A Specification Based Intrusion Detection Framework For Mobile Phones

Ashwin Chaugule, Zhi Xu, and Sencun Zhu

Department of Computer Science and Engineering,
The Pennsylvania State University, University Park, PA 16802
{avc114,zux103,szhu}@cse.psu.edu,

Abstract. With the fast growth of mobile market, we are now seeing more and more malware on mobile phones. One common pattern of many commonly found malware on mobile phones is that: the malware always attempts to access sensitive system services on the mobile phone in an unobtrusive and stealthy fashion. For example, the malware may send messages automatically or stealthily interface with the audio peripherals on the device without the user's awareness and authorization. To detect the unauthorized malicious behavior, we present *SBIDF*, a *Specification Based Intrusion Detection Framework*, which utilizes the keypad or touchscreen interrupts to differentiate between malware and human activity. Specifically, in the proposed framework, we use an application independent specification, written in *Temporal Logic of Causal Knowledge (TLCK)*, to describe the normal behavior pattern, and enforce this specification to all third party applications on the mobile phone during runtime by monitoring the inter-component communication pattern among critical components. Our evaluation of simulated behavior of real world malware shows that we are able to detect all forms of malware that attempts to access sensitive services without possessing user's permission. Furthermore, the SBIDF incurs a negligible overhead (20 μ secs) which makes it very feasible for real world deployment.
Keywords: Mobile Phone, Intrusion Detection, Messaging Attack, Audio Attack.

1 Introduction

With the fast growth of mobile market, surveys and research show that there is an increasing number of mobile phone malware. There are over 400 mobile phone viruses detected so far [12]. Over 17% of manufacturers reported more than 1 million attacks on mobile phones in 2008 [2]. Infected phones are even capable of bringing down the GSM infrastructure of a whole city by exploiting the SMS/MMS messaging protocols [7]. With the advent of newer more capable mobile phone platforms, this security risk will only increase.

Most commonly found malware [11, 23, 25, 29] on mobile phones share a common pattern, in which the malware always attempts to access sensitive services stealthily without authorization from the mobile phone user. For example,

– In the *Messaging Attack*, such as Cabir [8], Commwarrior [9] and Viver [10], the attacker interfaces directly with the exported serial port of the GSM engine and implement his own messaging framework, thus bypassing all the phone stack components. This can allow him to send messages and interfere with GSM calls. The attacker's intention may be to deplete the battery charge on the phone, spread the malware to other devices, or affect the monthly bill of the user[23, 13, 25].
– In the *Audio/Video Attack*, such as the research presented by Xu et al. [29], the attacker may access the audio/video peripherals and use them to covertly record an ongoing conversation, interfere in a conversation by playing an audio file or even record an ambient conversation.

To detect those stealthy unauthorized service accesses, the main challenge is *how to differentiate between a purely software generated action and a user initiated action*. Software generated actions without user awareness or interactions will have a high probability of being malicious. Another challenge is *how to detect the unauthorized access efficiently*. Due to the limited battery and computing resources, security solutions for desktops, such as the machine learning based misuse detection approach in [24] and mandatory access control based solutions [20], are not suitable for mobile phones.

In this paper, we propose *Specification Based Intrusion Detection Framework (SBIDF)*, a framework designed specifically for detecting the unauthorized access to sensitive services on mobile phones, specifically for SMS service and audio service. In the proposed framework, we solve the differentiation challenge by observing hardware interrupts. Mobile phones come with touchscreens and/or keypads which generate hardware interrupts for each key press event. This asynchronous notification mechanism is the key to differentiate between a user generated activity and a purely software generated one, since the latter cannot explicitly generate a hardware interrupt. To solve the efficiency challenge, the proposed SBIDF only monitors the Inter-Process Communication (IPC) for the critical components of the userspace stack, which include a *finite* and *small* set of applications *always* involved for performing any function of the device.

Briefly, SBIDF consists of two phrases: training phrase and enforcing phrase. In the training phrase, the vendor defines specifications, written in *Temporal Logic of Causal Knowledge(TLCK)* language, before shipping to the user. These specifications are independent to applications, thus no change on specifications is needed when new applications are installed on mobile phone. In the enforcing phrase, these specifications are then converted into a precise sequence of inter-component communication events in the system. During the usage of mobile phone, the SBIDF enforces the predefined specifications at runtime. Whenever a deviation from specifications is detected, the SBIDF will alert user immediately. The main contributions of this paper are summarized as follows.

– We define concise and precise specifications for legitimate behavior in the system for the events of messaging and calling. The specifications describe system activities related to calling and messaging originating from a hardware interrupt of a corresponding key press event.

– Our framework, SBIDF, resides in the kernel and enforces the behavior of system according to the specification and detects any malicious activity that tries to subvert SBIDF.
– We simulate behavior of real world malware and successfully prevent its malicious attempts to send multiple SMS's and control the audio hardware which interferes with an on going call or stealthily records ambient conversations.
– SBIDF is able to detect all malware with negligible runtime overhead on the system (20μ secs).

Here is the roadmap of this paper. Section 2 describes related research work, Section 3 describes a typical phone stack, Section 4 outlines the security analysis of design, Section 5 describes a design overview, Section 6 describes the framework in detail and Section 7 shows the promising feasibility of our work. We end with laying out future work in Section 8 and conclusions in Section 9.

2 Related Work

Most of the previous research work focused on optimizing desktop solutions for embedded devices. Very few of these propositions started the design from the ground up with an exclusive focus on mobile devices.

Enck et al. [22] and Ontang et al. [21] look at individual application requirements on the Android operating system and define mechanisms to enforce policies defined by the application provider. This approach is very application centric and thus implies numerous policies as number of applications increases. On the other hand, our approach is application agnostic and enforces policies defined only for critical components that provide services to other applications. Thus the number of applications doesn't affect the complexity of the framework.

Xie et al. [15] propose a behavior-based malware detection system named pBMDS, which uses a probabilistic approach to detect anomalous activities in phones via monitoring system calls. In this paper, our detection relies on the inter-components communications within the phone, including IPC events, system calls, and hardware interrupts. Moreover, the specification in SBIDF is predefined thus prevent potential false positive detections caused by false learning.

Bose et al. [3] propose a solution to logically order the events caused by applications on the device. They use machine learning theory to detect the pattern of these events and compare them to a whitelist of behavior signatures. However, their scheme requires a vulnerable complex framework in userspace to detect and monitor these learning patterns and they depend on remote analysis of behavior to reduce the overhead of computation on the device. Their framework is prone to mimicry attacks, so they can have false negatives with their approach.

Cheng et al. [4] proposes a collaborative detection and alert system where neighboring phones collect and analyze system data for intrusions. These designs may also need a proxy server in case collaborative analyses is not possible. They require additional user space components that constantly run in the background collecting this data, which are an expensive overhead for these devices.

Vigna et al. [18] show a way of labeling processes and data to prevent cross service attacks. By tagging resources used by processes upon network activity

they monitor the flow of data in between processes. Their scheme is effective in achieving their goal, but not without overheads. Labeling resources requires several rules, transition of labeling incurs monitoring overheads and false positives can easily occur when legitimate processes initiate network activity.

Vogel et al. [28] ported SELinux on the mobile phone to prevent the SMS/MMS related attacks. On the same lines Divya et al. [19] use a stripped down SELinux policy infrastructure based on the PRIMA model to define policies for applications running on the mobile phone to prove to remote verifiers that the system is safe to run their third party software. Desmet et al. [6] describe a way to securely run third party applications on mobile phones without the conventional sandboxing techniques. Their design uses secure execution techniques like run-time monitoring, static verification and proof carrying code. The run time monitors insert hooks into the applications and enforce correctness according to the policies and rely on defining complex policies manually, where improper settings could easily compromise the system. Also, addition of SELinux on mobile phones shows significant overheads in the kernel as shown by Nakamura [20].

Venugopal [26] came up with a faster way to lookup signatures using hashes. He focused on the overhead of detecting which signature matches the current system behavior. However, these anomaly detection techniques still have the downside of false positives and cannot detect zero day attacks. Venugopal et al. [27] describe a virus detection system for the Symbian platform which monitors the DLL functions used by applications. Using Bayesian decision theory and past virus samples, they observe malicious activity. Although they claim a 0% false negative rate, they are only able to detect 95% of the viruses.

3 Background

3.1 Cellphone Platform

For this research we used the Qtopia userspace stack (Qt-Extended-4.4.3) on the Openmoko Neo1973 handset which contains an ARMv7 based CPU (Samsungs S3c2410). This stack is widely used in many of Nokia phones which use the Linux kernel. This stack's design represents many other stack implementations very well. Although there will be differences, we believe our framework can be extended to other variants easily. We describe the implementation of SBIDF on Android platform in the Discussion Section.

Figure 1 shows the components of the Qtopia phone stack and the key interactions. Qtopia contains a critical component called QPE. This is the main server that interfaces with the operating system through device nodes and sockets for IPC communication. QPE is the first application to start when the stack executes. It opens all the necessary sockets and device nodes and then initiates other critical components like Message Server and Media Server. The Message Server controls the messaging and emailing functions. The Media Server controls the voice and audio related functions. However, QPE makes the final system calls to the kernel for submitting the SMS/MMS packet and the calls to alter the microphone state. QPE, Message Server and Media Server are started up when the

device boots and remain resident till reboot or shutdown. There are several other applications like Qtmail, Games, Browser etc. that are invoked as plugins. These plugins are separate binaries that link with the QPE server at runtime and are executed only when needed. They terminate after their function is over or when the user closes them explicitly. The plugins communicate with each other via QPE by using IPC mechanisms.

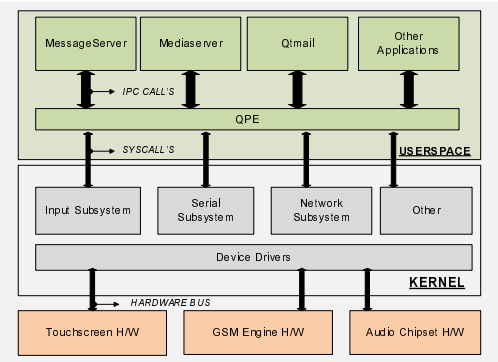

Fig. 1. Qtopia Stack Design

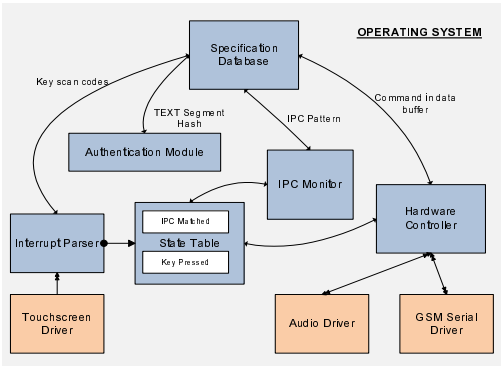

Fig. 2. Specification Based Intrusion Detection Framework (SBIDF)

In the case of Qtopia, the communication channels are implemented as Sockets and Pipes. These are created by QPE during startup and then the connection at the other end is completed by the other components when they are executed. Different components communicating over the same named socket can still be identified uniquely in the kernel. *The key concept in SBIDF is that we only need to monitor activity between critical components and as such define specifications only for their interactions.* So, the number of specifications does not increase as the number of applications downloaded to the phone increases.

3.2 Design Motivation

Our framework is based on a key observation which is: a user generated activity usually includes a hardware interrupt generated via touchscreens and/or keypads; however, a purely software generated activity cannot explicitly generate a hardware interrupt. This is true because the interrupt generated by the touchscreen hardware is received directly by the CPU. The CPU then responds by calling the interrupt handler of the touchscreen device driver. As the interrupt handler is part of operating system, userspace code cannot directly call it unless the operating system has been tampered with. e.g. via malicious system calls. In our trust model, we assume that the operating system is within the TCB. Therefore, the hardware interrupt handler can not be called directly by userspace code.

In mobile devices, the touchscreen interrupt out line is connected to the CPU interrupt [17] in line via an interrupt controller (which is part of the CPU).

Pressing the touchscreen, triggers an interrupt to the CPU directly from the touchscreen hardware. Each peripheral capable of raising hardware interrupts is assigned a unique IRQ number by the CPU designers. The operating system, detects which peripheral triggered the interrupt via a unique interrupt number. Then the operating system calls the interrupt handler of the touchscreen device driver which is part of the operating system code. Clearly, the legitimate path of flow for the interrupt is all through hardware and then the operating system. There is no userspace application interaction involved in the path.

Note that, userspace code can also raise exceptions like data aborts, software interrupts (system calls), floating point exceptions. However, these have different handlers in the operating system because they are not "hardware interrupts". As explained in the ARM Architecture Reference Manual [17], the CPU jumps to different addresses for these type of exceptions and thus calls different handlers. So unless the operating system is tampered with, there is no direct way in which the touchscreen interrupt handler will be called from userspace code.

4 Security Model

4.1 Threat Model

We assume that attackers use malware to exploit vulnerable components in userspace. Malware is considered as third party applications. Attackers may even compromise the integrity of existing components in the platform, such as the Message Server and Qtmail. For example, in the messaging attacks, we assume that the attacker can interface directly with the exported serial port of the GSM engine and implement his own messaging framework, thus bypassing all the phone stack components. In audio attacks, peripherals are exported through device nodes and can be configured through system calls (IOCTL's etc). The telephony stack on the device registers with these nodes and provides the respective service to other applications. But an attacker can easily open them by himself and use them to alter the audio configurations.

There are other threats where the attacker can interface directly to the network interfaces like BT or WiFi or even hijack the browser or mailing applications to spread malware. If the mobile phone's operating system is not protected (e.g., operating system memory maps are exposed to userspace), then he may even install rootkits that maliciously alter the kernel control flow. This later category is only commonly found on desktops where the user typically has to install device drivers and as such is not a major threat to mobile devices.

In this research, we address the messaging and audio attacks due to their significant implications and widespread occurrence. SMS is an ubiquitous and reliable method to communication among mobile phones. It exists on almost all mobile phones with more than 2.4 billion active users. Also, the mature telecommunication infrastructure makes the message delivery very reliable even when the receiver is offline at the moment of sending. Thus, it is one of favorite choices for spreading malware or building command-and-control channels. Audio attacks

are chosen because of its serious consequences. Leaking the audio information directly invades the user's privacy. Video attacks can be detected in a similar way as that of audio attack. We discuss other types in future work section.

4.2 Trust Model

In our work, the operating system running on the device mainly forms the Trusted Computing Base (TCB). The bootloader of phone should correctly load the kernel which contains the SBIDF implementation. Kernel rootkits which may compromise the kernel are not the focus of this research.

We assume that the kernel memory interfaces are not exported to userspace. This requirement is a necessary to prevent userspace applications from writing into kernel memory and altering kernel control flow[14]. For example, in Linux these are exported as /dev/mem and /dev/kmem. These interfaces can be easily disabled during kernel configuration time.

In the userspace, the integrity of critical userspace components should be guaranteed, such as *Message Server* and *Qtmail*. This requirement is needed to prevent a malicious application from bypassing the monitoring of SBIDF. To provide such guarantee, we authenticate the critical components by setting up a TEXT segment hash of the critical userspace component. We only need to trust the critical components during the training phase. After the phone is deployed, the attacker could tamper with them, but as we explain later, our SBIDF can easily prevent any damage.

Note that, those critical components are bundled with the kernel and shipped by the phone vendor. The user and third party applications lack permissions to modify those components. Therefore, their authentication only needs to be done once when the phone is shipped. If there is a need of update , the vendor will redo the authentication during the update.

5 Design Overview

SBIDF utilizes the keypad or touchscreen interrupts to differentiate between malware and human activity. Here we assume that specifications defining correct pattern have been defined and stored in SBIDF. We will explain how to create these specifications in details in Section 6.

Figure 2 shows the overview of our framework. Whenever these critical components start up for execution, the *Authentication Module* computes an md5 hash over the TEXT segment of the component. The TEXT segment is the only read-only segment of a process and any modifications to it during runtime will be detected by the operating system. However, the attacker may replace or infect the binary file of the critical component when it is stored on the flash device. In order to detect this, the *Authentication Module* compares the hash with a pre-computed copy of the hash of that component. This pre-computed copy is present in the *Specification Database*. It could be calculated for the first time by the phone stack provider during a training phase and then statically stored in

the *Specification Database* for future use. When the *Authentication Module* finds a match of the hash with the pre-computed copy, it sets an authentication bit in that component's Process Control Block (PCB), which is an in-kernel representation of the userspace application, to signify the successful authentication. This avoids recalculation of the hash for later stages.

In order to detect the messaging or calling activity in userspace from within the kernel, the *IPC Monitor* observes all read and write IPC calls and it checks the *State Table* for the *Key Pressed* flag. If the flag is unset, then it simply returns, since the current IPC transaction was not triggered by a hardware interrupt. If it is set, the *IPC Monitor* checks the PCB of the process that has made this IPC call. If it finds the authentication bit to be set, then it knows that the *Authentication Module* has authenticated this process previously. Similarly, *IPC Monitor* also checks if the communicating peer involved in the current IPC has been authenticated. Following this, it checks if the processes are communicating over a specific named socket. This socket information is present in the specifications in the *Specification Database*. Now the *IPC Monitor* knows which authenticated critical components are communicating over a specific socket and that this activity was triggered by a hardware interrupt. For each type of the attacks (i.e., messaging or audio attacks) we aim to prevent, the *Specification Database* contains unique specifications which define the expected IPC pattern between the critical components. Amongst all the critical components, there is usually a single component that finally sends out an SMS/MMS message from userspace to the kernel. If all the IPC transactions occur as specified, then the *IPC Monitor* sets a permission bit of that critical component in its PCB, granting it the permission to submit the SMS/MMS, and it also sets the flag *IPC Matched* in the *State Table*.

The *Hardware Controller* mediates all the read and write calls that occur on the GSM serial port and audio interface. When the *Hardware Controller* detects that there is a GSM command (e.g., *Submit SMS*) in the write call, it checks the *IPC Matched* flag in the *State Table*. If this flag is set, then the *Hardware Controller* checks if a permission bit is set for the critical component invoking the write. If both these conditions are true, it allows the message to go through the GSM hardware. If either of them is false, the message is denied.

SBIDF detects audio attacks in a similar way. The *Interrupt Parser* checks if the ACCEPT CALL or END CALL keys are pressed. The *IPC Monitor* checks if the expected IPC pattern occurs. The *Hardware Controller* checks if an expected authenticated critical component as defined by the specification has been granted permission by the *IPC Monitor* in the write call to the audio driver to switch the microphone ON . If the *IPC Matched* flag is set, and the critical component has been granted the permission, then the MIC is turned ON. The *Hardware Controller* parses the data buffer of the write call to the GSM serial driver so as to determine when to switch the MIC OFF. If it finds a *CALL HANGUP* GSM command, then it knows the call is over and then switches the MIC OFF.

Compare the above scenarios with what the malware would do. There will be no hardware interrupt generated. So, if the malware directly tries to access the

hardware, the *Hardware Controller* will deny access, because it will not see the permission bit set by the *IPC Monitor*. If the malware tries to masquerade as one of the critical components, it will be detected by the *Authentication Module*, which mediates all application's startups. If the malware tries to mimic the IPC pattern, the *IPC Monitor* will detect this, because it will not get the interrupt information from the *Interrupt Parser*, or it will not get the authentication information from the *Authentication Module* and therefore will not set the permission bit. Therefore, the *Hardware Controller* will always deny any malicious access.

6 Specification Design and Enforcement

In SBIDF, the specifications define the precise expected behavior of the system for specific events like messaging and calling. Each specification contains the $TEXT$ segment hash of the critical components involved in the activity, the key scan codes that trigger the IPC activity, the expected IPC pattern and the action for the *Hardware Controller*. The *Specification Database* is available to all the above components of the *SBIDF*. Each specification is third party application independent.

6.1 Specification Formalization

We use the TLCK (Temporal Logic of Causal Knowledge) described by Bose et al. [3] to describe the sequence of events in the system and the interactions in the SBIDF's state machine. Temporal events in the system are described by following notations.
$\bigodot_t$ is an event true at time t.
$\triangle_t$ is an event true before time t.
$\Box_t^{t-k}$ is an event true in the interval $[t-k, t]$.
The other operators such as $\wedge$ and $\vee$ etc. carry their usual meanings. Next we define some propositional variables.

- **KeyPressed(S)**: Where, $S = \{s_1, s_2, s_3, .., s_n\}$ which is a set of *'n'* scan code interrupts which we need to monitor. e.g.*SEND, RECORD, STOP* etc. $KeyPressed(S)$returns $TRUE$ if any of the monitored keys is pressed.
- **AuthApp(T)**: Where, $T = \{t_1, t_2, ..., t_m\}$ which is a set of *'m'* applications that we need to authenticate. e.g. *QPE, Qtmail, Mediaserver, Messageserver* etc. $AuthApp(T)$ returns $TRUE$ if the hash of the running application matches with a pre stored digest of that application.
- **IPC(T, Sockname)**: IPC encapsulates both IPC read and write functions over the AF_UNIX socket. The socket name is defined by *Sockname*. The IPC variable returns $TRUE$ iff all the IPC transactions occur over the specified socket in the specified sequence and within a timeframe. The pattern for the expected IPC is defined by a truth table as shown in the next section.
- **ParseATCMD(C)**: Where, $C = \{AT + CMGS, AT + CHLD = 1\}$ is the set of AT commands to be monitored. $ParseATCMD(C)$ returns $TRUE$ when the *Hardware Controller* finds any of these commands in the *data* buffer that is passed to the GSM engine over the serial port.

– **Perm(X)**: $X = \{QPE\}$. $Perm(X)$ returns $TRUE$ if QPE has its permission bit set as explained previously. For other stacks, X will contain the critical components which make the final system calls to access the hardware.

Now, we define specifications using the TLCK and the aforementioned notations for messaging attack and audio attack.

For Messaging Attack Let
$A \xrightarrow{r} B$ denote an IPC Read of application A from application B.
$A \xrightarrow{w} B$ denote an IPC Write of A to B.
As a first example let us consider the specification for submitting an SMS.
The set $S = \{SEND\}$
The set $T = \{Qpe, Qtmail, Messageserver\}$
The set $C = \{AT + CMGS\}$
$Sockname$ = *"/tmp/qtembedded-0/QtEmbedded-0"*

The truth table for $IPC(T, Sockname)$ is shown in Table 1. The IPC variable is $TRUE$ iff all the other variables are $TRUE$.

Variable	Value
$Qpe \xrightarrow{r} Qtmail$	T
$Qtmail \xrightarrow{r} Qpe$	T
$Qpe \xrightarrow{w} Qtmail$	T
$Qtmail \xrightarrow{w} Qpe$	T
$Qpe \xrightarrow{r} Msgserver$	T
$MsgServer \xrightarrow{r} Qpe$	T
$Qpe \xrightarrow{w} MsgServer$	T
$Msgserver \xrightarrow{w} Qpe$	T
IPC(T, Sockname)	**T**

Table 1. Messaging Truth Table

Variable	Value
$Qpe \xrightarrow{w} Mediaserver$	T
$Mediaserver \xrightarrow{r} Qpe$	T
$Mediaserver \xrightarrow{w} Qpe$	T
$Qpe \xrightarrow{r} Mediaserver$	T
IPC(T, Sockname)	**T**

Table 2. Audio Call Truth Table

The specification is defined as follows:

$\bigodot_t(SubmitSMS) = \triangle_t(KeyPressed(S) \wedge AuthApp(T)) \wedge (\Box_t^{t-k}(IPC(T, Sockname) \wedge Perm(X) \wedge ParseATCMD(C)))$

Here, $SubmitSMS$ is true only when a real user pressed the $SEND$ key on the touchscreen/keypad, the applications in set T were authenticated by the *Authentication Module* and there was an expected IPC transaction over the socket defined by $Sockname$ between these authenticated components within a time frame, the *IPC Monitor* set the permission bit for QPE and the *Hardware Controller* received an $AT + CMGS$ command to submit the SMS from QPE in the data buffer of the GSM serial port. The time frame for IPC can be customized depending upon the overheads of the IPC calls. When $SubmitSMS$ evaluates to $TRUE$, then the outgoing SMS is allowed, else denied.

For Audio Attack In a similar way, the audio attack specification can be constructed as follows.

The set $S = \{CALL, ENDCALL\}$
The set $T = \{Qpe, Mediaserver\}$
The set $C = \{AT + CHLD = 1\}$
Sockname="/tmp/qt-embedded/valuespace_applayer"

The truth table for IPC(T, Sockname) is shown in Table 2. The IPC variable is $TRUE$ iff all the other variables are $TRUE$. The specification is defined as :

$\bigodot_t(AllowAudio) = \triangle_t(KeyPressed(S) \wedge AuthApp(T)) \wedge (\Box_t^{t-k}(IPC(T, Sockname) \wedge Perm(X)))$

Here, *AllowAudio* is true when a real user presses the CALL key, the *IPC Monitor* confirms the IPC pattern amongst authenticated processes on the specified socket within the specified time frame and the authenticated QPE is given the permission to toggle the microphone, then the *Hardware Controller* turns the microphone ON. At all other times, any command to alter the microphone state is denied. Since the *Hardware Controller* controls the microphone, it needs to know when to turn it OFF again. For this we have another specification.

$\bigodot_t(DenyAudio) = \triangle_t(KeyPressed(S) \wedge AuthApp(T)) \wedge (\Box_t^{t-k}(IPC(T, Sockname) \wedge ParseATCMD(C) \wedge Perm(X)))$

Here the *Interrupt Parser* looks for the $ENDCALL$ scan code. The truth table for this IPC operation in case of the Qtopia stack is the same as the case for $CALL$. In order to ensure that a call is being dropped or ended, the *Hardware Controller* parses the AT commands passed to the GSM engine. So, when it detects the command $AT + CHLD = 1$ after the IPC operations, and QPE has the permission, it switches the microphone OFF.

6.2 Specification Enforcement

The description of specification enforcement in the *SBIDF* is described with a State Machine. In Figure 3, we show the state machine for specifications we introduced in previous subsection.

State INT: This is the Start state of the machine, where the *Interrupt Parser* is just parsing the Input key events. The set of key scan codes to be monitored is dependent upon the specification. When either of these key presses are detected, the *SBIDF* transitions to the IPC *State*.

State IPC: This is where the *IPC Monitor* begins to monitor the ensuing IPC communication between processes after a specific key press event. In this state, it also checks if the communication is being performed over a specified named socket as defined by the specification and the communicating peer processes are authenticated. If authentic, it proceeds to either the $T1$ or $T2$ *States*.

State T1: Here, the *IPC Monitor* detects there was an IPC Send operation from Qpe to Qtmail ($Qpe \xrightarrow{w} Qtmail$). This is an indication that there could be a event to form an outgoing SMS. Then the SMS timer is started. After this timer expires, the *IPC Monitor* checks how many expected IPC operations were performed. Here, the timer is set to expire in 1 second. The timer expiry value is decided by observing the overheads involved in a normal IPC call. Upon timer expiry, if all the expected IPC occurred and the *Hardware Controller* gets an $AT + CMGS$ command in the *data* buffer of the GSM serial driver from an

authenticated QPE with the permission granted, then the GSM transmit path is unlocked and the SMS is let though the GSM engine in $STATE\ MSG$.
State T2: Here the *IPC Monitor* detects if there was an IPC Send operation from QPE to Mediaserver ($Qpe \xrightarrow{w} Mediaserver$). This signifies that an outgoing GSM call could be in progress. The MIC timer is triggered to expire in 1 second. After this timer expires, the *IPC Monitor* checks if the expected IPC pattern occurs, then the *Hardware Controller* checks if the $AT + CHLD = 1$ command is found in the *data* buffer of the GSM serial driver written by an authenticated QPE with the permission granted. If this command is found, then it signifies an end of call, so the *Hardware Controller* switches the microphone OFF, else it is turned ON. This way, the microphone is kept ON only during the duration of a call and when the call is in progress, any modification to reroute the audio is denied by the *Hardware Controller*.

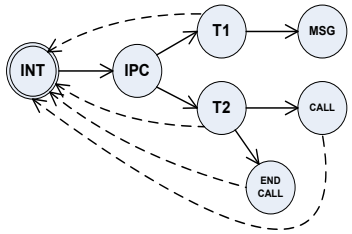

Fig. 3. State Machine

Reverse Edges: The dashed back arrows in the State Machine in Figure 3 are explained here. These edges characterize the false conditions of the propositional variables described previously.
INT → INT: This means the *Interrupt Parser* has not encountered any key press events for the ones it is monitoring.
IPC → INT: This could happen when

- The *Interrupt Parser* falsely recognized a key press event but the *IPC Monitor* did not find a matching IPC pattern.
- Unauthenticated entities tried to communicate over the specified socket. This could be when malware is masquerading as legitimate applications, or when malware is trying to mimic the IPC pattern.
- Authenticated entities sent data over some other socket than the one specified. This could happen when the ongoing pattern represents other system activity that is legitimate but has no relation to the attack prevention.
- When the *IPC Monitor* detected that the ensuing IPC activity was not triggered by an expected hardware interrupt.

T1 → INT: The *IPC Monitor* started the SMS timer, but the ensuing IPC transactions didn't match with the specification or didn't complete in time due to system noise.
T2 → INT: The *IPC Monitor* started the Call timer, but the ensuing IPC transactions didn't match with specification or didn't complete due to system

noise.
(MSG/CALL/ENDCALL) → INT: In any of these cases, the *Hardware Controller* did not get the commands in the data buffer from an authenticated QPE component. Or, the *IPC Matched* flag was not set, which implies QPE did not get permission. The latter case would arise when some userspace application tried to directly interface with hardware. This could also happen when the system noise mimicked the events in the specification but it was a false alarm.

Since we are only concerned with the interactions amongst the select few critical components in the phone stack, the complexity of the state machine does not increase with the number of applications running in the system. The number of nodes and state transitions will only increase if we aim to prevent more attacks than the ones we have described.

7 Evaluation

All instrumentation was performed completely in the kernel of the Openmoko device. The hardware of the device consists of a Samsung S3c2410 ARMv7 based CPU running at 266MHz, with 128MB SDRAM, 64MB NAND Flash. The Linux kernel version used was v2.6.24 with modifications to add the SBIDF code. Only 784 lines of kernel code were added and no userspace code was changed.

7.1 Security Evaluation

For the simulation of malware, we coded two representative applications to demonstrate the Messaging and Audio attacks that replicate the behavior of commonly found malware. The SMS attack program implemented its own message server and interfaced directly with the GSM serial port. Without the SBIDF, this program automatically sends SMSes. The SBIDF, denied all SMSes from this program, since the events in the system did not match with the specification therefore the *Hardware Controller* denied access to the hardware.

Note that although the attack we implemented is a specific one, to our knowledge all the existing messaging attacks are similar. Indeed, our framework is able to prevent any variant of the Messaging attacks since all these attacks finally try to use the GSM hardware either directly or indirectly. For example, such malware may exploit another application to interface with the GSM hardware on its behalf. Since the SBIDF uses precise specifications that define the behavior amongst critical applications to send SMSes, any malware that deviates from this behavior will be detected. Also, even if the malware attempts to mimic the specification behavior, the *Authentication Module* detects malware that masquerades as legitimate applications, the *IPC Monitor* detects any malicious activity since it checks the authenticity of the communicating peers, the *Hardware Controller* denies any access to hardware if the *IPC Matched* flag is not set and finally since there will be no hardware interrupt to trigger the IPC, the *Interrupt Parser* will not set the *Key Pressed* flag.

The audio attack, firstly implemented by us, tries to interface with the audio subsystem node to configure the audio chipset. This malware tries two attacks

on the audio interface. The first one tries to play a file during a call, with an intention to interfere with the on-going conversation by making the other peer hear the sound from the file. The second one tries to record an ongoing conversation with and without a call in progress. In the second case, the intention of the malware is to covertly record the conversations during a call and to record ambient conversations when the user is not on a call. The recorded audio is routed to a file, which can then be transferred to the attacker via an SMS or any other network interface. This part of transferring the recorded audio was not implemented. However, with the SBIDF, any attempt to alter the microphone configuration was denied by the SBIDF. Using the similar principles behind preventing the SMS/MMS attack, our framework is able to prevent any variation of the audio malware. Moreover, the video attack as described in [29] can also be easily prevented by the SBIDF, because the audio attack program closely matches with their video capture malware.

7.2 Overhead Evaluation

Application Text Segment Sizes The *Authentication Module* only calculates the hashes of the $TEXT$ segment of the critical applications. Their sizes as stored in the respective PCBs are as follows: QPE(176 KB), MediaServer(192KB), MessageServer(596KB), Qtmail(28KB).

Application Load Time The SBIDF affects the load time of only the critical components. All the other applications being loaded in the system are not considered by the SBIDF. Since *do_execv* is the system call to load the application into main memory for the Linux kernel, the table shown below shows the overheads for this call with and without the SBIDF.

Application	**Time**	
	W/ SBIDF(ms)	**W/O SBIDF**(ms)
QPE	374	2
MediaServer	210	78
MessageServer	561	66
Qtmail	40	10

Table 3. Application Load Overheads

The time taken for these applications to load through the *Authentication Module* interface may seem very high. But this is because it scans through the whole $TEXT$ segment of the application, calculates an md5 hash and then compares this hash with a pre-stored digest that was computed in the training run. This scan may include a page table walk to fetch the respective pages into memory. However, this is only a one-time overhead, since the *Authentication Module* sets a bit in that applications PCB after comparing the hashes to signify whether the application has been authenticated. This bit is then checked during the IPC transactions thus avoiding re-calcuation of the hashes.

Training Run Time For the training run, the SBIDF uses the same logic to calculate the hash of the running applications, but stores the hashes into a file,

which may then be statically added to the *Specification Database* for run-time usages. Table 4 includes the implicit file I/O overheads.

Application	**Time** (ms)
QPE	375
MediaServer	185
MessageServer	816
Qtmail	43

Table 4. Training Run Overheads

	With SBIDF(μs)	
	Send	**Receive**
No Key Press	1.0	1.0
Not Authenticated	1.0	1.0
Authenticated	20	17

Table 5. IPC Overheads

Input Event Overheads The *Interrupt Parser* parses the scan code of specific keys such as the *SEND, CALL, ENDCALL keys.* For the OpenMoko device, the touchscreen hardware produces screen co-ordinates which then map to scan codes. Each button on screen is represented by a range of $[x, y]$ co-ordinates. Hence the *Interrupt Parser* decodes the keys according to the range of co-ordinates for each button. It takes only $1.1\mu sec$ to check if an expected key is pressed. Thus, the SBIDF can easily detect when to proceed in the state machine even with random key presses.

IPC Overheads The *IPC Monitor* monitors each IPC Send and Receive operation over the AF_UNIX sockets. However, it only mediates IPC operations over specific sockets as defined by the specification. Also, the time taken by the IPC operation depends upon the data being transferred between the two processes. This varies for every run. Hence the results listed in Table 5 show the average time over 5 runs of sending an SMS and making a GSM call. For each IPC operation, first the *IPC Monitor* checks to see if any of the monitored keys was pressed. If not it simply returns, because that IPC operation was not triggered as a result of a hardware interrupt. Similarly, it then checks if the application that initiated the IPC operation was authenticated previously. If not, then it returns. This prevents malware from sending an SMS or modifying the microphone state. If the applications are authentic and some monitored key was pressed, then it checks which applications sent or received data and on which socket. Accordingly it decides which timer to trigger as per the specification. The cases shown here imply that the SBIDF takes on average just $20\mu s$ more while trying to prevent malware activity and negligible overhead for all other cases. This shows how the SBIDF can easily account for false alarms due to random system noise.

8 Discussions

8.1 Scalability

Specification Database We define specifications for legitimate sequence of events in the system for the attacks we aim to prevent. For each type of attack, the proposed framework defines a specification written in TLCK and enforces the specification with state machines. The number of specifications only increase if the number of attacks to be prevented increases. The length of the specification depends on the number of critical components in the system, which by design

in most phone stacks is a small number. Unlike the context-related policies [5], specifications in SBIDF focus on low-level system events.

Application to other platforms The SBIDF is a framework that uses observation of inter-component communications to detect abnormal behaviors of third party applications. Our preliminary experiments show that this framework can be implemented for the Android platform on the Android ADP1 developer phone. Android uses the Binder framework [1] for its IPC mechanism. A trace of system events in the kernel shows that the binder kernel driver is able to detect both peers involved in the IPC transaction. The critical components in this case are *rild, com.android.phone, system_server, media_server*. Since the binder framework does not use named sockets, we found that there were node identifiers for each object involved in the transaction. These objects are represented as nodes of a red-black tree per process/thread. The nodes in the tree act as senders or receivers of data. The *logcat* service on the *radio* log shows the *AT* commands exchanged with the GSM core. These commands could be parsed in the kernel driver that maps the memory for the shared memory bridge between the application processor and GSM core (smd_qmi.c). SBIDF will need to include additional hooks in the sliding keypad interface driver to mediate inputs other than the touchscreen.

8.2 Limitations and Future Work

Given the great variety of mobile phone platforms and the sophistication of attacks, we do not think any single or a few defense techniques will be sufficient for all cases. SBIDF focuses on detecting unauthorized access to sensitive services. Below we discuss some limitations as well as some ideas for future research.

Message Integrity At this stage, we do not consider the integrity of the outgoing messages. So, if there is a vulnerability in the data segment of the messaging application, once exploited, a malware could piggy-back malicious data along with an outgoing message. Here, the SBIDF will not be able to detect the alteration, but still let it pass since the message was initiated through the $SEND$ key interrupt followed by the specified IPC pattern. We think this is a challenging research problem using our framework. We might need to always trust the message server to protect the integrity of the message.

Reducing False Positives We notice that a few normal applications which automatically send SMS messages (e.g., with location data) in the background have emerged. By our design principle our system would not be able to differentiate these automated applications from malware which have similar behavior. Their difference is on the intention, not on the techniques. So we need to attest the intention of such automated messages. For an automated message detected by our system, we are considering to validate the application originating it through a graphic Turing test [16]. If the user confirms it is from an authorized application, we may white-list the application (assuming the application is not compromised yet at its first-time use). Such white-listed applications can be authenticated using our *Authentication Module*. If the application is authenticated and authorized, the message claimed to be from it will be allowed and otherwise denied.

Network Interfaces Parsing data communication via BT and WiFi to detect malicious activity without incurring high overheads and false positives is challenging. However, we think we can use SBIDF to mediate requests to power ON and OFF these interfaces, or alter their configuration at runtime. This is possible since, a real user will have to press a sequence of keys on screen. This is typically a system configuration screen with one button for BT/WiFi power ON/OFF. But we will not be able to parse events after these interfaces are switched ON, using current techniques. We plan to leverage the work of Bose et al. [3] to detect malware that tries to send sensitive files via these interfaces.

Component Authentication To protect the integrity of critical components, we use the *Authentication Module* to compute an md5 hash over the TEXT segment of the component. This authentication approach works well to protect the static part of the component. However, a component may also contain dynamic part which changes during run time. Attacker may take advantage of this dynamic part and bypass the authentication check. Verifying the dynamic part is still a hard problem requiring more effort and investigations.

9 Conclusions

The framework described in this research shows a specification based intrusion prevention approach to detect unauthorized access to sensitive services, such as SMS, audio, and video services. We believe our framework is the first of its kind to address mobile phone malware using hardware interrupts. For each type of attack, the proposed framework defines a specification written in TLCK and enforces the specification with state machines. The number of specifications only increase if the number of attacks to be prevented increases. Through evaluations, we show how the framework is capable of detecting unauthorized access to sensitive services at runtime with neglectable overhead.

10 Acknowledgment

We thank the reviewers for the valuable comments.This work was supported in part by NSF CAREER 0643906 and ARL CTA 2010-2015. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory or the U.S. Government.

References

1. Android developer phone. http://developer.android.com/.
2. F-secure report. http://www.vnunet.com/vnunet/news/2230481/f-secure-launches-mobile.
3. Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *MobiSys '08:*, 2008.
4. Jerry Cheng, Starsky H.Y. Wong, Hao Yang, and Songwu Lu. Smartsiren: virus detection and alert for smartphones. In *MobiSys '07*, 2007.

5. Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of ISC'10*.
6. Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens, and Dries Vanoverberghe. A flexible security architecture to support third-party applications on mobile devices. In *CSAW '07*, 2007.
7. William Enck, Patrick Traynor, Patrick McDaniel, and Thomas La Porta. Exploiting open functionality in sms-capable cellular networks. In *CCS '05*, 2005.
8. F-Secure. Cabir. http://www.f-secure.com/v-descs/cabir.shtml.
9. F-Secure. Commwarrior. http://www.f-secure.com/v-descs/commwarrior.shtml.
10. F-Secure. Viver. http://www.f-secure.com/v-descs/trojan_symbos_viver_a.shtml.
11. Alexander Gostev. Mobile malware evolution: An overview. http://www.viruslist.com/en/analysis?pubid=200119916.
12. Angus Kidman. Mobile viruses set to explode mobile viruses set to explode. http://www.zdnet.com.au/news/security/soa/mobile-viruses-set-to-explode/.
13. Hahnsang Kim, Joshua Smith, and Kang G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *MobiSys '08*, 2008.
14. Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *ACSAC '04*, 2004.
15. J. Seifert L. Xie, X. Zhang and S. Zhu. pbmds: A behavior-based malware detection system for cellphone devices. In *WiSec'10*, 2010.
16. Xie Liang, Zhang Xinwen, Zhu Sencun, Jaeger Trent, and Ashwin Chaugule. Designing system-level defenses against cellphone malware,. In *SRDS*, 2009.
17. ARM Limited. Arm architecture reference manual.
18. Collin Mulliner, Giovanni Vigna, David Dagon, and Wenke Lee. Using labeling to prevent cross-service attacks against smart phones. 2006.
19. Divya Muthukumaran, Anuj Sawani, Joshua Schiffman, Brian M. Jung, and Trent Jaeger. Measuring integrity on mobile phone systems. In *SACMAT '08*, 2008.
20. Yuichi Nakamura. Selinux for consumer electric devices. In *Linux Symposium*.
21. Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *ACSAC '09*.
22. William Enck Machigar Ongtang and Patrick McDaniel. On lightweight mobile phone app certification. In *CCS'09*, 2009.
23. Chen Racic, Ma. Exploiting mms vulnerabilities to stealthily exhaust mobile phone's battery. SecureComm 06.
24. R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *CCS '02*, 2002.
25. Patrick Traynor, William Enck, Patrick McDaniel, and Thomas La Porta. Mitigating attacks on open functionality in sms-capable cellular networks. In *MobiCom '06*, pages 182–193. ACM, 2006.
26. Deepak Venugopal. An efficient signature representation and matching method for mobile devices. In *WICON '06*, 2006.
27. Deepak Venugopal, Guoning Hu, and Nicoleta Roman. Intelligent virus detection on mobile devices. In *PST '06*, 2006.
28. Bjorn Vogel and Bernd Steinke. Using selinux security enforcement in linux-based embedded devices. In *MOBILWARE '08*, 2007.
29. N. Xu, F. Zhang, Y. Luo, W. Jia, and J. Teng. Stealthy video capturer: A new video-based spyware in 3g smartphones. In *WiSec'09*.