→`PVM`→`execution`;
3. modify the signed hash values of trusted Apps stored in the TAPL;
4. let the MTM sign data with the private signature key $Priv_{sig}$;
5. use the system calls to modify PCBs in a way that she overwrites executable code of a trustworthy `pid` with malware (e.g. enforce an execve behavior without calling `execve`$^{++}()$);

We analyse now how the proposed architecture resists against these threats.

Threat 1: The integrity of the security modules and the extended system functions such as `fork`$^{++}()$, `execve`$^{++}()$, and the scheduler CFS^{++} are verified by the MTM during the booting. The MTM releases the decryption key used for decrypting TAPL and the authentication key used for process authentication only if the integrity of those modules is intact. Hence, the attacker needs to break the security of the MTM for sealed storage and integrity check. However, this is not feasible according to MTM's specification.

Threat 2: The hash values of Apps signed with $Priv_{sig}$ are stored in the TAPL. AVM compares the hash value of the App with the signed value from TAPL before continuing with its execution via `fork`$^{++}()$, `execve`$^{++}()$ calls. Hence, the attacker should forge the signature for the hash value of the malicious App that she wants to execute on the platform. However, as $Priv_{sig}$ is only known to the MTM, this is computationally infeasible if a secure signature mechanism such as RSA is employed. Moreover, all processes running on the system are derived from system function calls (`fork`$^{++}$ and `execve`$^{++}$) starting from the trusted process `init()`. That is, the rest of the trust-chain `process`→`PAM`→`pid`→`PVM`→`execution` during the creation of a process relies on the trustworthiness of the system functions and the process authentication and verification with PAM and PVM. The attacker cannot forge the trustworthiness of the system functions and the secrecy of the authentication key according the Threat 1. The attacker cannot break the trust-chain for a creation of a new process according the Threat 3.

Threat 3: The last chain of the trust during the creation of a process is `execve`$^{++}()$. The attacker can break the trust-chain if (i) she can forge the function `execve`$^{++}()$ without being detected or (ii) she can provide a malicious App (App') with $h(App') = h(App)$, where App is a trusted application stored in the TAPL or (iii) she can overwrite the hash value of a trusted App stored in the TAPL. According to Threat 1, (i) is not possible. (ii) is not feasible when a second pre-image resistant has function is used. Finally, (iii) is not feasible as the hash values stored in the TAPL signed with the private signing key $Priv_{sig}$ which is only known to the MTM.

Threat 4: Only the owner of the key $K_{mtmauth}$ is enabled to let the MTM sign a data with the private signature key. Since only the administrator owns this key it is not possible for malware to generate a signature for a malicious App.

Threat 5: To not allow an attacker with privileged access rights to maliciously overwrite the PCB respectively the whole `TaskStruct` containing links to the executable and state we also propose to hold a digest of the whole PCB, belonging to the above `pid` from an `execve`$^{++}$() call. However, the PCB is changing with each context switch, and, even worse, selected fields for a fair scheduling like e.g. `priority` and `quality` will change even at a higher frequency than at context switching frequency. For this reason we propose a compromise by which only selected fields of each PCB are hashed and stored into a TPCBL. Such PCB fields do not change over the lifetime of a PCB. Thus, at PCB generation in particular the fields `pid` and *programcode* (and not the instruction counter) are hashed an stored within the TPCBL. This list is created and used in similarity to the TPIDL within `fork`$^{++}$(), `execve`$^{++}$() and `CFS`$^{++}$.

5 Conclusions and Outlook

The work at hand is conceptual work on an MTM based watchdog for malware famishment in smartphones. Besides OS related processes only such processes are allowed to be created and started which stem from a digitally signed App with a signature key stored within the MTM. Moreover, by managing the process cloning, process execution and eventually even the process scheduling with the help of the trusted lists TAPL, TPIDL and TPCBL our approach even controls the allocation of CPU slots such that only authenticated processes receive CPU slots. This harsh security policy perfectly fits to smartphone use cases in which companies or other organizations want to equip their employees with smartphones running only dedicated software. We will soon start with a proof of concept implementation and a performance study of our malware famishment watchdog based on an MTM emulator.

6 Acknowledgments

The work presented in this paper was supported by the German BMB+F SKIMS project. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the SKIMS project, or the BMB+F.

Bibliography

[AND11] The Developer's Guide, Android 3.0 r1. developer.android.com, Web Page, `http://developer.android.com/guide/index.html`, March 2011.

[BC00] D.P. Bovet and M. Cesati. Understanding the Linux Kernel. O'Reilly Book, First Edition, 2000.

[BIT11] BitLocker Drive Encryption. Microsoft, `http://msdn.microsoft.com/en-us/windows/hardware/gg487306`, 2011.

[EK11] J-E. Ekberg and A. Kylänpää. MTM implementation on the TPM emulator. Available at `http://mtm.nrsec.com/index.html`, 2011.

[JSS06] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *the 11th ACM symposium on Access control models and technologies (SACMAT'06)*, pages 19–28, Lake Tahoe, California, USA, June 2006.

[LWPM07] P.A. Loscocco, P.W. Wilson, J.A. Pendergrass, and C.D. McDonell. Linux kernel integrity measurement using contextual inspection. In *the 14th ACM workshop on Scalable trusted computing (STC'07)*, pages 21–29, Alexandria, Virginia, USA, November 2007.

[MTM10] TCG Mobile Trusted Module Specification. Specification Version 1.0, Revision 7.02, TCG Mobile Phone Work Group, April 2010.

[NAZA09] M. Nauman, M. Alam, X. Zhang, and T. Ali. Remote Attestation of Attribute Updates and Information Flows in a UCON System. In *the 2nd International Conference on Trusted Computing (Trust'09)*, pages 63–80, Oxford, UK, April 2009.

[NKZS10] M. Nauman, S. Khan, X. Zhang, and J-P. Seifert. Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform. In *the 3rd International Conference on Trusted Computing (TRUST'10)*, Berlin, Germany, June 2010.

[SEL09] Security-Enhanced Linux. National Security Agency (NSA), `http://www.nsa.gov/research/selinux`, 2009.

[SSM11] M. Strasser, H. Stamer, and J. Molina. Software-based TPM Emulator. Available at `http://tpm-emulator.berlios.de`, 2011.

[SZJvD04] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *the 13th conference on USENIX Security Symposium (SSYM'04)*, pages 16–16, San Diego, CA, USA, August 2004.