# Behavioral Detection of Malicious Programs on Mobile Handsets

Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park
The University of Michigan

## ABSTRACT

The rapidly growing capability and world-wide proliferation of smart phones and mobile handhelds have begun to attract the attention of virus writers in recent years. The past three years alone have witnessed an exponential rise in the number of distinct mobile malware families to over 30, and their variants to more than 170. These malware can spread via Bluetooth and SMS/MMS messages, enable remote control of a device, modify critical system files, damage existing applications including anti-virus programs, and block MMC memory cards, to name a few. Current-generation mobile anti-virus solutions are primitive when compared to their desktop counterparts, and may not be scalable given the small footprint of mobile devices as new families of cross-platform malware continue to appear.

This paper proposes a novel behavioral detection framework to capture mobile worms, viruses and Trojans, instead of the signature-based solutions currently available for mobile devices. First, we generate a database of malicious behavior signatures by studying over 25 distinct families of mobile viruses and worms targeting the Symbian OS, including their 140 variants, reported to date. Next, we describe a two-stage mapping technique that constructs these signatures at run-time from monitoring the system events and API calls in Symbian OS. We discriminate malicious behavior of malware from normal behavior of applications by training a classifier based on Support Vector Machines (SVMs). Our evaluation results indicate that behavioral detection can identify current mobile viruses and worms with over 96% accuracy. We also find that the time and resource overheads of constructing the behavior signatures from low-level API calls are acceptably low for practical deployment. Most mobile device manufacturers and mobile service providers can implement our proposed framework without any major modification of the handset operating environment.

## 1. INTRODUCTION

Mobile handsets are increasingly used to access services such as messaging, video/music sharing, and e-commerce transactions that have been previously available on PCs and servers only. However, with this new capability of handsets, there comes an increased risk and exposure to malicious programs (e.g., spyware, Trojans, mobile viruses and worms) seeking to compromise data confidentiality, integrity and availability of handset services. The handset manufacturers have responded to the increasing threat to mobile devices in several ways. For example, the Symbian Signed [1] framework derives a unique application certificate from the Symbian Root certificate issued by its Certificate Authority (CA) for signing an application. When a signed application is installed, the Symbian installer program verifies that the signature is valid before proceeding with the installation. This ensures that the software has not been tampered with during its distribution and has undergone a standard testing procedure as part of being Symbian Signed. However, given the vast number of mobile applications available on the Internet, especially peer-to-peer sites, one can not expect all applications to be signed with a certificate.[1] Note that an application that has been self-signed cannot be trusted to be free of malicious code. Moreover, even when an application is signed by a trusted CA, a malicious program can still enter the system via downloads (e.g., SMS/MMS messages with multimedia attachments), and it may exploit known vulnerabilities of an unsigned helper application.

The most prevalent approach for securing handsets is to detect malicious programs at their arrival via anti-virus software. A number of handset manufacturers and network operators have partnered with security software vendors to offer anti-virus programs for mobile devices [2, 3]. However, current anti-virus solutions for mobile devices are not as sophisticated as their counterparts in desktop environments, and rely primarily on signature-based detection. As a result, these tools are mostly useful for post-infection cleanup. For example, if a handset is infected with a mobile virus, these tools can be used to scan the *system* directory for the presence of files with specific extensions (e.g., .APP, .RSC and .MDL in Symbian-based devices) and filenames typical of virus payload. Although the infected files are deleted by the anti-virus tool, the underlying vulnerability — *overwriting the system directory* — is *not* patched. As a result, a cleaned handset may get infected again by another instance of the same virus, requiring repeated cleanup. Moreover, due to

[1]Currently, very few operators lock down handsets to prevent users from installing unsigned applications.

their limited CPU, memory and storage resources, signature-based detection will not be viable for mobile devices in future if the threats continue to grow at a fast rate. The emergence of crossover worms and viruses [4] that infect a handset when it is connected to a desktop for synchronization (and vice versa) requires that mobile applications and data be checked against both traditional as well as mobile virus signatures. This limits the extent to which signature-based schemes can be deployed on handsets.

Most of the published studies [5, 6, 7, 8] on the detection of Internet malware have focused on their network signatures (i.e., traffic generated due to scanning, failed connection attempts, and DNS server accesses). Constructing network signatures of mobile malware is extremely difficult due to the mobility of devices and the relatively closed nature of cellular networks. There is very little published work on the detection of mobile malware which can spread via non-traditional vectors, such as Bluetooth, content like games and video clips, and SMS/MMS messaging [9, 10]. Compared to traditional OSs, Symbian and other mobile OSs have important differences in the way file permissions and modifications to the operating system are handled. Therefore, development of a detection framework that overcomes the limitations of signature-based methods, while addressing unique features and limitations of the mobile operating system environment, is an important area of research.

An alternative to signature-based methods, *behavioral detection* [11], has emerged as a promising technique for preventing the intrusion of spyware, viruses and worms. In behavioral detection, the run-time behavior of an application (e.g., its memory and file accesses, API calls) is monitored and compared against a set of malicious behavior profiles. The malicious behavior profiles can be specified as global rules that apply to all applications, as well as fine-grained application-specific rules. Behavioral detection can detect polymorphic (malware that change their payload signatures) [12] and zero-day (malware that exploit a previously-unknown vulnerability) [13] worms since it does not rely on payload signatures. Also, a typical database of behavior profiles and rules should be smaller than that needed for storing specific payload signatures of many different classes of malware. This makes behavioral detection methods particularly suitable for handsets. However, deploying behavioral detection poses two challenges. The first is *specification* of what constitutes either normal or malicious behavior that covers a wide range of applications, while keeping the number of false positives for malicious behavior detection low. The second is *on-line reconstruction* of potentially suspicious behavior from the run-time behavior of applications, so that the observed signatures can be matched against a database of malicious signatures.

The primary contribution of our work is to overcome these two challenges for the mobile operating environment so that behavioral detection can be deployed to identify mobile malware. Our approach attempts to address the shortcomings of current-generation mobile anti-virus tools and can be deployed in handsets without any modification of the mobile operating system. The starting point of our approach is to generate a catalog of malicious behavior signatures by examining the behavior of current-generation mobile viruses, worms and Trojans that have been reported in the wild so far. We specify malicious behavior as a collection of system calls and resource access attempts made by these malicious programs, interposed by a temporal logic called the *temporal logic of causal knowledge* (TLCK). Monitoring system call events and file accesses have been used successfully in intrusion detection [14, 15] and backtracking [16]. In our approach, we reconstruct the higher-level malicious behavior signatures on-line from lower-level system calls and file accesses similar to how individual pieces are put together to form a jigsaw puzzle. The TLCK-based behavior specification addresses the first challenge of behavioral detection, by providing a compact "spatial-temporal" representation of malicious behavior. The next step is fast and accurate reconstruction of these malicious signatures during run-time by monitoring system calls and resource accesses so that appropriate alerts can be generated. This overcomes the second challenge for deployment of behavioral detection in mobile handsets. In order to detect malicious programs from their partial or incomplete behavior signatures, we train a classifier based on *Support Vector Machines* (SVMs) [17, 18] so that partial signatures for malicious behavior can be classified from those of normal applications running on the handset. For real-life deployment, the resulting SVM model and the malicious signature database are preloaded onto the handset by either the handset manufacturer or a cellular service provider. These are updated only when new behaviors (i.e., not minor variants of current malware) are discovered. The updating process is similar to how anti-virus signatures are updated by security vendors. However, since new behaviors are far fewer than new variants, the updates are not expected to be frequent.

The paper is organized as follows. Section 2 describes how to construct behavior signatures using the TLCK logic and shows examples from current-generation mobile malware. These signatures are generated based on our extensive survey of mobile viruses, worms and Trojans discovered to date. We also describe generalized behavior signatures that cover broad categories of malware so that a compact malicious behavior database can be created. In Section 3, we describe the implementation of a monitoring layer in Symbian that constructs these signatures from captured API calls and system events via a two-stage mapping technique. Section 4 discusses a class of machine learning algorithms called *Support Vector Classification* (SVC) that we use to tell malicious behavior from normal behavior based on captured partial signatures. This step is necessary in order to detect a malware before its complete behavior signature can be captured by the monitoring layer. We evaluate the effectiveness of behavioral detection in Section 5 by first training the SVM

classifier and then testing it against 5 current-generation mobile viruses and worms. We review the related literature in Section 6 and make concluding remarks in Section 7.

# 2. MALICIOUS BEHAVIOR SIGNATURES

## 2.1 Temporal patterns

We define ***behavior signature*** as the resulting manifestation of a specification of resource accesses and events generated by applications, including malware. We are interested in only those behavior signatures that indicate the presence of a malicious activity such as damage to the handset operating environment (e.g., draining the battery or overwriting files in the system directory), installing a rootkit or worm payload, sending out an infected message, etc. To this end, it is not sufficient to monitor a single event (e.g., a file read or write access) of a process in isolation in order to classify an activity to be malicious. In fact, there are many steps a malicious worm or virus performs in the course of its life-cycle that may appear to be harmless when analyzed in isolation. However, a logical ordering of these steps over time often clearly reveals the malicious intent. The *temporal pattern*—i.e., the precedence order of these events and resource accesses in time—is therefore key to detecting such malicious intent. For example, consider a simple file transfer by calling the Bluetooth OBEX system call (e.g., *CObexClient::Put()* and related OBEX operations) in Symbian. This is often used by applications for exchanging data such as games and music files among nearby handsets. On their own, any such calls will appear to be harmless. However, when the received file is of type *.SIS* (Symbian installation file) *and* that file is later executed, *and* the installer process seeks to overwrite files in the *system* directory, we can say with a high degree of certainty that the handset has been infected by a virus such as Mabir [19] or Commwarrior [20]. With subsequent monitoring of the files and processes touched by the above activities, the confidence level of detection can be improved further. This means that if we view the handset as a system exhibiting a wide range of behaviors over time, we can classify some of the temporal manifestations of these behaviors as malicious. Note that the realization of specific behaviors is dependent on how a user interacts with the handset and specific infection vectors of a malware. However, the *specification* of temporal manifestation of malicious behaviors can still be prescribed *a priori* by considering their effect on the handset resources and the operating environment.

A simple representation of malicious behavior can be given by ordering the corresponding actions using a vector clock [21] and applying the "and" operator to the actions. However, for more complex behavior that requires complicated temporal relationships among actions performed by different processes, simple temporal representations may not be sufficient. This suggests that behavior signatures are best specified using temporal logic instead of classical propositional logic. Propositional logic supports reasoning with statements that evaluate to be either true or false. On the other hand, temporal logic allows propositions whose evaluation depends on time, making it suitable for describing sequences of events and properties of correlated behaviors. There have been significant research in applying temporal logic to study distributed systems, and software programs. There are also various branches of temporal logic such as linear time and branching time logic [22]. In Linear Time Temporal Logic (LTTL), program execution behavior can be modeled as a linear sequence of states, where each state has a fixed (i.e., deterministic) output. On the other hand, the Branching Time Temporal Logic (BTTL) allows state transitions via a finite (or infinite) number of reachable states where the states can be seen to form a reachability tree. Since program execution and file/memory accesses are not always linear in time, LTTL is not a suitable choice for our purpose. A variant of BTTL called the *temporal logic of causal knowledge* (TLCK) [23], on the other hand, allows concurrency relations on branching structures that are naturally suitable for describing actions of multiple programs. Therefore, we adopt the specification language of TLCK to represent malicious behaviors within the context of a handset operating environment.

## 2.2 Temporal Logic of Malicious Behavior

This section describes how to specify malicious behavior in terms of system calls and events, interposed by temporal and logical operators. The specification of malicious behavior is the first step of any behavioral detection framework. Although our presentation is primarily targeted to the Symbian OS, it can be extended for other mobile operating systems as well.

First, let us formally define a behavior signature as a finite set of propositional variables interposed using TLCK, where each variable (when true) confirms the execution of either (i) a single or an aggregation of system calls, or (ii) an event such as read/write access to a given file descriptor, directory structure or memory location. Note that we do *not* keep track of all system calls and events generated by all processes — doing so will impose unacceptable performance overhead in constructing behavior signatures. Therefore, only those system calls and events that are used in the specification of malicious behavior are to be monitored. In fact, we find that specifying behavior signatures for the majority of mobile malicious programs reported to date, requires monitoring only a small subset of Symbian API calls.

Let $PS = \{p_1, p_2, \cdots, p_m\} \bigcup \{i | i \in N\}$ be a set of $m$ atomic propositional variables belonging to $N$ malicious behavior signatures. Atomic propositions can be joined together to form higher-level propositional variables in our specification. The logical operators *not* ($\neg$) and *and* ($\wedge$) are defined as usual. The temporal operators defined using past-time logic are as follows:

- $\odot_t$ *true at time t*
- $\diamondsuit_t$ *true at some instant before t*
- $\Box_t$ *true at all instants before t*

- $\Diamond_t^{t-k}$ true at some instant in the interval $[t-k,t]$.

The operator $\Diamond_t^{t-k}$ is a quantified temporal operator to range $k$ time instants over the time variable $t$. We make the following assumptions.

1. Time is represented by an infinite sequence of discrete time instants.
2. A duration is given by a sequence of time instants with initiating and terminating instants.
3. A system call or an event is instantiated at a given instant but may take place over a duration.
4. The strong synchrony hypothesis [24] holds for the handset operating system environment, i.e., the instantiation of a single event at a given instant can generate other events synchronously. In case of synchronous events, one can still use relative order to denote relationship among events.
5. Higher-level events and system calls of greater complexity can be composed by temporal and logical predications of the above atomic propositional variables.

To illustrate the application of the above logic, we apply it to specify the behavior of a family of mobile worms known as Commwarrior. Following this, we will specify behavior signatures that are general enough to cover different families of mobile worms. This generalization is a key benefit of using a behavioral detection approach as opposed to payload signatures, given the small memory and storage footprint of these devices.

## 2.3 Example: The Commwarrior worm

The Commwarrior worm [20] targets Symbian Series 60 phones and is capable of spreading via both Bluetooth and MMS messages. The worm payload is transferred via a SIS file with randomly-generated names. The payload consists of the main executable *commwarrior.exe* and a boot component *commrec.mdl* that are installed under `\System\updates`, `\System\Apps` and `\System\Recogs` directories. Figure 1 shows the organization of the Symbian filesystem. Each of the drive letters (*C:*, *D:*, *E:* and *Z:*) has an identical (but separate) filesystem tree rooted at *System*. Once the payload is installed, the SIS file installer automatically starts the worm process *commwarrior.exe*. It then rebuilds a SIS file from the above files and places it as `\System\updates\commw.sis`. Commwarrior spreads via Bluetooth by contacting all devices in range and by sending a copy of itself in a round-robin manner during the time window from 08:00 to 23:59 hours based on the device clock. It also spreads via MMS by randomly choosing a phone number from the device's phonebook, and sends an MMS message with *commw.sis* as an "*application/vnd.symbian.install*" MIME attachment so that the target device invokes the Symbian installer program upon receiving the message. The daily window for replication via MMS is only from 00:00 to 06:59 hours, again based on the device's own clock. Figure 2 presents a graphical representation of the behavior of the Commwarrior worm. Our

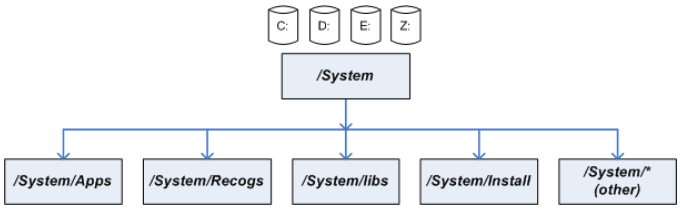


**Figure 1: Symbian filesystem directories targeted by malware (OS v8 and earlier)**

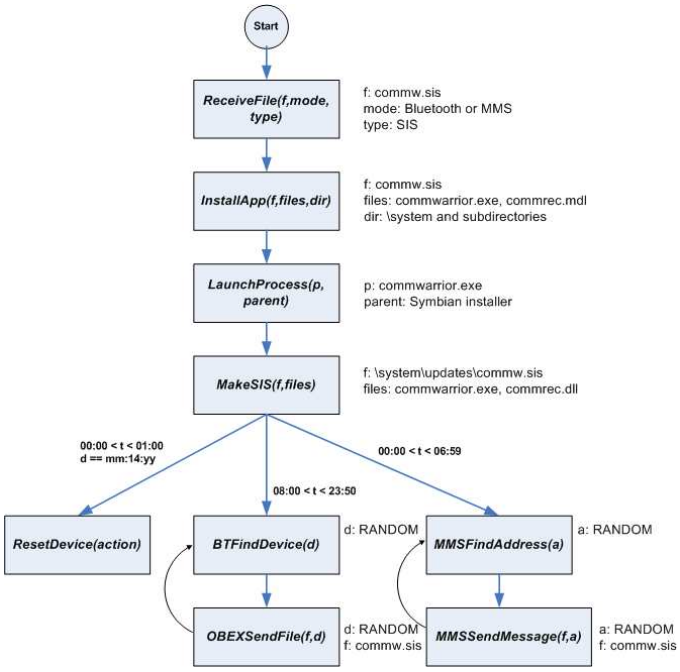


**Figure 2: Behavior signature for Commwarrior worm**

goal is to convert this graphical representation into a behavior signature using logical and temporal operators defined in Section 2.2.

Note that the specification of Commwarrior behavior requires monitoring of a small number of processes and system calls ($N = 6$), namely, the Symbian installer, the worm process (commwarrior.exe), two Symbian Bluetooth API calls and the native MMS messaging application on the handset. By generalizing the behavior signatures across many families of mobile malware, we hope to keep $N$ to be a small number. To specify Commwarrior in terms of TLCK logic, we first identify the set $PS$ of atomic propositional variables:

*ReceiveFile(f,mode,type):* Receive file $f$ via either mode= Bluetooth or mode=MMS of type SIS. When mode=MMS, the MIME attachment is of type *application/vnd.symbian.install*.
*InstallApp(f,files,dir):* Install a SIS archive file $f$ by extracting *files* and installing them in directory *dir* of the handset. The specific elements of $f$, $files$ and $dir$ are as shown in Figure 2.

*LaunchProcess(p,parent):* Launch an application *p* by a parent process *parent*, which is typically the Symbian installer.
*MakeSIS(f,files):* Create a SIS archive file $f$ from files $files$ (files are assumed to have fully-qualified path names).
*BTFindDevice(d):* Discover a random Bluetooth device *d* nearby.
*OBEXSendFile(f,d):* Transfer a file $f$ (with fully-qualified path name) to a nearby Bluetooth device *d* via the OBEX protocol.
*MMSFindAddress(a):* Look up a random phone number *a* in the device Phonebook.
*MMSSendMessage(f,a):* Send MMS message with attachment $f$ to a random phone number *a*.
*SetDevice(act,<condition>):* Perform action *act* (e.g., reset device) when $<condition>$ holds true. $<condition>$ is typically expressed as a set of other predicates to verify device time and date (see below).
*VerifyDayofMonth(date,<mm:dd>):* Verify if current date is $<mm:dd>$, e.g., "the 14th day of any month."

Next, we combine the atomic variables into 7 higher-level signatures that correspond to the major behavioral steps of the worm family. These seven signatures can be monitored during run-time and out of these seven, four signatures can be placed in our malicious behavior database to trigger an alarm. In particular, "$bt-transfer$" and "$mms-transfer$" are perfectly harmless signatures, where as "$activate-worm$", "$run-worm-1$", "$run-worm-2$" and "$run-worm-3$" can be used to warn the user, or trigger an appropriate preventive action, e.g. quarantine the outgoing message instead of sending it right away. Later, in Section 4, we show that the detection of malicious behavior can be made more accurately by training a SVM model.

- $\odot_t(bt-transfer) = \diamondsuit_t(BTFindDevice(d)) \wedge (\odot_t(OBEXSendFile(f,d)))$
- $\odot_t(mms-transfer) = \diamondsuit_t(MMSFindAddress(a)) \wedge (\odot_t(MMSSendMessage(f,a)))$
- $\odot_t(init-worm) = \odot_t(ReceiveFile(mode=Bluetooth)) \vee \odot_t(ReceiveFile(mode=MMS))$
- $\odot_t(activate-worm) = \diamondsuit_t(init-worm) \wedge (\odot_t(InstallApp) \wedge \odot_t(LaunchProcess))$
- $\odot_t(run-worm-1) = \diamondsuit_t(activate-worm) \wedge (\odot_t(MakeSIS) \wedge \odot_t(VerifyDayofMonth) \wedge (\diamondsuit_{1:00}^{0:00}(SetDevice)))$
- $\odot_t(run-worm-2) = \diamondsuit_t(activate-worm) \wedge (\odot_t(MakeSIS) \wedge (\diamondsuit_{23:59}^{8:00}(bt-transfer)))$
- $\odot_t(run-worm-3) = \diamondsuit_t(activate-worm) \wedge (\odot_t(MakeSIS) \wedge (\diamondsuit_{6:59}^{0:00}(mms-transfer)))$

## 2.4 Generalized Behavior Signatures

In order to create generalized signatures that are not specific to each variant of malware, we studied over *25* distinct families of mobile viruses and worms targeting the Symbian OS, including their *140* variants, reported to date. For each family of malware, we generated propositional variables corresponding to its actions, identified the argument lists for each variable and assigned TLCK operators to construct the behavior for the malware family. Then, we looked at these signatures across families of malware, and wherever possible, extracted the most common signature elements and recorded the Symbian API calls and applications that must be monitored to reconstruct a possible match. The result is a database of behavior signatures for malware targeting Symbian-powered devices reported to date that depends very little on specific payload names and byte sequences, but rather on the behavior sequences. We find that the malware actions can be naturally placed into three categories based on which layer of the handset environment the behavior manifests itself. The categorization identifies three points of insertion where malware detection and response agents can be placed in the mobile operating system.

(1) *User Data Integrity (UDI)*: These actions correspond to damaging the integrity of *user* data files on the device. Most common user data files are address and phone books, call and SMS logs, and mobile content such as video clips, songs, ringtones, etc. These files are commonly organized in the `\System\Apps` directory on the handset. The actions (and, in turn, propositional variables defined to express them) in this group, when true, confirm execution of system and API calls that open, read/write and close these data files.
Example: Acallno [25] is a commercial tool for monitoring SMS text messages to and from a target phone — the tool has been recently classified as a spyware by security software vendors. Acallno forwards all incoming and outgoing SMS messages on the designated phone to a pre-configured phone number. We define three UDI variables, *CopySMSToDraft(msg)*, *RemoveEntrySMSLog(msg)* and *ForwardSMSToNumber(msg,phone number)*, to represent the major tasks performed by Acallno. *CopySMSToDraft(msg)* copies the last SMS message *msg* received into a new SMS message in the Drafts folder. *RemoveEntrySMSLog(msg)* is true when the corresponding entry for *msg* is successfully deleted from the SMS log so that the user is not aware of the presence of Acallno. *ForwardSMSToNumber(msg,phone number)* is true when *msg* is forwarded to an external *phone number*. These three variables, when interposed with appropriate temporal logic, represent the behavior of "SMS spying" on a device. The UDI variable called "*InstallApp(f, files, dir)*" that we have already used earlier for Commwarrior has the following argument values for Acallno: f [SMSCatcher.SIS], files [s60calls.exe, s60system.exe, s60system1.exe, s60calls.mdl, s60sysp.mdl, s60syss.mdl] and dir [`\System\Apps`, `\System\recogs`]. These four UDI actions are present in all SMS spyware programs such as Acallno, MobiSpy and SMSSender, and the resulting *generalized* behavior signature can be used for their detection in place of their specific payload signatures.

(2) *System Data Integrity (SDI)*: Several malware attempt to damage the integrity of system configuration files and helper application data files by overwriting the original files in the Symbian system directory with corrupted versions. This is possible for two reasons: (i) the malware files are installed in flash RAM drive `C:` under Symbian with the same path as the operating system binaries in ROM drive `Z:`. The Symbian OS allows files in `C:` take precedence over files in `Z:` with the same name and pathname, and therefore, any file with the same path can be overwritten; and (ii) Symbian does not enforce basic security policies such as file permissions based on user and group IDs and access control lists. As a result, the user, by agreeing to install an infected SIS file, unknowingly allows the malware to modify the handset operating environment. The SDI actions (and the propositional variables) correspond to attempts to modify critical system and application files including files required at device startup.

Example: The actions of Skulls, Doomboot (or, SingleJump), AppDisabler, and their variants can be categorized under SDI. These malware overwrite and disable a wide range of applications running under Symbian, including InfraRed, File Manager, System Explorer, Antivirus (Simworks, F-Secure), and device drivers for camera, video recorders, etc. The target directories are, for example, `\System\Apps\IrApp\` and `\System\Apps\BtUi\` for Infra Red and Bluetooth control panels, respectively. Any file with the ".APP" extension in these directories is an application that is visible in the applications menu. If any of these files is overwritten with a corrupted version, the corresponding application is disabled. Since there are many application directories under `\System\Apps`, our goal is to monitor only those directories that contain critical system and application files such as fonts, file manager, device drivers, startup files, anti-virus, etc. We define the variable *ReplaceSystemAppDirectory(directory)* where *directory* is a canonical pathname of the target directory of a SIS archive.[2] The variable returns true when *directory* matches against a hash table of pre-compiled list of critical system and application directories. At this point, the installation process can be halted until the user permits to go ahead with the installation.

Another serious SDI action is deletion of subdirectories under `\System`. One of the actions performed by the Cardblock Trojan is deleting `bootdata, data, install, libs, mail` in `C:\System`. The `install` directory contains installation and uninstallation information for applications. Many Symbian applications log error codes in `C:\System\bootdata` when they generate a panic. Without these directories, most handset applications become unusable. As a general rule, no user application should be able to delete these directories. We, therefore, define a variable called *DRSystemDirectory(directory)* where *directory* checks against a hash table of these directories whenever a process attempts to either delete or rename a subdirectory under `C:\System`.

(3) *Trojan-like Actions*: This category of actions are performed by a malware when it is delivered to a device via either another malware ("dropper") or an infected memory card. These actions attempt to compromise the integrity of user and system data on the device (without requiring user prompts) by exploiting specific OS features and by masquerading as an otherwise useful program ("cracking"). Once a malware infects a device with Trojan-like actions, it may use UDI and SDI actions to alter the handset environment. To date, we find that there are two types of vectors for mobile Trojans: (i) memory cards and (ii) other malware. The memory cards used in cell phones are primarily Reduced-Size MultiMediaCard (RS-MMC) and micro/mini Secure Digital (SD) cards that can be secured using a password. As shown in Figure 1, the Symbian drive `E:` is used for memory cards with the same `\System` directory structure as of the other drives.

Example: The Cardblock Trojan mentioned earlier, is a cracked version of a legitimate Symbian application called InstantSis. InstantSis allows a user to create a SIS archive of any installed application and copy them to another device. Cardblock appears to have the same look and feel of InstantSis, except that when the user attempts to use the program, it blocks the MMC memory card and deletes the subdirectories in `C:\System` (SDI action). The Trojan-like action of Cardblock is the locking of the MMC card which it accomplishes by setting a random password to the card. Detection of Cardblock must be done either when it is first installed on the device or before it actually performs its two tasks (MMC blocking and deleting system directories). We define a variable called *SetPasswdtoMMC()* to capture the event that a process is attempting to set a password to the MMC card without prompting the user.

For lack of space, we do not provide a listing of behavior signatures organized by these three action categories. We refer to [29] for a complete listing of these signatures for mobile malware we have analyzed to date.

### *2.4.1 SDI Actions and Symbian OS V9*

In order to restrict applications from accessing the entire filesystem, Symbian has recently introduced *capabilities* beginning with Symbian OS v9 [1]. A capability is an access token that allows the token holder to access restricted system resources. In previous versions of Symbian OS, all user-level applications had read/write access to the entire filesystem, including `\System` and all its subdirectories. Therefore, malicious applications can easily overwrite or replace critical system files in all previous versions of Symbian, including OS v8. However, in the new Symbian platform security model, access to certain functions and APIs will be restricted by capabilities. In order to access the sensitive capabilities, an application must be "Symbian Signed" by

[2] When there are multiple target directories, *ReplaceSystemAppDirectory(directory)* is evaluated for each entry in the target list.

Symbian. In case of self-certified applications, the phone manufacturer must recommend the application developer for access to desired capabilities from Symbian. The three capabilities that can prevent many SDI actions currently performed by mobile malware are *AllFiles*, *TCB* (Trusted Computing Base) and *DiskAdmin*. Without these capabilities, an application will no longer be able to access the "/sys" directory where most of the critical system executables and data are stored. For example, it requires *AllFiles* capability to read from and *TCB* capability to write to "/sys". Most user applications in Symbian OS v9 are allowed to access a single directory called "/sys/bin" to install executables and create a private directory called "/private/SID" for temporary files, where SID refers to the Secure ID of the caller application, assigned when the application is Symbian Signed. There are also important changes in OS v9 regarding how an application is installed. The "`\System\Apps`" subdirectory previously used by applications for storing application information (resource files, bitmap files, helper application, etc.) is no longer supported. Instead, a separate filesystem path called "`\resource\apps`" is used for storing application information. By separating system and application data in different filesystems and by introducing capabilities for accessing sensitive system resources, Symbian OS v9 clearly improves the security model for mobile devices and will prevent a number of current-generation malware from damaging the integrity of the device. However, it may not prevent (i) mobile worms that spread via SMS/MMS or Bluetooth and social engineering techniques, (ii) malware from launching DoS attacks on other phones or communication infrastructure due to other vulnerabilities.

## 3. RUN-TIME CONSTRUCTION OF BEHAVIOR SIGNATURES

To build a malware detection system, the behavior signatures described in Section 2 must be constructed at runtime by monitoring the target set of system events and API calls. For the early generation of mobile handsets, building such a monitoring layer in the OS would cause unacceptably high performance overhead. However, in recent years, many embedded microprocessor vendors, especially ARM, have implemented features that allow real-time tracing of program instruction flow and data accesses. There are already a number of commercial tracing and debugging tools for ARM cores with an Embedded Trace Macrocell (ETM) unit, for resolving real-time application issues when traditional "halt-and-debug" methods cannot be used. In what follows, we describe the implementation of the monitoring layer in Symbian.

### 3.1 Monitoring of API calls via Proxy DLL

Since Symbian is a proprietary OS and provides neither kernel monitoring APIs nor system-wide hooks (e.g., Windows *message hooks* or Linux *netfilter hooks*), intercepting API calls is extremely difficult, if not impossible. Fortu-

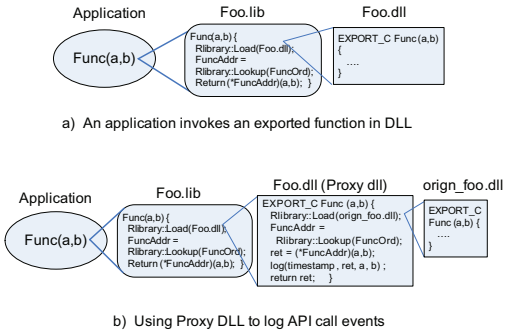


**Figure 3: Proxy DLL to capture API call arguments**

nately, the Symbian SDK is accompanied with a Symbian OS emulator which is a Windows application that accurately emulates almost all functionalities of a real handset, such as input devices, user interfaces and APIs for services (Bluetooth, SMS/MMS, filesystem accesses). Most Symbian-based handset developers, therefore, build and test mobile applications using the emulator before transferring them to the real handset. Moreover, the emulator implements all the Symbian APIs in the form of Dynamic Link Libraries (DLLs) which are loaded into memory at run-time. This is the feature that we were able to utilize to build our monitoring layer. Specifically, due to the dynamic load feature of the DLLs, the API traces of applications running in the emulator could be collected via a "*Proxy DLL*" shown in Figure 3. This is a popular technique used by many anti-virus tools written for Windows — they need the ability to hook into Winsock's I/O functions, e.g., to analyze data being transferred between email clients and mail servers for virus signatures.

Before delving into the details of *Proxy DLL*, we briefly discuss how DLLs work in the Windows OS. When a DLL is built, each function exported by the DLL is assigned a unique integer value known as its *ordinal number*. DLL functions are invoked at run-time by first loading the DLL library into memory, then looking up this ordinal in the DLL to find the memory address of the corresponding functions, and finally executing them. However, since the ordinal number is difficult to use and remember, programs using functions in the DLL often statically link to an *import library* (.lib). The role of the import library is to define the same set of functions as their counterparts in the DLL that are statically linked to the corresponding executables. In each function, the import library simply invokes its counterpart in the DLL file based on its ordinal number. Therefore, with the import library, we can invoke functions with more meaningful (and easy to remember) function names instead of their ordinal numbers.

Figure 3 shows an example of Proxy DLL that we implemented in the Symbian OS emulator to log our target API calls (e.g., func(a,b) exported by foo.dll). Figure 3(a) shows that without the Proxy DLL, when an application makes a function call func(a,b), the import library will load the corresponding DLL (i.e., foo.dll), search for the function address

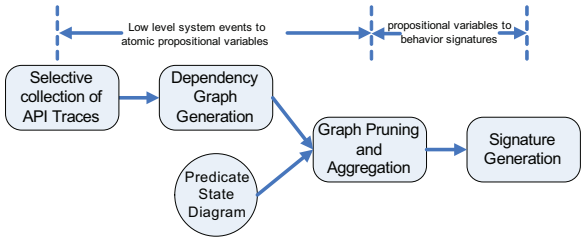


**Figure 4: Major components of the monitoring system**

and invoke the correct function. The DLL is loaded at runtime and transparently to user applications. Thus, we can replace the original DLL files with new Proxy DLLs instrumented with logging functionalities without any modification of both applications and import libraries. For instance, in Figure 3(b), the original foo.dll is replaced with a Proxy DLL with the same name and exported function (func(a,b)). When the import library (foo.lib) loads the DLL with the name foo.dll, the new Proxy DLL is loaded into the memory. After the application makes a function call func(a,b), the import library invokes the exported func(a,b) in the Proxy DLL which then loads the original DLL (origin_foo.dll) into the memory and runs the true func(a,b). Meanwhile, the Proxy DLL logs information about these API invocation events, including process ID, timestamp, parameters passed to the function and its return value.

Since we are not interested in logging every API call, the monitoring system was customized to log only those functions that can be exploited by mobile malware. In particular, only functions that constitute the atomic proposition variables described in Section 2 were entered in the Proxy DLL so that they can be logged. The number of function calls to be monitored may increase in future as new malware families emerge. However, the logging overhead is relatively low (600 microseconds) and acceptable. For microprocessors that allow real-time tracing, this overhead is expected to be minimal.

The rest of this section describes a two-stage mapping technique that we have used to construct the behavior signatures from the captured API calls. Figure 4 presents a schematic diagram of how low-level system events and API calls are first mapped to a sequence of atomic propositional variables (see Section 2.2), and then by graph pruning and aggregation, a set of behavior signatures. These two stages are elaborated next.

## 3.2 Stage I: Generation of Dependency Graph

Using the Proxy DLL, our monitoring agent logs a sequence of API calls invoked by all processes running in the system. The next step is to correlate these API calls using the TLCK logic described in Section 2.2, and build the behavior signatures (see Section 2). Note that the monitoring layer captures system-wide events and therefore API calls from different processes are intermingled with each other in the log. However, constructing behavior signatures requires application of TLCK logic to calls made by different processes. To efficiently represent the interactions among processes, we construct a dependency graph from logged API calls that effectively correlates different processes. This is achieved by applying the following rules to the captured API calls.

**Intra-process rule:** API calls that are invoked by the same process IDs are directly connected in the graph according to their temporal order. For example, in Figure 5, we represent the dependency graphs for two processes that generate two atomic propositional variables, *MakeSIS(f,files)* and *OBEXSendFile(f,d)*, respectively. The dependency graph for Process 2 (a set of API calls for sending files via Bluetooth) is an example of intra-process temporal ordering. Because all the functions had been called by a single process, they are connected with directed arrows indicating their temporal order. The result of this temporal ordering is the atomic propositional variable *OBEXSendFile(caribe.sis,d)* becoming true.

**Inter-process rule:** Since malware behavior signatures often involve multiple processes, we define two inter-process rules.

1. **Process-process relationship** where a process creates another process by forking and cloning within the context of a single application. In this case, the API calls become a new branch in the forked or cloned process.
2. **Process-file relationship** where a process creates, modifies or changes the attributes of a file, *and* the same file is read by another process. Establishing a chain of events from *process-file access* relationships is similar to the concept of backtracking [16], which identifies potential sequences of activities that occurred during an intrusion. We use a similar procedure to construct calling-process dependency relationships. Figure 5 shows an example of the inter-process dependency rule, where *Process 1*, *createsis* packages some files into a SIS archive file (caribe.sis), and subsequently, *Process 2* reads the file and sends it via Bluetooth. The result of this step is the construction of a larger signature: $MakeSIS(caribe.sis,..) \wedge OBEXSendFile(caribe.sis,..)$.

## 3.3 Stage II: Graph Pruning and Aggregation

Since every process has its own call-chain graph and may be connected to other processes via dependency links, the graph for system-wide process interactions could be very large. Note that propositional variables created from the monitoring log should be automatically assigned an expiration time so that one can discard as many unnecessary dependency graph elements as possible. A simple expiration policy is to destroy the call-chain graph of a process upon its

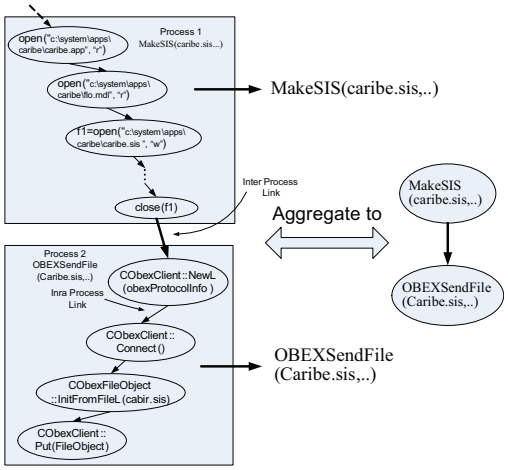


**Figure 5: Dependency graphs for constructing atomic propositional variables**

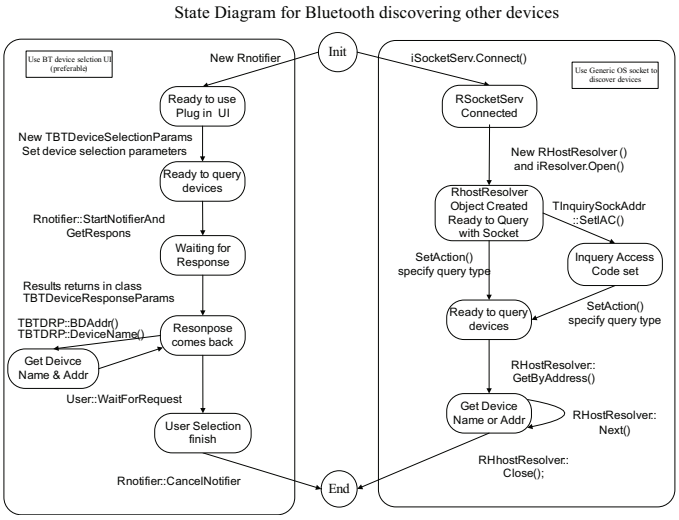


**Figure 6: State-transition diagram for signature FindDevice**

termination. However, this has an undesirable consequence because it will not allow building a future inter-process dependency graph with propositional variables generated by another process or application. This "information loss" can be exploited by a mobile malware by waiting for some time after each of its steps and avoiding detection by not letting its behavior signature to be completely built! To avoid such a scenario while still keeping memory requirements reasonable for generating behavior signatures, we implemented the following rules in the monitoring layer.

The dependency graph and propositional variables generated from API calls made by a process are discarded (upon its termination) if and only if:

1. The process didn't have inter-process dependency relationships with any other process (i.e., it is independent);
2. Its graph doesn't *partially* match with any behavioral signature that has inter-process dependencies;
3. It didn't create or modify any directory in the list of directories maintained in a hash table of critical user and system directories (see Section 2.4); and
4. It is a helper process that takes input from a process and returns data to the main process.

Since the dependency graphs can grow over time, we aggregate each API call sequence (e.g., Process 1 and Process 2 in Figure 5) as early as possible to reduce the size of the overall storage.

Finally, To construct a behavior signature by composing TLCK operators over the propositional variables, we use a state transition graph for each behavior signature, where the transition of each state is triggered by the invocation of one or more atomic propositional variables. The advantage of encoding each atomic variable into a state transition graph is that the monitoring system can easily validate the variable from operations performed in Stage I. A behavior signature is, therefore, constructed as a jig-saw puzzle by confirming a set of atomic propositional variables along its state transition graph until a terminal state is reached. This process of applying TLCK operators in the state diagram is shown in Figure 6 for the behavior signature *FindDevice*. It shows two parallel branches used to discover Bluetooth devices nearby, depending on which protocol is invoked by the application. State transitions along a branch are invoked by specific Symbian API calls.

The outcome of the two stages is a behavior signature that is to be classified either malicious or harmless by the detection system.

## 4. BEHAVIOR CLASSIFICATION

The behavior signatures for the complete life-cycle of a malware, such as those developed in Section 2, are placed in a malicious behavior database for run-time classification of signatures constructed via the two-stage mapping technique described above. However, if we wait until the complete behavior signature of a malware is constructed by the monitoring layer, it may be too late to prevent the malware from inflicting some damage to the handset. In order to activate early response mechanisms, our malicious behavior database must also contain partial signatures that have a high probability of manifesting as malicious behavior. These partial signatures (e.g., *bt-transfer*, *sms-transfer* and *init_worm* in Section 2.3) are directly constructed from the complete life-cycle malware signatures in the database. However, this introduces the problem of false-positives, i.e., partial signatures that may also represent the behavior of legitimate applications running on the handset, but may be falsely classified as malicious. Therefore, we need a mechanism to separate the partial (or incomplete) malicious behavior signatures from similar signatures of legitimate applications.

We use a learning method for classifying these partial be-

havior signatures from the training data of both normal and malicious applications. In this paper, we focus on the binary classification problem where the goal is to generate a function that can classify the input behavior signatures as belonging to either malicious (+1) or not (-1). In what follows, we describe a particular machine learning approach called *Support Vector Machines* (SVMs)that we implemented for the binary classification problem of partial behavior signatures.

## 4.1 Support Vector Machines

SVMs, based on the pioneering work of Vapnik [30] and Joachim [31] on statistical learning theory, have been successfully applied to a large number of classification problems, such as intrusion detection, gene expression analysis and machine diagnostics. SVMs address the problems of overfitting and capacity control associated with the classical learning machines such as neural networks. Traditional neural networks suffer from generalization, resulting in models that can overfit the training data. For a given learning task with a finite training set, the learning machine must strike a balance between the accuracy obtained on the given training set and the *capacity* of the machine which measures its ability to learn future unknown data without error. A machine with either high or low capacity may result in falsely classifying new observations. The flexible generalization ability of SVMs makes the approach suitable for real-world applications with a limited amount of training data. Here we refer to solving classification problems using SVMs as *Support Vector Classification* (SVC).

Let $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_m, y_m)$ denote $m$ observations (or the training set) of behavior signatures $\mathbf{x}$. Each behavior signature $\mathbf{x}_i$ is of dimension $d$ corresponding to the number of propositional variables, and $y_i = \pm 1$ is the corresponding class label (i.e., malicious or non-malicious) assigned to each observation $i$. We denote the space of input signatures (i.e., $\mathbf{x}_i$'s) as $\Theta$. Given this training data, we want to be able to *generalize* to new observations, i.e., given a new observation $\bar{\mathbf{x}} \in \Theta$, we would like to predict the corresponding $y \in \{\pm 1\}$. To do this, we need a function, $k(\mathbf{x}, \bar{\mathbf{x}})$, that can measure similarity (i.e., a real-valued scalar distance) between data points $\mathbf{x}$ and $\bar{\mathbf{x}}$ in $\Theta$:

$$k : \Theta \times \Theta \rightarrow \Re \tag{1}$$

$$(\mathbf{x}, \bar{\mathbf{x}}) \rightarrowtail k(\mathbf{x}, \bar{\mathbf{x}}). \tag{2}$$

The function $k$ is called a *kernel* and is most often represented as a canonical dot product. For example, given two behavior vectors $\mathbf{x}$ and $\bar{\mathbf{x}}$ of dimension d, the kernel $k$ can be represented as

$$k(\mathbf{x}.\bar{\mathbf{x}}) = \Sigma_{i=1}^{d} (\mathbf{x})_i . (\bar{\mathbf{x}})_i. \tag{3}$$

The dot-product representation of kernels allows geometrical interpretation of the behavior signatures in terms of angles, lengths and their distances. In fact, the dot product represents the cosine of the angle between vectors $\mathbf{x}$ and $\bar{\mathbf{x}}$ when their lengths are normalized to 1. A key step in SVM is mapping of the vectors $\mathbf{x}$ from their original input space $\Theta$ to a higher-dimensional dot-product space, $F$, called the *feature space*. This mapping is represented as $\Phi : \Theta \rightarrow F$. The mapping functions are chosen such that the similarity measure is preserved as a dot product in $F$:

$$k(\mathbf{x}, \bar{\mathbf{x}}) \rightarrow K(\mathbf{x}, \bar{\mathbf{x}})) := (\Phi(\mathbf{x}).\Phi(\bar{\mathbf{x}})) \tag{4}$$

There are many choices for the mapping functions in the feature space, such as polynomials, radial basis functions, multi-layer perceptron, splines and Fourier series, leading to different learning algorithms. We refer to [17] for an explanation of requirements and properties of kernel-induced mapping functions. We found the Gaussian radial basis functions an effective choice for our classification problem:

$$K(\mathbf{x}, \bar{\mathbf{x}})) = \exp(-\frac{\|\mathbf{x} - \bar{\mathbf{x}}\|^2}{2\sigma^2}). \tag{5}$$

With these definitions, the two basic steps of SVC can be written as: (i) map the training data into a higher-dimensional feature space via $\Phi$, and (ii) construct a hyperplane in feature space F that separates the two classes with maximum margin. Note that there are many linear classifiers that can separate the two classes but there is only *one* that maximizes the distance between the closest data points of each class and the hyperplane itself. The solution to this linear hyperplane is obtained by solving a distance optimization problem given below. The result is a classifier that will work well on previously-unseen examples leading to good generalization. Although the separating hyperplane in F is linear, it yields a nonlinear decision boundary in the original input space $\Theta$. The properties of the kernel function $K$ allow computation of the separating hyperplane without explicitly mapping the vectors in the feature space. The equation of the optimal separating hyperplane in the feature space to determine the class of a new observation $\mathbf{x}$ is given by:

$$\begin{aligned} y = f(\mathbf{x}) &= sgn\left(\sum_{i=1}^{m} y_i \alpha_i . (\Phi(\mathbf{x}) . \Phi(\mathbf{x}_i)) + b\right) \\ &= sgn\left(\sum_{i=1}^{m} y_i \alpha_i . K(\mathbf{x}, \mathbf{x}_i) + b\right). \end{aligned} \tag{6}$$

The Lagrange multipliers $\alpha_i$'s are found by solving the following optimization problem:

$$maximize \quad W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}, \mathbf{x}_i) \tag{7}$$

subject to the following constraints:

$$\alpha_i \geq 0, i = 1, 2, \cdots, m \tag{8}$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0, \tag{9}$$

where $\mathbf{x}_i$'s denote the training data of the behavior vectors. Note that only those data that have non-zero $\alpha_i$ contribute to the hyperplane equation. These are termed *Support Vectors* (SVs). If the data points are linearly separable, all the SVs

will lie on the margin and therefore, the number of SVs will be small. As a result, the hyperplane will be determined by a small subset of the training set. The other points in the training set will have no effect on the hyperplane. With an appropriate choice of kernel $K$, one can transform a linearly non-separable training set into one that is linearly separable in the feature set and apply the above equations as shown. The parameter $b$ (also called the "bias") can be calculated from:

$$b = \frac{1}{2}\sum_{i=1}^{m} \alpha_i y_i [K(\mathbf{x}_i, \mathbf{x}_r) + K(\mathbf{x}_i, \mathbf{x}_s)] \quad (10)$$

where $\mathbf{x}_r$ and $\mathbf{x}_s$ are any SVs from each class satisfying $\alpha_r, \alpha_s > 0$ and $y_r = -1, y_s = 1$.

In practice, a separating hyperplane may not always be computed due to high overlap of the two classes in the input behavior vectors. There are modified formulations of the optimization problem, e.g., with slack variables and *soft margin classifiers* [17], resulting in well-generalizing classifiers in this case. We have not explored this in the present study.

# 5. EVALUATION AND RESULTS

## 5.1 Methodology

We evaluate the proposed behavioral detection framework as follows. First, we wrote several applications for the Symbian OS that emulated known Symbian worms: Cabir, Mabir, Lasco, Commwarrior and a generic worm that spreads by sending messages via MMS and Bluetooth. For each malware, we faithfully reproduced the infection state machine, in particular, the API calls and system events that these malware invoke in the Symbian OS. We also included variants of each malware based on our reviews of the malware family published by various anti-virus vendors. For example, we included 32 variations of Cabir in our implementation, from the descriptions available as part of F-Secure [2] virus descriptions. For most malware, this involved adding different variations in application lifetime, number and subject of messages sent to other devices, file type and attachment sizes, different installation directories for the worm payload, etc. We also built three legitimate applications that shared several common partial behavior signatures with the worms. These are Bluetooth OBEX file transfer, MMS client, and the *MakeSIS* utility in Symbian. The latter creates a SIS archive file from a given list of files and directory names. It is also one of the applications that are typically invoked by Commwarrior and other worms to create a payload on the victim host.

These eight (5 worms and 3 legitimate) applications contain many execution branches corresponding to different behavior signatures that can be captured by the monitoring layer. To execute all possible API calls of these various branches, we run these applications in the emulator many times so that most branches are executed at least once. Each run of an application results in a set of behavior signatures that are captured by the monitoring layer. Depending on the time window over which these behavior signatures are created from the monitoring logs, we obtain partial signatures of various predicate lengths. Next, we filter all repeated signatures (from the same branch and same set of input variables), and collect only the unique signatures generated from the above runs to create a training dataset and a test dataset that are subsequently used for our evaluation. We generate several training and test datasets by repeating the above procedure in the emulator so that expected averages of classification accuracy, false positive and false negative rates can be calculated. Table 1 shows the distribution of unique behavior signatures in one of our training sets with different numbers of atomic propositional variables ("signature length"). The training set consists of a total of 302 malicious signatures (labeled as +1) and 634 legitimate signatures (labeled as -1). Next, we use the training data to calculate the SVM model parameters (see Section 4.1), and classify each signature in the test data using this model to determine the SVM classification accuracy.

| Length | Signatures |
|---|---|
| 1 | 179 |
| 2 | 537 |
| 3 | 175 |
| 5 | 4 |
| 6 | 12 |
| 7 | 11 |
| 8 | 14 |
| 9 | 4 |

**Table 1: Distribution of unique behavior signatures by length (number of propositional variables)**

## 5.2 Accuracy of SVM Classification

To evaluate the effectiveness of the kernel function, we first vary the size of the training set to determine its effect on the classification error.

Table 2 shows the classification accuracy, number of false positives and false negatives for a test data size of 905 unique signatures and different training data sizes. We found that SVC almost never falsely classifies a legitimate application signature to be malicious. On the other hand, for small training data sizes, the number of false negatives (malicious signatures classified as legitimate) is high. However, as the training data size is increased, the classification accuracy increases quickly, reaching near 100% detection of malicious signatures. In our experiments with other training and test dataset sizes, we observed very similar behavior of the classification system.

Table 3 shows the number of Support Vectors (SVs) for each training set. The SVs indicate the size of the SVM model that must be included in the monitoring layer for classifying the run-time behavior signatures. Since a training data size of 150 is sufficient for the five worms we studied, on average, about 50 SVs are included in the SVM model for run-time detection. Each SV corresponds to a signature

| Training Set Size | Accuracy % | False Positives (total count) | False Negatives (total count) |
|---|---|---|---|
| 22 | 82.1 | 0 | 16 |
| 47 | 97.9 | 1 | 18 |
| 56 | 97.5 | 0 | 22 |
| 74 | 98.4 | 0 | 14 |
| 92 | 99.4 | 0 | 5 |
| 122 | 99.5 | 0 | 4 |
| 142 | 99.2 | 0 | 7 |
| 153 | 99.6 | 0 | 3 |
| 256 | 100 | 0 | 0 |
| 356 | 99.7 | 0 | 2 |
| 462 | 100 | 0 | 0 |
| 547 | 99.8 | 0 | 1 |
| 628 | 99.8 | 0 | 1 |
| 720 | 100 | 0 | 0 |
| 798 | 99.8 | 0 | 1 |

**Table 2: SVM classification accuracy of partial behavior signatures**

| Training Set | Support Vectors |
|---|---|
| 22 | 21 |
| 47 | 22 |
| 56 | 20 |
| 74 | 34 |
| 92 | 29 |
| 122 | 30 |
| 142 | 51 |
| 153 | 38 |
| 256 | 48 |
| 356 | 82 |
| 462 | 61 |
| 547 | 95 |
| 628 | 106 |
| 720 | 68 |
| 798 | 186 |

**Table 3: Number of Support Vectors (SVs) for different training data sizes**

in the training dataset and therefore, the number of signatures needed for classification for 100's of variants of these five worms is relatively small.

## 5.3 Generality of Behavior Signatures

A major benefit of behavioral detection is its capability of detecting new malware based on existing malicious behavior signatures in cases where the new malware shares some of the behavior of the existing malware signatures. In case of payload signature-based detection systems, updates must be made to the database to detect the new malware. In order to evaluate the generalization effectiveness of our malicious behavior signatures, we divide the four worms (Cabir, Mabir, Lasco, and Commwarrior (CW)) into two groups. The signatures of the first group ("known worms") are placed in the malicious behavior signature database including their partial signatures. These worms are used to train the SVM classification model. The worms in the second group ("unknown worms") are then executed in the emulator — their signatures are captured in the monitoring layer and comprise the test dataset. The resulting detection rates for different combinations of known and unknown worms are summarized in Table 4. The results show that the combination of TLCK-based signature generation and SVC classification methodology detect unknown worms even when the training data sets are relatively small. This is especially true for malware that are similar in behavior to each other, e.g., Lasco and Cabir. We plan to explore this further as part of our future work so that the size of the malicious signature database can remain small as new strains of malware targeting handsets are discovered.

| Training Set | Testing Set | | | | Overall |
|---|---|---|---|---|---|
| | Cabir | Mabir | CW | Lasco | |
| Cabir | 100 | 17 | 35 | 72.5 | 56 |
| Mabir | 100 | 100 | 51 | 27 | 69.5 |
| CW | 100 | 30.5 | 100 | 69.5 | 75 |
| Lasco | 64.5 | 17.5 | 38.5 | 100 | 55.1 |
| Cabir Mabir | 100 | 100 | 42 | 54 | 74 |
| Cabir CW | 100 | 45 | 100 | 100 | 86.3 |
| Cabir Lasco | 100 | 27 | 50.5 | 100 | 69.4 |
| Mabir CW | 100 | 100 | 100 | 100 | 100 |
| Mabir Lasco | 100 | 100 | 100 | 100 | 100 |
| CW Lasco | 100 | 34.5 | 100 | 100 | 86.3 |
| Cabir Mabir CW | 100 | 100 | 100 | 76.5 | 94.1 |
| Cabir Mabir Lasco | 100 | 100 | 100 | 100 | 100 |
| Cabir CW Lasco | 100 | 99.5 | 100 | 100 | 99.9 |
| Mabir CW Lasco | 100 | 100 | 100 | 100 | 100 |

**Table 4: Detection accuracy (%) for unknown worms**

## 5.4 Overhead of Proxy DLL

The major overhead of our monitoring system comes from replacing the original DLLs with a Proxy DLL that enables real-time logging of API call sequences. To estimate the overhead imposed by Proxy DLL, we measure the execution time of functions before and after they are wrapped by Proxy DLL. Some of the typical function calls are: establish a session with the local Bluetooth service database, display a message in the screen, SMS messaging library calls and allocate new objects. The average overhead is shown in Table 5. To measure the overhead, each function is executed $10,000$ times and we divide the overall time taken by 10,000 to get the overhead of an individual function call. The overhead of Proxy DLL is, on average, 600 microseconds. We conjecture that this is primarily due to the disk access overhead, since each time a function being monitored is called, the monitoring system updates the log file. Since we only selectively monitor a small subset of all the APIs, this overhead is acceptably low for practical deployment. In future, we plan to implement the monitoring layer using ARM's native APIs for real-time tracing so that the overhead can be reduced.

Overall, we find that the behavior signature-based detection is highly effective for mobile malware discovered to date. Our proposed framework can be easily integrated in most mobile OS platforms, without any modification of the

| Session Establishment | Display Message | Object Creation | Average |
|---|---|---|---|
| 564.2 $\mu s$ | 670 $\mu s$ | 625.8 $\mu s$ | 608.5 $\mu s$ |

**Table 5: Overhead of Proxy DLL invocation**

operating environment. Further, the behavioral detection offers a better alternative to signature-based detection due to the small number of behaviors that are sufficient to represent many families of malware.

## 6. RELATED LITERATURE

The most relevant to our work are analysis of mobile viruses and worms [9, 32, 33], behavior-based worm detection [11, 34], backtracking [16] and Support Vectors for intrusion detection [35, 36]. Many well-known mobile viruses and worms, including some of the malware mentioned in this paper, have been analyzed in [9] and [33]. There have also been recent studies to model propagation of such malware in cellular and adhoc (e.g., in Bluetooth piconets) networks. For example, the authors of [32] proposed an analytical model called probabilistic queuing for modeling malware propagation in an ad-hoc Bluetooth environment. Although the focus of our study is primarily handset-based detection, analysis and propagation modeling of mobile viruses and worms help us devise appropriate behavior signatures and response mechanisms.

The pioneering work by Ellis *et al.* [11] was the first to present a novel approach for automatic detection of Internet worms using their behavioral signatures. These signatures were generated from worm behaviors manifested in network traffic, e.g., during transfer of infected payloads to other hosts, tree-like propagation and reconnaissance and changing a server into a client. Our approach is fundamentally different from [11] since it is extremely difficult to generate behavior signatures from network traffic in a cellular network due to their closed nature. The Primary Response from Sana Security [34] is another host-based behavioral approach that monitors desktop applications and employs multiple behavioral heuristics (e.g., writing to Windows Registry, calls to keylogging procedures, process hijacking, etc.) to identify a malicious application. It also correlates actions of multiple running applications to decide whether an application is Spyware. Both of these studies do not address mobile malware that can spread via non-traditional vectors such as Bluetooth and SMS/MMS messages. To the best of our knowledge, there does not exist any prior work to formulate a behavioral detection model for mobile environments. The goal of the BackTracker [16] is to automatically identify potential sequences of activities that occurred in an intrusion. Starting with a single detection point (e.g., a suspicious file), BackTracker recursively identifies files and processes that could have affected the detection point, and displays chains of events in a dependency graph. We use a similar technique to build dependency graphs for generating behavior signatures that manifest in interactions among multiple applications.

Recently, Support Vector Machines (SVMs) have been used in intrusion detection. For example, [35] compares the performance of neural networks-based and SVM-based systems for intrusion detection using a set of benchmark data from DARPA (Defense Advanced Research Projects Agency). The authors of [36] describe Adaptive Model Generation (AMG), a real-time architecture for implementing data mining-based intrusion detection systems. The AMG uses SVMs as one specific type of model generation algorithms for unsupervised anomaly detection. Methods for unsupervised SVM [37] can be easily implemented in our framework, eliminating the need for labeled training data.

## 7. CONCLUDING REMARKS

We have presented a novel detection framework for emerging viruses, worms and Trojans that increasingly target mobile handsets (smart phones, PDAs and similar devices). Our framework begins with extracting key behavior signatures of such malware by applying TLCK logic on a set of atomic steps that these malware attempt to perform on a target host. We have generated a malicious behavior signature database based on a comprehensive review of mobile malware reported to date. Since behavior signatures are fewer and shorter than traditional payload signatures, the database is compact enough to be placed on a handset. Further, a behavior signature describes behavior for an entire family of malware including its variants. This eliminates frequent updating of the behavior signature database as new variants appear. We have implemented a monitoring layer in Symbian for runtime construction of behavior signatures from low-level API calls and system events. In order to identify malicious behavior from partial signatures, we have used SVM to train a classifier based on training data we obtained from the monitoring layer. Our results indicate that behavioral detection not only results in very high detection rates (over 96%) but may also detect new worms and viruses if they display any behavioral pattern already in the database.

## 8. REFERENCES


[1] Symbian, "Symbian Signed platform security," http://www.symbiansigned.com.

[2] F-Secure Corporation, "F-Secure mobile anti-virus," http://mobile.f-secure.com, 2006.

[3] Trend Micro Incorporated, "Trend Micro mobile security," http://www.trendmicro.com/en/products/mobile/tmms/, 2006.

[4] F-Secure, "SymbOS.Cardtrap Trojan description," http://www.f-secure.com/v-descs/cardtrap_a.shtml, September 2005.

[5] S. Singh, , C. Estan, G. Varghese, and S. Savage, "The EarlyBird system for real-time detection of unknown worms," *ACM Workshop on Hot Topics in Networks*, 2003.

[6] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen, "HoneyStat: local worm detection using honeypots," *International Symposium on Recent Advances In Intrusion Detection (RAID)*, 2004.

[7] C. C. Zou, W. Gong, D. Towsley, and L. Gao, "The monitoring and early detection of Internet worms,"

*IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 961–974, 2005.

[8] S. Schechter, J. Jung, and A. Berger, "Fast detection of scanning worm infections," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.

[9] A. Bose and K. G. Shin, "On mobile viruses exploiting messaging and Bluetooth services," *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2006.

[10] W. Enck, P. Traynor, P. McDaniel, and T. La Porta, "Exploiting open functionality in SMS-capable cellular networks," *ACM Conference on Computer and communications security*, pp. 393–404, 2005.

[11] D. R. Ellis, J. G. Aiken, K. S. Attwood, and Scott D. Tenaglia, "A behavioral approach to worm detection," in *ACM Workshop on Rapid malcode (WORM)*, 2004, pp. 43–53.

[12] J. Newsome, B. Karp, and D. Song, "Polygraph: automatically generating signatures for polymorphic worms," *IEEE Symposium on Security and Privacy*, pp. 226–241, 2005.

[13] K. Wang, G. Cretu, and S. J. Stolfo, "Anomalous payload-based worm detection and signature generation," *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[14] H. H. Feng, O. M. Kolesnikov, P. Fogla, and W. Lee, "Anomaly detection using call stack information," *IEEE Symposium on Security and Privacy*, pp. 62–75, 2003.

[15] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," *IEEE Symposium on Security and Privacy*, vol. 120, 1996.

[16] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 1, pp. 51–76, 2005.

[17] N. Christianini and J. Shawe-Taylor, "An introduction to Support Vector Machines and other kernel-based learning methods," Cambridge University Press, 2000.

[18] K. A. Heller, K. M. Svore, A. D. Keromytis, and S. J. Stolfo, "One class Support Vector Machines for detecting anomalous Windows Registry accesses," *IEEE Data Mining for Computer Security Workshop*, vol. 1401, 2003.

[19] Symantec, "SymbOS.Mabir Worm Description," http://securityresponse.symantec.com/avcenter/venc /data/symbos.mabir.html, April 2005.

[20] Symantec, "SymbOS.Commwarrior Worm Description," http://securityresponse.symantec.com/avcenter/venc /data/symbos.commwarrior.a.html, October 2005.

[21] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[22] E. A. Emerson and J. Y. Halpern, *Decision procedures and expressiveness in the temporal logic of branching time*, ACM Press New York, NY, USA, 1982.

[23] W. Penczek, "Temporal logic of causal knowledge," *Proc. of WoLLiC*, vol. 98, 1998.

[24] Gerard Berry and Georges Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[25] F-Secure, "SymbOS.Acallno Trojan description," http://www.f-secure.com/sw-desc/acallno_a.shtml, August 2006.

[26] Symantec, "SymbOS.Skulls Trojan description," http://www.symantec.com/security_response/ writeup.jsp?docid=2004-111913-4830-99, November 2004.

[27] Symantec, "SymbOS.Doomboot Trojan description," http://www.symantec.com/security_response/ writeup.jsp?docid=2006-032110-2210-99, March 2006.

[28] Symantec, "SymbOS.Appdisabler Trojan description," http://www.symantec.com/security_response/ writeup.jsp?docid=2006-111016-4820-99, November 2006.

[29] Author names removed, "On generation of mobile malicious signatures," *Technical Report*, 2007.

[30] V. Vapnik, *The Nature of Statistical Learning Theory*, Springer, New York, 1995.

[31] T. Joachims, "Making large-scale support vector machine learning practical," in *Advances in Kernel Methods: Support Vector Machines*, B. Scholkopf, C. Burges, and A. Smola, Eds. MIT Press, Cambridge, MA, 1998.

[32] James W. Mickens and Brian D. Noble, "Modeling epidemic spreading in mobile environments," in *2005 ACM Workshop on Wireless Security (WiSe 2005)*, September 2005.

[33] S. Töyssy and M. Helenius, "About malicious software in smartphones," *Journal in Computer Virology*, vol. 2, no. 2, pp. 109–119, 2006.

[34] S. Hofmeyr and M. Williamson, "Primary response technical white paper," in *Sana Security*, 2005.

[35] S. Mukkamala, G. Janoski, and A. Sung, "Intrusion detection using neural networks and support vectormachines," *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, vol. 2, 2002.

[36] A. Honig, A. Howard, E. Eskin, and S. Stolfo, "Adaptive model generation:: An architecture for the deployment of data minig-based intrusion detection systems," *Data Mining for Security Applications*, 2002.

[37] B. Scholkopf, J.C. Platt, J. Shawe-Taylor, A.J. Smola, and R.C. Williamson, "Estimating the Support of a High-Dimensional Distribution," *Neural Computation*, vol. 13, no. 7, pp. 1443–1471, 2001.