pBMDS: A Behavior-based Malware Detection System for Cellphone Devices

Liang Xie
The Pennsylvania State University
University Park, PA, USA
lxie@cse.psu.edu

Xinwen Zhang
Samsung Information System America
San Jose, CA, USA
xinwen.z@samsung.com

Jean-Pierre Seifert
Deutsche Telekom Lab and
Technical University of Berlin
jean−pierre.seifert@telekom.de

Sencun Zhu
The Pennsylvania State University
University Park, PA, USA
szhu@cse.psu.edu

ABSTRACT

Computing environments on cellphones, especially smartphones, are becoming more open and general-purpose, thus they also become attractive targets of malware. Cellphone malware not only causes privacy leakage, extra charges, and depletion of battery power, but also generates malicious traffic and drains down mobile network and service capacity. In this work we devise a novel behavior-based malware detection system named pBMDS, which adopts a probabilistic approach through correlating user inputs with system calls to detect anomalous activities in cellphones. pBMDS observes unique behaviors of the mobile phone applications and the operating users on input and output constrained devices, and leverages a Hidden Markov Model (HMM) to learn application and user behaviors from two major aspects: process state transitions and user operational patterns. Built on these, pBMDS identifies behavioral differences between malware and human users. Through extensive experiments on major smartphone platforms, we show that pBMDS can be easily deployed to existing smartphone hardware and it achieves high detection accuracy and low false positive rates in protecting major applications in smartphones.

Categories and Subject Descriptors

C.2.m [**Computer-Communication Networks**]: Miscellaneous; C.2.0 [**General**]: Security and Protections

General Terms

Security, Design, Algorithms, Experimentation, Performance

Keywords

Cellphone Malware, Behavior Learning, System Call

1. INTRODUCTION

Mobile communication systems which support both voice and data services have become ubiquitous and indispensable in people's daily lives. This popularity however comes with a price — mobile devices (e.g., smartphones and PDAs) become attractive targets of attackers. Popularity of mobile services (e.g., email, messaging) and their dependence on common software platforms such as Symbian, Windows Mobile, and Linux, have made mobile devices ever more vulnerable. This situation gets worse as mobile devices quickly evolve [5]. Today's smartphones typically have the same order/sort of processing power and functionalities as a traditional PC; hence, they are likely to face the same kind of malware that have been surging in the PC world. According to F-Secure, there were more than 350 mobile malware (including virus, worms, Trojans, and spy tools) in circulation by the end of 2007, most of which propagate via user downloading via Bluetooth and MMS [24]. Examples of some notorious threats on Symbian-based smartphones include Skull [18], Cabir [19], and Mabir [17]. McAfee's 2008 mobile security report [7] revealed that nearly 14% of global mobile users had been directly infected or had known someone who was infected by a mobile virus. The number of infected mobile devices had a strong increases in McAfee's 2009 report [8].

In this paper, we refer to *cellphone malware* as malicious codes that exploit vulnerabilities in cellphone software and propagate in networks through popular services such as Bluetooth and messaging (SMS/MMS) services. Cellphone malware are devastating to both users and network infrastructures. Users of compromised cellphones could be unconsciously charged for numerous messages delivered by malware and their phone batteries could be quickly drained. Other reported damages include loss of user data and privacy and software crashes. For example, a Trojan spy named Flexispy [4] monitors a victim's call history and contacts, and delivers these sensitive data to a remote server. In addition, automated malware which exploit buffer-overrun vulnerabilities [29] in cellphone software can generate huge unauthorized traffic and cause misuse or denial-of-service to network systems. Therefore, cellphone makers and service providers have strong motivations to employ effective countermeasures to contain or defeat these attacks [7].

Mobile computing platforms such as smartphones have unique features which have made the battle against malware even more challenging than in the desktop systems. Severe limitations on battery life, computing power, and bandwidth availability reduce the effectiveness of traditional defenses from PC platforms. A straightforward defense for cellphones based on security patches does not work well. It is difficult for cellphone users to obtain signature

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WiSec'10, March 22–24, 2010, Hoboken, New Jersey, USA.
Copyright 2010 ACM 978-1-60558-923-7/10/03 ...$5.00.

files from security vendors in a timely manner, and patch downloading requires radio resources and often causes service charges. Some recent work proposes more reliable solutions from the network side. For example, Bose et al. [14] designed an algorithm to automatically identify compromised phones based on user interactions with the wireless network and proposed a proactive containment framework to quarantine those suspicious devices. Chen et al. [15] designed a collaborative virus detection/alerting system named SmartSiren. These approaches, however, rely on network or external agents to throttle malware propagation without providing real-time and early protections on cellphones. We believe that countermeasures deployed within a cellphone itself have more strength both in detecting malware timely and in preventing malicious traffic from entering the network. This is consistent to the survey result in McAfee's 2009 mobile security report [8].

Considering the uniqueness of the cellphone platform, especially its input and output methods such as its small keypad and display, we propose a behavior-based malware detection system named pBMDS, which employs a statistical approach to learn within cellphones the behavioral difference between user initiated applications and malware compromised ones. In addition, we show how to achieve system-level protection against malware by integrating the proposed mechanism into mainstream smartphones (e.g., Symbian and Linux-based smartphones). To our knowledge, we are among the first to introduce artificial intelligence (AI) techniques into smartphones to secure their application software. The major advantage of our solution is that, its malware detection capability converges in the way that it focuses on recognizing non-human behavior instead of relying on known signatures to identify malware. Therefore, in the training process, it does not require the number of the negative samples to be equivalent to that of the positive samples. As a result, pBMDS has the capability of detecting unknown malware (e.g., zero day worms).

pBMDS provides two-level protection against both the existing and the emerging malware. Specifically, it learns not only process state transitions of the cellphone applications, but also human operational patterns during the processes of these applications. These two aspects distinctly reflect major behavioral differences between the human initiated/operated applications and the malware-compromised ones. A basic pBMDS system learns only application states to detect the existing malware. These states can be predicted by vendors and shared among cellphones. In addition to this, an enhanced pBMDS system also learns user-application profiles for cellphone users and these usage patterns acquired help pBMDS build a more powerful and accurate detection engine against even elaborated malware.

Different from previous solutions which only showed proof-of-concept models through PC or phone simulators, we have implemented our defense on major smartphone platforms and performed very thorough evaluations. Cellphone environment is resource constrained, hence any good solution should consider this uniqueness and fit into it. Our experimental results including benchmark and performance data demonstrate that our countermeasure is capable of effective identifying and blocking malware within a wide variety of smartphones (detection rates could exceed 99% in our experiments). Also, our defense is lightweight and can be conveniently deployed with the existing smartphone hardware.

Outline: The rest of the paper is organized as follows. In Section 2, we discuss why behavior-based solution is effective and practical for cellphone devices. Section 3 provides an overview of our proposed behavior-based countermeasure. We present design and implementation details of the defense in Section 4 and evaluate its effectiveness and performance through experiments in Section 5. We introduce related work in Section 7 and conclude in Section 8.

2. ATTACK MODEL

We consider a general attack model in which a malware such as Commwarrior [27] adopts MMS/SMS and/or Bluetooth as its major infection vector. Specifically, after compromising a cellphone, the malware executes its malicious code to propagate and cause damages to the cellphone. Typically, a MMS/SMS-based malware scans the phone address book and/or call history and randomly chooses some contacts inside and sends malicious messages to these new victims. A Bluetooth-based malware takes control of the victim phone's Bluetooth interface and continuously scans other Bluetooth-enabled cellphones within its range. Once a new target is detected, the malware inter-connects two devices and transfers a malicious file to the target, which easily gets infected and becomes a new attack source.

Riding on these propagation vehicles, a malware is able to spread more damages to other vulnerable cellphones. For example, a user's private information such as her friends' names and contacts could be stealthily collected by a malware (e.g., a Trojan spy named Flexispy [4]) and delivered to some external servers; unconsciously delivering numerous messages will quickly deplete a compromised cellphone's battery power; running applications could crash due to the malware attack and important data could be erased from the device.

In our discussion, we assume that malware always launch attacks from the application software. They could compromise phone applications such as email and MMS/SMS messaging, but they cannot break the kernel. To out best knowledge and according to F-Secure [24], kernel level rootkits have not been found on cellphone devices. Also, we notice that there are a few techniques which can be adopted in cellphones to prevent kernel-hacking, e.g., using integrity measurement [34, 31] to identify falsifications on kernel code, and using hypervisor techniques [6, 12] to isolate attacks from a legal OS on embedded devices. These defenses are not the focus of this paper.

3. OVERVIEW OF APPROACH

3.1 Basic Principle

A cellphone application (e.g., MMS agent) typically involves a series of GUI interactions between the user and the device. For example, to compose an MMS message, a user activates an input window on the phone screen and enters message content, i.e., she goes through a series of GUI cycles by touching the keypad and reacting to the display on the LCD. These GUI interactions can be recorded by the keyboard and display drivers at kernel level. Essentially, a messaging process invokes a series of key system calls to access resources (e.g., file, socket) and acquire system services when delivering the message. For example, to search for the recipient's email address or phone number, a communication process calls *open("address_book", O_RDONLY)* and reads the contact list in the address book; to send a message through Wi-Fi, an email process named $smtpclient$ first calls *socket(AF_INET, SOCK_STREAM)* to create a stream socket and then communicates with the SMTP server; to deliver an MMS/SMS message, a process named $mmsclient$ calls *fd=open("/dev/ttyS0", O_RDWR)* to open the modem device $ttyS0$ and delivers the composed message to its buffer by invoking *write(fd, message, length)*. The modem then transmits it to the air interface. Figure 1 illustrates a partial behavior-graph of an email/MMS messaging process. We can see that between a pair of key system calls (including the starting

point), there exist a series of keyboard-display interactions which cause process state transitions.

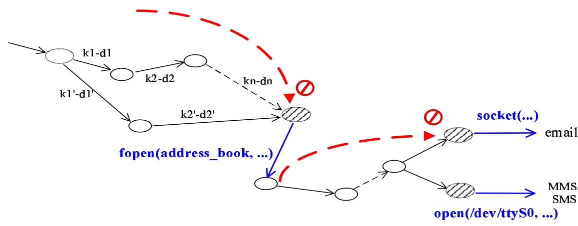


Figure 1: **A partial behavior-graph of the email/messaging process. Each node represents a process state and each directed edge denotes a transition (through function calls); a shadowed node represents a state to execute the key system calls; {k-d} denotes a round of keyboard-display interaction between user and device.**

We notice that these key system calls are important monitoring points in the kernel, because application-layer malware tend to invoke these system calls as normal processes do to gain accesses to important system resources and use system services to launch attacks. Therefore, to examine the difference between a normal application and a malicious process, we first investigate the process behaviors between consecutive key system calls, e.g., $(.) \rightarrow open()$, $open() \rightarrow socket()$, where $(.)$ denotes the starting point. Once a process deviates from its expected normal behavior (reflected through GUI interactions) to a certain degree, it is considered suspicious and the access to the system resources should be denied (as illustrated in Figure 1). Therefore, we can verify key system calls in the kernel level by authenticating the behavior of the on-going process.

Malware demonstrate their malicious behaviors in compromising cellphones and/or in propagating to other victims. The behaviors are essentially different from those of normal applications initiated from human beings in that either malware make use of system resources and require system services in an unexpected way to launch attacks, or malware cannot simulate normal human operations on cellphones which follow some user-specific patterns of usages and reflect human intelligence. First, from the application's point of view, malware attacks always cause anomalies in process states and state transitions. Such anomalies are reflected through malware's function (API) calls, usages of system resources, and requests for system services. We adopt function call-trace techniques [21, 26] and human intelligence techniques in the context of cellphones to identify process misbehavior. Second, from the user operation's point of view, each cellphone user has his/her own unique and private operational patterns (e.g., while operating keypad or touch-screen), which cannot be easily learned and simulated by malware. From these two aspects, our behavior-based malware detection system (pBMDS) provides comprehensive protection against malware. pBMDS leverages a Hidden Markov Model (HMM) to learn process behaviors (states and state transitions) and additionally user operational patterns, such that it can effectively identify behavior difference between malware and human users for various cellphone applications.

The above problem space could be huge and complex in the PC platform (even in a cellphone environment), because we need to monitor a great many function (system) calls to identify anomalies. We propose a Hidden Markov Model (HMM) based malware detection engine which takes only a limited number of observations (user inputs) as the input and associates process states (hidden states that cannot be directly observed) and their transitions with these observations. We note that user operations are more predictable on cellphones. Specifically, compared with more complex user behaviors in PCs, cellphone users' behaviors have the following unique features due to the constrained input and output methods which lead to the design of pBMDS. First, cellphone GUI has been designed to be very intuitive and simple to enable comfortable and convenient operations, hence user input behaviors in this context are relatively predicable. Second, cellphone keypads have less buttons than PC keyboards. Nokia Communicator 9300 with a "full keyboard" has 58 character keys, whereas a PC keyboard normally has over 100 keys. Most cellphones today have keypads with less than 25 keys. Third, many high-end smartphones have touch-screens which support flexible user input methods such as virtual keyboard and handwriting recognition. Fourth, cellphones are more privately owned than PCs. Therefore, user operation patterns are more meaningful in cellphones, and this greatly reduces the problem space, which means pBMDS only needs to monitor some designated system calls to achieve its effective behavior-based authentication.

3.2 Architecture and Components

Figure 2 shows the architecture of pBMDS, which consists of a behavior analyzer, a learning engine, a system call monitor and a malware detection engine. The first two components (shadowed blocks) belong to the training phase, and the other two belong to the real-time detection phase.

To perform automated malware detection, we need to analyze user behaviors in cellphone applications such as messaging. Initially, the behavior analyzer collects event logs of keyboard operations and LCD displays and correlates pairs of input/output events using their time stamps in the logs. These event pairs reflect intermediate process states during the service. For example, a user follows the menu to input a recipient's number either through retrieving recently received calls or through looking up the address book. Meanwhile, the analyzer collects all the raw input/output pairs and refers to an adjustable policy file to construct a behavior graph, which reflects intermediate process states towards a key system call; for instance, it uses keypad events to generate a behavior graph G_{open} for system call *open()*. The policy file (policy 1) helps perform a first-level sanitization and filtering on the raw events and controls the granularity of the process states to be reflected in the graph. In this way, a set of raw GUI events can be converted to a meaningful data set which characterizes the process's behavior.

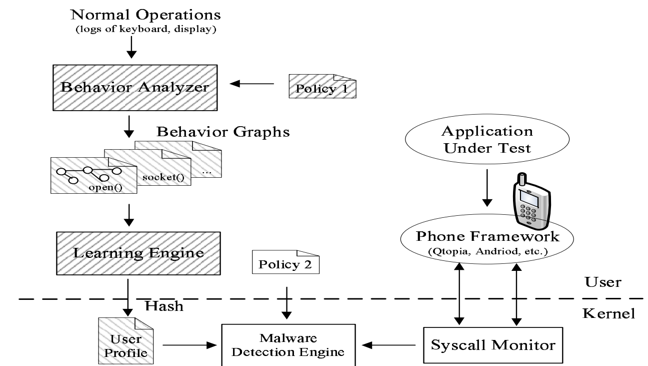


Figure 2: **Behavior-based malware detection system in cellphones.**

In the next step, a learning engine extracts *user-behavior features* from the application. This can be done through observing the key/display event pairs in the graph and extracting unique features including both the user's personal operational patterns (e.g., time between keystrokes, keystroke durations, and pressure on touch-

screen) and the sequence of process state transitions. For instance, when the user uses navigation key to choose entries in the phonebook menu, we can obtain a rule for the key sequence starting with K_{menu}

$$K_{menu} \rightarrow K_{navi} \rightarrow K_{navi} \rightarrow ... \rightarrow K_{enter}.$$

These captured key sequences (observations) potentially reflect the state transitions of the current cellphone application. To directly monitor or measure process state transitions, we can monitor and measure all system calls that have been invoked during the application execution. However, this is quite complex, especially in the context of cellphone, because it involves many system calls and hence a lot of process states. In our design, we introduce a Hidden Markov Model (HMM) based malware detection engine, in which the HMM takes only those limited number of observations (user inputs) as the input and associates the number of process states (hidden states that cannot be directly observed) with the number of observations. As such, the complexity of the malware learning/detection engine in cellphone environments can be greatly reduced. In addition, the HMM-based learning engine records a user's operational patterns (preferences in providing the keypad/touch screen inputs) and learns a specific operational profile for the user. This additional profile can be used to increase the detection accuracy of pBMDS and defeat more elaborated malware in the future.

After learning the user operational patterns and the state transition rules, the engine encodes the learning result by hashing it to the kernel, where it can be safely saved as a user profile. Note that a phone can have multiple user profiles stored in the kernel, although in most cases it has only one user. To test whether a running process is malware or compromised by malware, the run-time detection engine authenticates at key system call points [1] whether the program behaviors follow the user profile and the detection policy. A policy (policy 2) defines a set of verifiable properties for each key system call. For example, the following one says that an application

$$\begin{aligned} &Permit\ open && as && user\ Alice && (1)\\ &Parameter && 0 && equals\ /opt/qtopia/phonebook \\ &Parameter && 1 && equals\ 85\% \end{aligned}$$

can invoke the $open()$ system call only when its behavior pattern follows (with 85% similarity) Alice's profile, and its input parameter is "/opt/qtopia/phonebook". Thus, the application's behaviors (i.e., process states and operations) are examined based on existing user profiles and its access to important system resources will be denied if it deviates from the normal process to a certain degree.

4. DESIGN AND IMPLEMENTATION

4.1 User Behavior Analyzer

To track process behaviors and record related user operations to obtain representative user patterns, we monitor system I/O events such as a user's keypad/touch-screen inputs and consequent LCD displays, and further examine correlations of these events. Cellphone platform has its unique I/O features: flexible input methods, limited number of key codes, and event-driven displays. Therefore, the first issue here is to decide where to set monitoring points within the device and what granularity event logging should take, as it affects the complexity of user behavior analysis. Below we use Qtopia [2] in Linux-based smartphone as an example to describe the process.

[1] We discuss details of system call monitoring in Section 4.3.

Monitoring Input Events Fig.3 shows how user inputs from keypad/touchscreen are processed in Qtopia (Qt). When a user presses a key, the keypad sends corresponding raw scancodes to keyboard driver (keyboard.c) in the kernel. The *handle_scancode()* function in the keypad driver parses the stream of scancodes and converts it into a series of key press and release events called keycode by using a translation-table via *kbd_translate()* function. Each key is provided with a unique pair of keycodes. For example, pressing key *k* produces keycode *k* (1 ∼ 127), while releasing it produces keycode *k+128*. After the above handling, the obtained characters are periodically put into the standard input device */dev/tty0*.

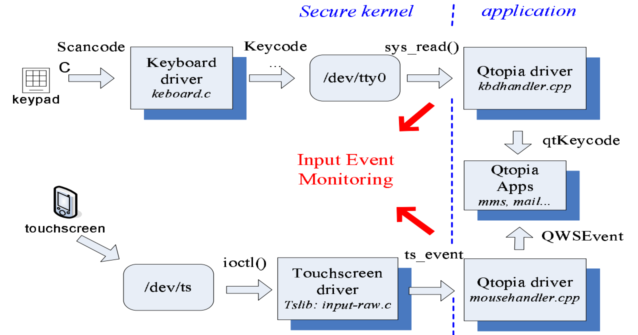


Figure 3: Logging user input events in a cell phone device.

When a window view in user process (e.g., $mmsclient$) requires an input, it uses the application-layer driver interface $kbdhandler$ to read keypad input by calling $KbdHandler :: readKbdData()$, which invokes a system call $sys_read()$ to fetch input keycodes from */dev/tty0*. Once a keycode has been read, $kbdhandler$ translates it into a Qt event and passes it to the user process. For instance, when a user presses MENU/SELECT (keycode 0x1c and 0x9c for push and release), the application is notified of a Qt event ($qtKeycode = Qt :: Key_Select$) and invokes a function named *processKeyEvent(qtKeycode, ...)* to handle the input.

To monitor user keypad inputs, we choose to place a keystroke hook in the kernel to intercept and log user keypad events before they are passed to user processes. Specifically, we insert a hook in system call $sys_read()$ so that whenever an user application reads keycodes from the standard input device, the modified system call first executes the hook function, which records the keystroke on its keycode pair, duration between the key press and release, and the time-stamp of this input event. For example, when the user presses key 'a', we collect the following event in the log file.

$$\{time_stamp : code_down('a'), duration, code_up('a')\}$$

For touch-screen inputs, we place a hook in the universal touchscreen driver $tslib$ ($input{-}raw.c$) to monitor raw events (ts_event) received by user applications, as shown in Fig.3. We collect the following attributes for each ts_event:

$$\{time_stamp : pos.x, pos.y, pressure\}$$

where $\{pos.x, pos.y\}$ and $pressure$ denote the touch position and pressure on the screen, respectively. Note that when an application view reads a touch input, it translates the raw event into $QWSEvent$ and further explains it. For example in Fig.4, $PhoneLaunchView$ (GUI application) maps the touch position into a meaningful item code which is then processed in the same way as a keypad input.

An alternative of the above kernel-level monitoring is to trace input events through driver interfaces in application layer. We may

```
PhoneLaunchView::mousePressEvent(QWSEvent *event)
{
    itemCode = getItemAt(event→pos());
    updateUI(itemCode);
    /* processed like a keypad input */
    processItem(itemCode);
    ...}
```

Figure 4: Mapping a touch-screen event to an application item.

insert hooks to $kbdhandler$.*cpp* and $mousehandler$.*cpp* for logging $qtKeycode$ and $QWSEvent$ (Fig.3). However, this makes event traces less reliable because these interfaces themselves could be compromised by malware. Moreover, in addition to events, kernel-level monitoring provides details on user-specific input patterns such as one's keystroke speed, touch-screen pressure, etc.

Sanitizing and Filtering Input Events A user could inadvertently generate some invalid inputs to an application view. For example, when she opens address book and uses navigation keys to choose a contact, only up/down/select key inputs are processed by the current active window view named $AddressbookWindow$; other key inputs such as left/right navigation keys and '*', '#' are ignored. Since our event-logger intercepts raw inputs from kernel, event records need to be sanitized and filtered before being further analyzed.

Our solution is to correlate keyboard/touch-screen inputs with LCD displays, such that only when the display frame buffer has some corresponding data output to the LCD screen within a short time period, can the input be considered valid; otherwise, it is aborted. This correlation is based on the time-stamps of a pair of input/ouput events. We first monitor system calls in the frame buffer driver to record output events. Specifically, we place a hook in a driver named $Qscreenlinuxfb$.*cpp* (in Qtopia Core) to record the occurrences of system-call *msync()*, which is invoked during each display event. In cellphone platform OS such as Linux, frame buffer is an abstract device ($/dev/fb0$) which allows user processes to write output directly to the LCD video memory. An active application view can use system call *open(/dev/fb0,...)* to open the frame buffer device and use *mmap()* to map the device to its logical address space. Whenever this application generates output to display, it directly writes data to this mapped address space and immediately invokes *msync()* to flush this new data to LCD screen. When an application view becomes deactivated, Qtopia Core invokes system call *munmap()* to cancel this address mapping.

We use a time-stamp to label each address-mapping system call mentioned above. To verify an input, we search the display event-log and see if there is a *msync()* which immediately follows the input event within a certain time window. Because each valid input incurs an update of the current application view and the corresponding output will be written to the frame buffer, at least one *msync()* should be detected shortly. Due to the difficulty of monitoring the content of frame-buffer, this time-based correlation is more practical. One issue here is the length of the time window. For most cellphones, display response towards an input is fast and we can use a short time window (≤ 0.5 second). Also, a series of display events are continuous and closely related with each other. They should not be broken by next user input event.

Generating Behavior Graphs After sanitizing user input events, the next step is to generate a behavior graph which reflects intermediate process states towards each key system call. As we have mentioned earlier, key system calls such as *open()* and *socket()* are invoked by a number of processes for accessing important system resources (e.g., address book and modem). We aim at examining the intermediate states of these resource-requesting processes. However, these process states are hidden from us and what we can observe is the user's keypad operations and the GUI displays on the screen. These observations faithfully reflect the behaviors of an on-going process. For example, applications such as messaging, voice calls, and contact lookup/edit each could open the address book file to achieve its goal. Therefore, for system call $open(\text{"}addressbook\text{"}, ...)$, we exploit the application's event log and construct a directed behavior graph G_{open} based on the observations from keypad/touch-screen inputs and GUI displays. In this graph, we define observations as graph nodes and evolutions between observations as directed edges.

Fig.5 illustrates the behavior graph for a simple text-messaging process. To simplify the example, we show only keypad inputs. Raw inputs from keypad are categorized into a number of independent *key observations*, each includes one or a number of consecutive key operations and represents a node in the graph. For example, observation O_NUM represents operations on number keys (0-9) and observation *O_MSG* stands for an operation on the messaging key itself. We then examine evolutions (links) between these observations (nodes). As shown in Fig.5, a user who initiates an SMS process first enters the GUI by pressing a messaging key and chooses an appropriate service type using navigation keys (step $1 \sim 4$), then she composes message content using numeric and alphabetic key combinations and starts the delivery through a menu selection (step $5 \sim 7$). Finally, she presses navigation keys (up/down) to select a recipient from the address book and touches the send key to transmit the message (step $8 \sim 10$). Information about these observations and related transitions are obtained from event logs. Of course, there could be other optional branches the user chooses to complete the messaging. However, these branches should basically follow this behavior graph if they are generated by this user. Note that the behavior graph can also reflect the user's operational pattern in the process. Each observation can be associated with its key touch duration and each evolution can be associated with a time duration. These time values reflect a user's preference in keypad operations and her familiarity with the messaging process.

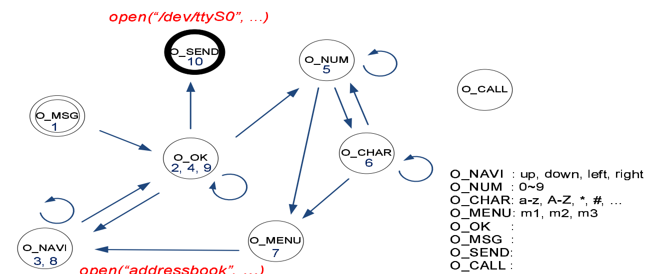


Figure 5: A simple keypad-based behavior graph for a text messaging process (SMS). Each key observation is labeled with one/multiple sequence number(s) during the process state transitions.

4.2 Behavior Learning Engine

Our behavior learning engine takes behavior graphs and optional user input patterns from these graphs as inputs and learns intermediate state transitions of applications.

Profiling Process Behaviors using HMM To profile process behaviors in cellphones, we use a Hidden Markov Model (HMM) [32]. HMMs are often used to model finite-state stochastic processes, in which the true *state* of a system is unknown and thus is represented with hidden random variables. What is known are *ob-*

servations that depend on states, which are represented with *known* output variables. One common problem of interest in an HMM is the *parameter estimation problem*, where parameters such as the transition probabilities among states are learned (estimated) from sequences of observations [32].

Our first step is to decide the number of states N to be used for malware-targeted applications in cellphones. Traditional methods [40] suggest choosing process states roughly corresponding to the number of distinct system calls used by the program. However, in our case it is neither feasible to assume processes invoke a fixed set of system calls nor scalable to deal with too many states ($60 \leq N \leq 70$) in a resource-constrained cellphone. We choose the model size N roughly corresponding to the number of unique key observations, which is limited due to a limited number of keys on modern cellphone devices. Note that process states are fully connected and transitions are allowed from any state to any other state; key observations are essentially captured user reactions towards the application GUI which reflects process states, driven by appropriate key inputs.

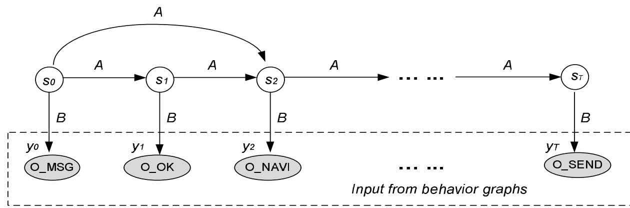


Figure 6: **Transitions among T process states during messaging; $T \leq N$ states are involved.**

Fig.6 shows the HMM state transitions of a single messaging process. It is represented as a statistical graphical model. Circles represent random variables. Shaded circles (y_i) are key observations while unshaded circles (s_i) are unknown state variables. The arrows from s_i to s_{i+1} and from s_i to y_i indicate that the latter is conditionally dependent on the former; the value on the arrow is an entry in the probability matrix. So here we have $p(s_{i+1}|s_i) = a_{s_i,s_{i+1}}$, which is the probability of state s_{i+1} appearing after state s_i. We also have $p(y_i|s_i) = b_{s_i}(y_i)$, which is the probability the user reacts y_i to state s_i in the previous step. Note that for simplicity, here we only illustrate a simple HMM based on the behavior graphs of text messaging. In real implementation, an uncompromised cellphone records the user's normal operations during her phone calls, messaging and contact lookups/changes, etc. The behavior graphs of these key-resource–accessing processes are then generated and fed into the HMM as input data (observations), such that transition probabilities (1) between each pair of process states (matrix A), and (2) between each pair of process states and observations (matrix B) can be statistically learned.

HMMs learning can be conducted by the *Baum-Welch* [36] or *forward-backward algorithm* [32]. The latter belongs to a generalized Expectation-Maximization (EM) approach.

Profiling User Operational Preference Traditional HMM learning process [32], however, does not consider a user's operational preference during state transitions. Here we use the time feature of user operations as an example to show how to augment an HMM in learning a cellphone user's normal behaviors. As we have mentioned earlier, within a behavior graph we also measure the averaged key press/release duration for each key observation and the time duration for each evolution between key observations. Therefore, we can exploit this time information to augment a standard HMM.

In our case, we aim at building an HMM model $\lambda = (A, B, \pi)$, which includes state transition probability distribution A, observation symbol probability distribution B, and initial state distribution π. We derive a detailed profiling process in Appendix A.

Two-Level Behavior Learning Initially a cellphone is not compromised by any malware right after being produced from its vendor and sold to a customer. Event records of normal user activities such as voice/data calls, messaging, and emailing can be automatically collected by pBMDS. These records are then used to generate training data for behavior learning. A user can also add more activity data later to further improve the learning engine. For example, after she has done some operations during messaging, the system challenges her whether her logs are valid when she starts to send the message [38, 39].

We note that two types of profiles can be generated by the pBMDS learning engine, namely the *application profile* and the *user-application profile*. In the former case, only key/touch-screen sequences (in behavior graphs) are included in the learning process; user operational preferences such as transition time and speed are not considered. Therefore, a standard HMM can be adopted for training and the result profile, for example, $\lambda_{mms} = (A, B, \pi)$, only reflects normal process behaviors of the standard MMS application. In the latter case, both key/touch-screen sequences and user operational patterns are included, hence an extended version of HMM is required for training/testing, and the result profile, for example, $\lambda_{(mms,u)} = (A^{'}, B^{'}, \pi^{'})$, reflects normal process behavior of MMS which is operated by user u (i.e., follows user u's operational preferences). A basic pBMDS system learns only application profiles for malware detection and these profiles can be generated by vendors and shared among cellphones with the same application framework; an enhanced pBMDS system learns user-application profiles for different users and applications, and these profiles are different between cellphones. Because a cellphone only has a limited number of applications and it is owned by a single user, the pBMDS learning engine is quite scalable in this context.

Profiles generated by the behavior learning engine are sensitive and should be securely stored in a cellphone, such that malware cannot access and falsify their contents. One approach to achieve secrecy is to encrypt the profiles using the detection engine's public key; a detection process decrypts the profiles before using them for malware detection. To ensure integrity of the profiles, the behavior learning engine can hash the learning result and store the value in kernel space or in a trusted hardware such as trusted platform module (TPM) [11]. Each time when a detection process retrieves the profile, it first verifies the correctness of the profile by re-computing its hash and checking with the stored hash value. Although this approach ensures both profile confidentiality and integrity, encryption-decryption overhead and key distribution complexity are involved. An relatively easier approach is to leverage existing kernel-level security mechanisms (e.g., platform security in Symbian OS [23] or embedded SELinux [41]) to protect the profiles in file systems. For example, through defining a mandatory access control (MAC) policy, profiles are stored in a specific directory which can only be accessed by the detection process; any illegal access is blocked. Enabling such a per-process–based security mechanism also incurs overhead to the system.

4.3 Malware Detection Engine

Key System Call Monitoring We first discuss key system resources in cellphones which could be misused by malware. Malware exploit these resources through invoking a series of key system calls to launch attacks on local devices and propagate throughout networks. Table 1 shows some possible exploits in an OpenMoko

Table 1: Possible monitoring points in an OpenMoko smartphone

Key System Resources	Devices Accessed	Some Key System (or API) Calls Invoked	Comments
User address book	Address book in phone Address book in SIM card	fd=open("adr_file", O_RD); read(fd, buf, data_len); see AT commands to modem below, use read() system all	

GSM/GPRS Modem
used for messaging, email
	/dev/ttyS0
	/dev/gprs0 network device
		fd=open("/dev/ttyS0", O_RDWR); write(fd, cmd, len); // AT cmd
		fd=open("/dev/gprs0", O_RDWR);
		socket(AF_INET, SOCK_STREAM, 0), // web browser, email, etc	

WiFi interface	/dev/ar6000 low-level packet delivery wi-fi sockets	read, write, open, close, ioctl provided by sdio ar6000 wi-fi driver htc_connect()/htc_send()/htc_receive() via HIFReadWrite() interface socket(AF_INET, SOCK_STREAM, 0),...	Neo FreeRunner

Bluetooth interface	/dev/bluetooth/rfcomm/0
	bluetooth_rfcomm socket
		open("../rfcomm/0", O_RDONLY|O_NOCTTY);
		socket(AF_BLUETOOTH, SOCK_RAW, BTPROTO_RFCOMM);
		m_socket->connect(local, remote, channel),...;
			based on L2CAP

smartphone [10]. Taking a Bluetooth-based malware (e.g., Mabir) as an example, after it has compromised cellphone software, it stealthily opens $/dev/bluetooth/rfcomm/0$ (a virtual serial port device) and creates a client socket (e.g., $RFCOMM$) to establish a stream-based Bluetooth connection with each external victims target. These connections are used for transferring malicious data. Similarly, system calls that must be invoked for various communication purposes (shown in Table 1) are the most common system-level entries visited by malware, hence they are the most important monitoring points in pBMDS. Another reason for choosing these system calls is that they are the *last automated steps towards the completion of both a normal and a compromised communication process*, i.e., these are the key points at which we can use previously-built behavior knowledge to differentiate normal users and malware.

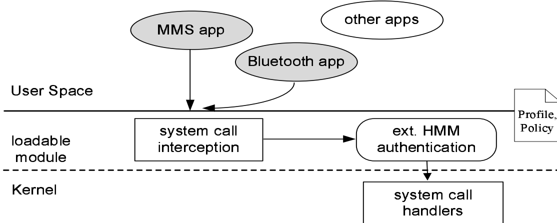


Figure 7: Malware detection engine in cellphone

We place a set of hooks in kernel space so that whenever these sensitive system calls are invoked by a user process, the corresponding hooks start the authentication (described below) and decide whether to execute the original (intercepted) system calls. We adopt a similar mechanism as Linux Security Module (LSM) and build a kernel module for the malware detection engine. When this module is loaded into the platform OS, the set of hooks are enabled and security protection is launched on the cellphone, as shown in Figure 7.

HMM-based Malware Detection To authenticate the initiator of the key system call, the malware detection engine tests whether the current trace of the application is intrusive, i.e., whether it deviates from normal process behavior. Basically, this can be achieved by computing the likelihood of a given observation sequence based on a learned HMM model. Specifically, we use profile $\lambda = (A, B, \pi)$ to derive the probability $P(O|\lambda)$ for a given observation sequence $O = O_1, O_2, ..., O_T$ traced from an application. If this $P(O|\lambda)$ is lower than a threshold τ, the current application under investigation is intrusive; otherwise, it is normal. Note that in a case when pBMDS generates only application profiles, we can directly use the standard forward algorithm to derive this probability [32]. However, when the system adopts user-application profiles, for instance, $\lambda_{mms,u}$, the *forward-backward algorithm* needs to be extended for computing $P(O|\lambda = \lambda_{mms,u})$, the probability of the observation sequence given knowledge on user u's operational patterns during MMS applications. We show a detailed detection procedure in Appendix B.

To determine a default threshold τ, we use the cross validation approach [1] in statistics. Specifically, some captured events representing either normal or abnormal process behaviors are first divided into a *training set* and a *cross validation set*. The former is used to learn the model λ and the latter is used to estimate the detection threshold τ through confirming and validating the learning result.

As we have mentioned in Section 3, our malware detection engine is policy-based, i.e., a policy file which can be securely passed to the loadable module (e.g., using a vfs or *netlink-like* mechanism) specifies a set of malware detection parameters for configuring the malware detection engine. As shown in Equation 1, a policy file typically defines the application to investigate, the key resources and the system calls to monitor, the user profile, and the decision criterion – the expected behavior similarity (i.e., $P(O|\lambda)$) to identify a normal process, for example, 85% in Equation 1. Note that in real applications, this criterion should be fine-tuned according to τ, the default detection threshold initially learned from training data.

Detection Errors We know that a user's operations sometimes do not exactly follow expected patterns. For example, during messaging, a user could inadvertently push invalid key combinations, or push several keys and pause for a while, then continue. These could bring wrong key combinations or biased key-touch durations to the observations, thus causing false negatives in the detection engine. We have already mentioned in Section 4.1 that invalid user events can be effectively filtered by correlating keypad/touchscreen inputs with the LCD outputs. Moreover, when a user generates mostly wrong key combinations, the application will not eventually reach the designated monitoring points. For example, during messaging, the system call *fd=open("/dev/ttyS0", O_RDWR)* can only be invoked when a user has passed all the major operational stages. Biased key-touch durations from the same user also cause false negatives. However, this can be prevented either by the application itself or by the event sanitizer, through defining a normal time threshold between consecutive key touches. Most cell phones start a screen-saver or turn off LCD screen after a long wait for an expected user's keypad/touchscreen input. In pBMDS, we employ the following mechanism to mitigate this detection error. Basically, each time when the detection engine authenticates an ongoing process and gets $P(O|\lambda) < \tau$, it does not immediately conclude the program as illegitimate. Instead, it challenges the user through a trusted application, such as a Turing test [38, 39], and verifies whether the current detection result is merely a false posi-

tive, which should of course be ignored. Meanwhile, the detection engine also fine-tunes its decision criterion in the policy to reduce such false positives based on challenge results.

We note that a cross validation process also helps reduce false positives. Generally, when more training samples (positive and negative) are provided, the classifier (detector) becomes more accurate. However, pBMDS only requires a small number of negative samples to achieve high accuracy, i.e., the learning process is heavily biased on positive samples. Its performance does not rely on known malware signatures. Instead, it mostly depends on the convergence of cellphone users' normal behavior and it aims at detecting those outliers. We evaluate the performance of pBMDS in Section 5.

5. EVALUATIONS

In this section, we evaluate the feasibility and effectiveness of our countermeasure. We first test the case when pBMDS is deployed to secure typical cellphone applications without considering user profiles. We then combine process state transitions and user operational preferences to evaluate the effectiveness of an enhanced pBMDS engine. Time performance of the defense is also measured.

5.1 Experimental Settings

Linux-based Smartphone Linux-based smartphones have had substantial growth in recent years. In our tests, we chose OpenMoko [10]–a Linux-based smartphone, and OMAP-5912 OSK [3]–a generic platform for developing both Windows CE and Linux-based smartphones. The OMAP board includes TI processor ARM926-TEJ operating at 192MHZ, 32 MB RAM, Mistral's Q-VGA and touch-screen, ethernet/USB/serial port. Original OMAP board ships with MontaVisa Linux OS for OSK (kernel 2.4). We customized it to use kernel version 2.6.20.4 for deploying our kernel-resident defense mechanism. To develop phone applications based on the hardware, we chose Trolltech [2] Qtopia Phone Edition 4.2 as the application platform. Qtopia is currently running on a wide variety of Linux-based phones (e.g., Motorola A1200 and OpenMoko), and it provides a complete set of C++ SDKs, user-friendly tools and APIs to application developers. Our source codes of malware and defenses were first built into PC executables and tested in a Linux-based emulator. Finally, these programs were cross-compiled into target executables for the ARM-9 processor and deployed to the OMAP board.

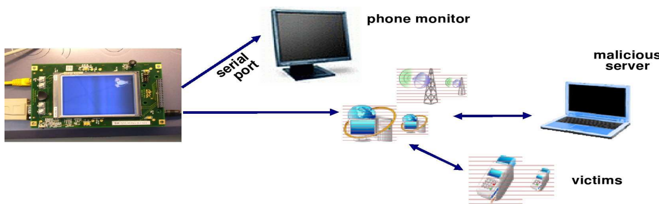


Figure 8: **Configuration of experiments (OMAP-5912OSK platform).**

Experiment Configuration Figure 8 shows the configuration of our smartphone experiments. An administrator can control an OMAP board and log its events through an external *Minicom* terminal (serial port). Each OMAP board connects to a standard modem device through which it communicates with other smartphones within 2G/3G cellular networks. In addition, each board can access the Internet through its on-board network interface. We also implemented an external malicious server, which establishes connections (SSL) with the on-board malware and receives private user contact information stealthily gathered from the OMAP board. Therefore, besides the malware attacks inside the board, the malicious server itself can exploit disclosed user information and launch automated messaging attacks to vulnerable phones by executing its own messaging service such as *sendmail*.

We implemented three major malware: Cabir [19], Comm-Warrior [27], and Lasco [16] in both the OpenMoko smartphone and the OMAP board. Although these malware were only reported from Symbian smartphones [24], we extended these cases to Linux environment. In our implementation, when a cellphone user unwittingly opens an attachment in a message titled "Breaking News", a CommWarrior process is started and it randomly retrieves recipients from address book and secretly delivers (in background) malicious messages with similar attractive titles to these victims. In each of these messages, malware specifies the MIME type as a valid installation/application file and attaches a malicious file which contains a complete installation of CommWarrior.

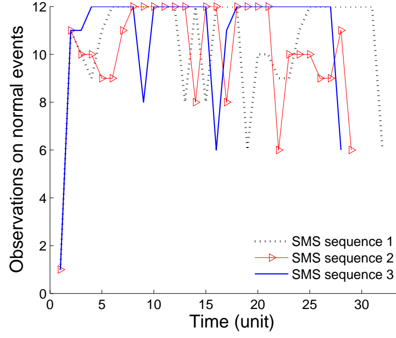
In addition, we implemented another attacking strategy of automated malware on our evaluation platforms. Using email service in OMAP-5912OSK as an example, when a user just clicks on (or highlights) a newly arrived message entitled "Breaking News", a running process named *qtmail* which belongs to Qtopia's email framework invokes the function *EmailHandler::mailRead(Email* mail)* to process the message and interpret it to the screen. However, there is a buffer-overrun vulnerability inside this function (e.g., no boundary check for temporary storage of the message content). Malware exploits this vulnerability and hijacks the program flow by replacing the return address of this function. Now that the execution of *qtmail* has been redirected to the injected code[2], which starts a malware timer by executing *QTimer::start()* and associates the timer event to an attack function named *attackLoop()*. This *attackLoop()* incurs similar attacks as we have mentioned above. Similarly, for MMS messaging, malware seeks buffer-overrun vulnerability in a function named *MMSHandler::MMSRead()*, which is invoked by the MMS messaging process *qtmms*. We note that in these cases, malware do not add any message attachments and do not wait for user reactions to install themselves.

To implement more elaborated malware, we used the above automated attack strategy and let the malware code also simulate the program behavior of the 'hijacked' messaging process. Specifically, a malware which has gained control of the messaging process not only executes the malicious code to launch its attack, but also tries to simulate the entire or some part of the messaging process. Note that such attacks can bypass a malware detection engine which works based on verifying system states at certain checkpoints.

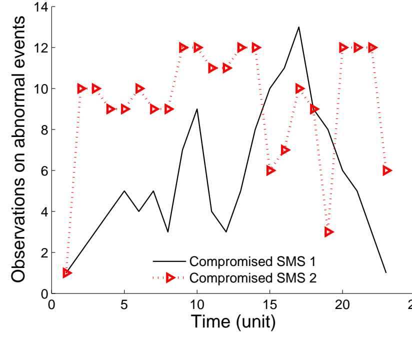
5.2 Experimental Results

pBMDS without User Operational Preferences We chose Short Message Service (SMS) as an example to demonstrate behavior difference between normal processes and malware compromised ones. For normal SMS applications, we monitored 10 different users' keypad and touch-screen input events on both OMAP board and OpenMoko. For compromised SMS applications, we tested two different cases when malware adopt different attack strategies as we mentioned above. Furthermore, we let malware randomly invoke keypad input events to simulate normal process behavior in the second strategy. Fig.9 illustrates the test result. From Fig.9(a) we can see that, although users (here we only show 3 of them) have

[2]This malicious code could either exist in the message content or be pre-injected into some library files such as libqtmail.so and libqtmms.so by the malware.



(a) Normal SMS processes



(b) Compromised SMS (simulated keypad events)

Figure 9: Behavior difference between normal/compromised SMS applications; each number in y-axis represents one type of keypad observations (as defined in Figure 5)

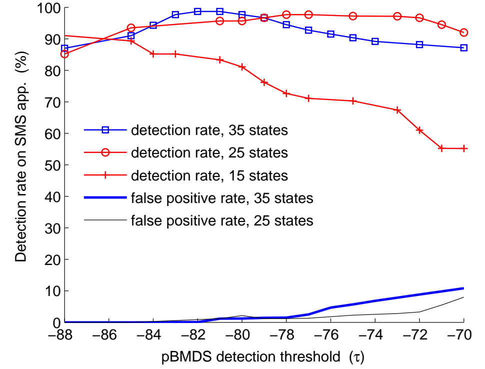


Figure 10: pBMDS in SMS messaging

different operational patterns and their ways of starting and operating in cellphone applications (here SMS) may slightly vary, normal process behaviors tend to converge and follow a normal path of application states, reflecting through a normal set of program state transitions. Fig.9(b) shows that, even for some intelligent malware (in current stage they are not) which can simulate random input events, their behaviors (hence program state transitions) distinctively deviate from those of the normal cases, not mentioning their simulations of human behaviors incur incorrect displays on the cellphone screen which could remind users of on-going attacks. Because all existing cellphone malware do not even simulate users' input events [24], their abnormal behaviors which cause abnormal process state transitions can be easily detected by our pBMDS system.

To evaluate the effectiveness of pBMDS in securing SMS processes, we collected 10 users' keypad and touch-screen inputs to generate normal observations and used randomly generated inputs from malware (with both attack strategies) as negative samples. Specifically, we provided 317 complete SMS sequences and 29 incomplete (or failed) SMS sequences of observations from these 10 users, and provided 27 abnormal SMS sequences of observations randomly generated by malware. Observations used for training pBMDS are heavily biased on known user behaviors instead of malware signatures. Fig.10 shows the test result when we used a trained pBMDS engine to detect major cellphone malware, including Cabir [19], CommWarrior [27], and Lasco [16] and their variants. Among the test cases, 59 belong to SMS compromised applications and 41 test sequences are selected from normal human operations. The figure shows that the pBMDS engine has an overall good performance (accuracy above 95%) in detecting these cellphone malware. As the number of process states defined for HMM in pBMDS increases, detection accuracy also gets improved. Also, the performance of each pBMDS engine is closely related with a detection threshold τ. By adjusting τ, pBMDS achieves an optimal detection accuracy, while keeping false positives low. We note that here false negatives have been reduced to a relatively low level using approaches we described in Section 4.3.

Table 2 shows a trade-off between detection accuracy and detection time. Using SMS messaging as an example, when the number of process states defined for HMM in pBMDS increases, detection accuracy of pBMDS is improved. However, this incurs a longer training and detection process. In our test, we used OMAP board (operating at 192MHZ, 32 MByte RAM) as the test platform to represent existing hardware configuration of smartphones in the market. We found out that 3-second training time and 130-millisecond detection time are acceptable to users. Hence, we believe choosing 35 as the number of process states for SMS applications nicely deals with the trade-off between detection time and detection accuracy.

Table 2: Time performance (in microsecond) of pBMDS engine on OMAP board; SMS application

#HMM states	Learning time	Detection time
80	41,731,674	651,967
60	10,144,915	382,182
35	3,263,524	130,559
25	1,242,360	71,700
15	324,845	33,679

Table 3: pBMDS on different cellphone applications (training size = 125)

	SMS	MMS	Email	Bluetooth
#HMM states	25	40	35	35
Detection rate	92.1%	96.4%	95.2%	94.9%
False Positive	6.3%	2.8%	3.7%	4.5%
False Negative	1.6%	0.8%	1.1%	0.6%

Table 3 shows the result when applying pBMDS to different cellphone applications. An HMM model is built for each application that is the possible target of malware attacks. We can see that the more complex a cellphone application is, the more process states are needed for configuring the HMM model. For example, MMS requires the largest number of process states because it involves more user-machine interactions and more internal program processing than other applications in the table. Note that here we only use plain-text email service (without attachment) as the example. Again, the time-and-accuracy tradeoff needs to be considered. Our result shows that pBMDS works well in securing various cellphone applications.

pBMDS with User Operational Preferences The above results have shown that pBMDS is effective in detecting malicious process behaviors. However, considering the fact that malware are becoming more and more intelligent, in some extreme cases, they could even successfully simulate all normal keypad and touch-screen input events on certain applications, such that a detection engine

Table 4: **Profiling user operational pattern (16 features)**

Training	Testing	Model	Accu	False pos
44 (A)	37 (A)	A-B,C,M	97.3%	2.7%
35 (B)	20 (B)	B-A,C,M	95.8%	4.2%
12 (C)	8 (C)	C-A,B,M	83.1%	16.9%
15 (M)	13 (M)	M-A,B,C	94.5%	5.5%

trained only on known process state transitions is not enough to defeat them. To address this challenge, we add users' operational patterns to further secure cellphone applications. In our experiments, we chose 3 different users and collected their operational patterns on SMS usages. These patterns include 16 different user-specific features which can be extracted from the logging data.

- Keypad input: number of keypad touches (k_t), speed of keypad touches (p_k), and key stroke holding time (k_h).
- Touch-screen input: number of touches on the screen (s_t), touch pressure (p_t), time duration per touch (d_t), and movement per touch (m_x, m_y).

These features are computed based on the mean value m and the standard deviation s_d. We get $8 * 2 = 16$ features in total:

$$(k_t + p_k + k_h + s_t + p_t + d_t + m_X + m_Y) * (m + sd)$$

Also, we improved the attack capability of CommWarrior [27] and let it simulate all required human keypad and touch-screen inputs during the messaging process. However, it does not have the knowledge of private users' operational patterns. We tested whether the operational features of malware distinctively deviate from that of human operations on the same application. In addition, we tested whether specific users follow their own operational patterns, i.e., whether user profiles are useful in characterizing certain users. Table 4 shows the test result. We can see that, given enough training samples (e.g., user A and B each provides 44 and 35, respectively), a one-class SVM [13] successfully identifies (with high accuracy above 95%) specific users from others. However, because user C does not provide enough training samples, identifying C becomes relatively more difficult (accuracy around 83%) than identifying A or B. Another important result is that, even without many training samples from cellphone malware M, it is still very easy (accuracy around 94%) to differentiate it from human users because of their behavior difference in cellphone operations.

Table 5: **Combining process behavior with operational patterns (time-based)**

App	Mode	Accu	Fp	Mode	Accu	Fp
SMS-25	P	92.1%	7.9%	P+U	97.6%	2.4%
SMS-35	P	94.5%	5.5%	P+U	**99.3%**	0.6%
MMS-30	P	93.1%	6.9%	P+U	94.8%	5.2%
MMS-40	P	96.4%	3.6%	P+U	**98.9%**	1.1%
Email-20	P	90.6%	9.4%	P+U	91.7%	8.3%
Email-35	P	95.2%	4.8%	P+U	**98.2%**	1.8%

Table 5 shows the result when we combined process transitions and user operational patterns in pBMDS to detect malware attacking different cellphone applications. We used Cabir [19], CommWarrior [27], and Lasco [16] to generate attacks on the OMAP board. We also introduced the strategy of more elaborated malware we have mentioned in Section 5.1 to test the effectiveness of our countermeasure. To reduce overhead, we did not install an additional classifier (SVM) in the resource-constraint smartphone to learn user operational patterns. Instead, we chose the time feature associated with the user operations and incorporated it into the HMM model (see Section 4.2). In this way, the pBMDS detection engine is more light-weighted and fits the resource-constraint cellphone environment. Our result demonstrates that a combination of both process state transitions (P) and user operational patterns (U) helps improve detection accuracy ($\sim 99\%$) of pBMDS and reduce detection errors. In addition, it provides two-level protection against malware. In the case when a malware simulates process states to surpass a detection engine, user operational patterns can be used by pBMDS to effectively identify the attack. Also note that the number of process states in the HMM model remains to be an important factor in improving detection accuracy.

6. DISCUSSION

In this paper, we use the SMS, MMS, and email applications as the examples to illustrate the mechanism and the effectiveness of pBMDS. These services have been reported [7, 8] as the most popular and vulnerable applications that are under severe malware attacks in the current stage of smartphones. We have also given details on key system calls and possible monitoring points in Table 1 for WiFi, Bluetooth, phone book, and web browser in smartphone frameworks such as OpenMoko and Qtopia. The most important thing here is that our examples represent typical application behaviors to access critical system resources on smartphones to achieve their functional goals such as communications or entertainments.

Certainly, pBMDS is not going to solve all problems. Indeed, given the great variety of cellphone platforms and the sophistication of attacks, we do not think any single or a few defense techniques will be sufficient. As more functions are integrated into smartphones, the complexity of applications on smartphones also increases. However, we believe that even though increasing complexity of mobile applications can appear, their user interfaces are still much simpler than those of desktop applications [9]. Based on our development experience on Android, iPhone, and Qtopia platforms, application menus usually have fixed items and layout (e.g., for Android), and each window has much fewer UI components (e.g., buttons and select list) than that in Windows and Linux desktop applications. This is partially because many smartphones use touch screen, so UI design has to consider good user experience such as single-hand operation without stylus [25]. On the other side, according to our pBMDS approach, new application profiles are relatively easy to get because mobile phone vendors already know deterministic application behaviors and they can easily generate the profiles before releasing the product to the market. User operational profile has to rely on capturing enough user behavior data to build an accurate detecting engine. However, it does not rely much on new applications. One limitation of pBMDS is that user behavior data have to be completely collected after some time, which means pBMDS needs time to build a user's "signature". pBMDS can generate false alarms for some designated automated programs in the phone, e.g., sending messages or videos by itself. We believe that it can be explicitly specified by the user to allow such dedicated automated programs, i.e., we just allow exceptions in this case.

Other challenges come from the fact that mobile handsets are getting more and more flexible and diversified in their input methods. For example, nowadays QWERTY keyboard is becoming popular in smartphones. This will require that more transition states be defined. Furthermore, malware are becoming more and more intelligent. Our on-going research is trying to capture more attacks and devise a more effective version of our pBMDS malware detection engine to defend against them.

7. RELATED WORK

Computer viruses have been plaguing the Internet for years, and a number of detection and defense mechanisms have been proposed. However, only recently have cellphone malware attracted considerable attentions in research community. The initial studies on cellphone malware [22, 33, 30, 29] mainly focused on understanding the threats and behaviors of emerging malware. For example, Guo et al. [22] examined various types of attacks that can be launched to a compromised cellphone, and suggested potential defenses. Radmilo et al. [33] revealed the vulnerability of MMS/SMS, which can be exploited to launch attacks on battery exhaustion. Mulliner et al. [30] demonstrated a proof-of-concept malware which crosses service boundaries in Windows CE phones. They also revealed buffer overflow vulnerabilities in MMS [29].

To solve the problem of quarantining cellphone malware outbreaks, Bose et al. [14] proposed an algorithm to automatically identify vulnerable users based on interactions and a proactive containment framework to quarantine suspected users. Cheng et al. [15] designed a collaborative virus detection system named SmartSiren to secure smartphones. Smartsiren collects communication data from phones and performs joint analysis to detect abnormal phone behaviors and alert users. These solutions are network-based (i.e., they detect anomalies through analyzing user messaging logs on centralized servers to detect anomalies). Our work differs from these solutions in that we place our defense at the frontier towards the malware - system level to identify malware and block unauthorized communications at an earliest stage. Therefore, our defense is more real-time than network-based schemes.

At the device level, Mulliner et al. [30] adopted a labelling technique to protect the phone interface against malware attacks coming through the phone's PDA interface. Specifically, resources and codes are labeled based on the interfaces that they come from. A process can access a resource or invoke a code only when it has been labeled with the same label as the resource or code; or, if it is not labeled, it is labeled with the same label of the resource or code and then gets the access. Any process or resource created by a process is labeled with the same label as the creating process. However, their approach is access-control based and it cannot defeat more intelligent malware as we mentioned in Section 5.

System call trace has long been adopted to detect malware in PC platforms [21, 26]. Forrest et al. [20] introduced a simple anomaly detection method based on monitoring the system calls issued by active, privileged processes. Lee et al. [28] established a more concise and effective anomaly detection model by using Ripper to mine normal and abnormal patterns from the system call sequences. Wespi et al. [37] further developed Forrest's idea and proposed a variable-length approach. Warrender et al. [35] proposed a Hidden Markov Models (HMM) based method for modeling and evaluating invisible events. These methods, however, have never been applied to the battlefield of mobile phones where they can show their better strength (see Section 3) against malware. Moreover, our correlation of user inputs with system call trace further reinforce its applicability and effectiveness in the mobile platform.

8. CONCLUSION

We foresee security attacks in cellphones will become smarter and devastating as more people are switching to smartphones which resemble years-old PCs. Existing signature-based approaches including security updates are neither realtime nor independent on users' awareness. Designing human intelligence-based defenses to differentiate malware from human beings hence becomes one of the most promising solutions for smartphones. In this work, we study malware behavior and focus on the behavior differences between them and propose a system-level countermeasure. We have showed through extensive smartphone experiments that our defense is effective, light-weight, and easy to deploy. In spite of some remaining issues such as diversity of platform OS and potential kernel-level attacks, our solution provides a practical way for containing existing or even future malware which could be more elaborated and intelligent. These remaining issues are our future work.

Acknowledgement We thank the reviewers for their valuable comments. This work of Liang Xie and Sencun Zhu was supported by CAREER NSF-0643906.

9. REFERENCES

[1] http://en.wikipedia.org/wiki/cross_validation.
[2] http://trolltech.com/products/qtopia.
[3] http://www.elinux.org/osk.
[4] http://www.f-secure.com/v-descs/flexispy_a.shtml.
[5] http://www.us-cert.gov/press_room/trendsandanalysisq108.pdf.
[6] http://www.virtuallogix.com/.
[7] Mcafee mobile security report 2008, mcafee.com/us/research/mobile_security_report_2008.html.
[8] Mcafee mobile security report 2009, mcafee.com/us/local_content/reports/mobile_security_report_2009.pdf.
[9] Mobile device ui design, http://blueflavor.com/blog/2006/apr/04/mobile-device-ui-design/.
[10] OpenMoko. http://wiki.openmoko.org.
[11] TCG mobile reference architecture specification version 1.0. https://www.trustedcomputinggroup.org/specs/mobilephone.
[12] Virtualization for embedded systems, http://www.ok-labs.com/.
[13] A. Bose and et al. Behavioral detection of malware on mobile handsets. In *Proc. of MobiSys*, 2008.
[14] A. Bose and K. Shin. Proactive security for mobile messaging networks. In *Proc. of WiSe*, 2006.
[15] J. Chen, S. Wongand, H. Yang, and S. Lu. Smartsiren: Virus detection and alert for smartphones. In *Proc. of MobiSys*, 2007.
[16] E. Chien. Security response: Symbos.lasco.a, symantec, 2005.
[17] E. Chien. Security response: Symbos.mabir, symantec, 2005.
[18] E. Chien. Security response: Symbos.skull, symantec, 2004.
[19] P. Ferrie, P. Szor, R. Stanev, and R. Mouritzen. Security response: Symbos.cabir. Symantec Corporation, 2004.
[20] S. Forrest, S. Hofmeyr, A. Somayaji, and T. longstaff. A sense of self for unix processes. In *Proc. of the IEEE Symposium in Security and Privacy*, 1996.
[21] S. Forrest and B. Pearlmutter. Detecting instructions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.
[22] C. Guo, H. Wang, and W. Zhu. Smartphone attacks and defenses. In *HotNets-III, UCSD*, Nov. 2004.
[23] C. Heath. Symbian os platform security. In *Symbian Press*, 2006.
[24] M. Hypponen. State of cell phone malware in 2007, http://www.usenix.org/events/sec07/tech/hypponen.pdf.
[25] A. K. Karlson and B. B. Bederson. One-handed touchscreen input for legacy applications. In *Proc. of CHI*, pages 1399–1408, 2008.
[26] E. Kirda and et al. Behavior-based spyware detection. In *Proc of USENIX Security Symposium*, 2006.

[27] M. Lactaotao. Security information: Virus encyclopedia: Symbos_comwar.a: Technical details. Trend Micro Inc., 2005.

[28] W. Lee, S. Stolfo, and P. Chan. Learning patterns from unix process execution traces for intrustion detection. In *Proc. of AAAI*, 1997.

[29] C. Mulliner and G. Vigna. Vulnerability analysis of mms user agents. In *Proc. of ACM ACSAC*, 2006.

[30] C. Mulliner, G. Vigna, D. Dagon, and W. Lee. Using labeling to prevent cross-service attacks against smartphones. In *DIMVA*, 2006.

[31] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *Proc. of the 13th ACM Symposium on Access Control Models and Technologies*, 2008.

[32] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proc. of the IEEE*, 1989.

[33] R. Racic, D. Ma, and H. Chen. Exploiting mms vulnerabilities to stealthily exhause mobile phone's battery. In *IEEE SecureComm*, 2006.

[34] R. Sailer, X. Zhao, T. Jaeger, and L. Doom. Design and implementation of a tcg-based integrity measurement architecture. In *Proc. of Usenix Security Symposium*, 2004.

[35] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium in Security and Privacy*, 1999.

[36] L. Welch. The shannon lecture: Hidden markov models and the baum-welch algorithm. In *IEEE Information Theory Society Newsletter*, 2003.

[37] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable length audit trail patterns. In *Proc. of RAID*, 2000.

[38] L. Xie, H. Song, T. Jaeger, and S. Zhu. Towards a systematic approach for cell-phone worm containment. In *Proc. of International World Wide Web Conference (WWW), poster*, 2008.

[39] L. Xie, X. Zhang, A. Chaugule, T. Jaeger, and S. Zhu. Designing System-level Defenses against Cellphone Malware. In *Proc. of 28th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2009.

[40] D. Yeung and Y. Ding. Host-based intrustion detection using dynamic and static behavioral models. In *Pattern Recognition, Issue.1*, 2003.

[41] X. Zhang, O. Aciicmez, and J. Seifert. A trusted mobile phone reference architecture via secure kernel. In *ACM workshop on Scalable trusted computing*, 2007.

APPENDIX

A. PROFILING USER OPERATIONAL PREFERENCE

We have the following notation.

T = *length of the sequence of observations (behavior graph)*
N = *number of process states in the model*
M = *number of possible key observations*
$S = \{s_1, s_2, ..., s_N\}$: *finite set of possible process states*
$V = \{v_1, v_2, ..., v_M\}$: *finite set of possible key observations*
$A = \{a_{i,j}\}$: $N \times N$ *matrix*, $a_{ij} = P(q_{t+1} = s_j | q_t = s_i)$ *is the probability of making a transition from state* s_i *to* s_j
$B = \{b_j^t(k)\}$: $N \times T \times M$ *matrix*, $b_j^t(k) = P(O_t = v_k | q_t = s_j, t)$ *is the probability of emiting* v_k *at time index* t *at state* s_j

Note that here we encode temporal information which reflects user operational preferences into key observations. Now we have the output probability of a sequence of key observations:

$$P(O_1, O_2, O_3, ..., O_T | q_j = s_j) = \prod_{t=1}^{T} P(O_t | q_j = s_j, t) \quad (2)$$

Let $\xi_t(i,j)$ denote the probability being in state s_i at time t and the state s_j at time $t+1$, we derive an extended version of Baum-Welch algorithm [36]:

$$\begin{aligned} \xi_t(i,j) &= P(q_t = s_i, q_{t+1} = s_j | O, \lambda) \\ &= \frac{\alpha_t(i) a_{ij} b_j^{t+1}(O_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t(i) a_{ij} b_j^{t+1}(O_{t+1}) \beta_{t+1}(j)} \end{aligned} \quad (3)$$

Forward variable $\alpha_t(i) = P(O_1...O_t, q_t = s_i | \lambda)$ is defined as the probability that the model is in state s_i at time t and has generated observations up to step t. Backward variable $\beta_t(j)$ is analogously defined to be the probability that the model is in state s_i at time t and will generate the remainder of the given target observations. Using the EM approach, a new model $\bar{\lambda} = (\bar{A}, \bar{B}, \bar{\pi})$ can be re-estimated using the following equations

$$\bar{\pi}_i = \sum_{j=1}^{N} \xi_1(i,j), \quad \bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_t(i,j)}, \quad (4)$$

and

$$\bar{b}_j^t(k) = \frac{\sum_{t=1}^{T} (\delta(O_t, v_k) \sum_{j=1}^{N} \xi_t(i,j))}{\sum_{t=1}^{T} \sum_{j=1}^{N} \xi_t(i,j)}, \quad (5)$$

where $\delta(O_t, v_k) = 1$, if $O_t = v_k$, and 0 otherwise.

In each round of behavior learning, we replace λ with the estimated $\bar{\lambda}$ based on the training set of behavior graphs. Clearly, we have $P(O|\bar{\lambda}) > P(O|\lambda)$ [32], which means λ converges to the actual model. We obtain a final λ when some convergence criterion is met (e.g., sufficiently small change in the estimated values of the parameters on subsequent iterations).

B. HMM-BASED MALWARE DETECTION

For simplicity, we still use time duration as the user operational feature. The forward variable $\alpha_t(i)$ is defined as

$$\alpha_t(i) = P(O_1...O_t, q_t = s_i | \lambda), \ \ \lambda = (A, B, \pi), \quad (6)$$

i.e., the probability of the partial observation sequence until time t (denoted as $O_1, O_2, ...O_t$) and the process state s_i at time t, given the model λ. We solve for $\alpha_t(i)$ inductively, as follows:

1) Initialization:

$$\alpha_1(i) = \pi_i b^1{}_i(O_1), \quad 1 \le i \le N \quad (7)$$

2) Induction:

$$\alpha_{t+1}(j) = [\sum_{i=1}^{N} \alpha_t(i) a_{ij}] b_j^{t+1}(O_{t+1}), \qquad 1 \le j \le N$$
$$1 \le t \le T-1 \quad (8)$$

3) Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i) \quad (9)$$