Behavioral Detection of Malware on Mobile Handsets

Abhijit Bose
IBM TJ Watson Research
New York, USA
bosea@us.ibm.com

Xin Hu   Kang G. Shin
The University of Michigan
Michigan, USA
{huxin,kgshin}@umich.edu

Taejoon Park
Samsung Electronics
Gyeonggi-Do, Korea
joy.park@samsung.com

ABSTRACT

A novel behavioral detection framework is proposed to detect mobile worms, viruses and Trojans, instead of the signature-based solutions currently available for use in mobile devices. First, we propose an efficient representation of malware behaviors based on a key observation that the logical ordering of an application's actions over time often reveals the malicious intent even when each action alone may appear harmless. Then, we generate a database of malicious behavior signatures by studying more than 25 distinct families of mobile viruses and worms targeting the Symbian OS—the most widely-deployed handset OS—and their variants. Next, we propose a two-stage mapping technique that constructs these signatures at run-time from the monitored system events and API calls in Symbian OS. We discriminate the malicious behavior of malware from the normal behavior of applications by training a classifier based on *Support Vector Machines* (SVMs). Our evaluation on both simulated and real-world malware samples indicates that behavioral detection can identify current mobile viruses and worms with more than 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 their deployment in mobile devices.

Categories and Subject Descriptors

C.5.3 [**Computer System Implementation**]: Microcomputers—*Portable devices*; D.4.6 [**Operating System**]: Security and Protection—*Invasive software*; K.6.5 [**Management of Computing and Information System**]: Security and Protection

General Terms

Design, Security

Keywords

Security, Mobile Handsets, Worm Detection, Machine Learning

MobiSys'08, June 17–20, 2008, Breckenridge, Colorado, USA.
Copyright 2008 ACM 978-1-60558-139-2/08/06 ...$5.00.

1. INTRODUCTION

Mobile handsets, much like PCs, are becoming more intelligent and complex in functionality. They are increasingly used to access services, such as messaging, video/music sharing, and e-commerce transactions that have been previously available on PCs 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) that attempt to compromise data confidentiality, integrity and availability of services on handsets. The first mobile worm Cabir [15] appeared in June 2004 targeting Symbian OS and shortly thereafter, the anti-virus industry was startled again by the Windows CE virus, WinCE.Duts, which was the first file injector on mobile handsets capable of infecting all the executables in the devices' root directory. The following three years witnessed a considerable increase in number of both malware families and their variants. By the end of 2006, the known number of mobile malware families and their variants increased by 59% and 75% from year 2005, reaching 35 and 186, respectively [30]. Although mobile malware have not yet caused major outbreak, making some people mistakenly think that mobile malware exist only in the labs of anti-virus companies, their threats are far more real and mobile handsets are expected to become targets of increasing number of malware [29]. For example, in less than one year, the infection of both Cabir and Commwarrior worms have been reported in more than 20 countries [31] and 0.5–1.5% of MMS traffic in a Russian mobile network is made up of infected messages (which is already close to the fraction of malicious code in the email traffic) [32].

In response to this increasing threat, a number of handset manufacturers and network operators have partnered with security software vendors to offer anti-virus programs for mobile devices [25, 11]. However, current anti-virus solutions for mobile devices rely primarily on signature-based detection and are thus useful mostly 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) typical of virus payload. However, several important differences exist between mobile and traditional desktop environments, making conventional anti-virus solutions less efficient or even unworkable for mobile devices. First, mobile devices generally have limited resources such as CPU, memory, and battery power. Although handsets' CPU speed and memory capacity have been increasing rapidly at low cost in recent years, they are still much less than their desktop counterpart. In particular, energy-efficiency is the most critical requirement that limits the effectiveness of complex anti-malware solutions in battery-powered handsets. Because the signature-based approach must check if each derived signature of an application matches any signature in the malware database, it will not be efficient for resource-constrained mobile devices, especially in view of the fact that their malware threats will grow at a fast rate with soon-to-emerge all IP mobile devices based on Wibro and WiMAX technologies. The emergence of crossover worms and viruses [19] that infect a handset when it is connected to a desktop for synchronization (and vice versa) requires mobile applications and data to be checked against both traditional as well as mobile virus signatures. Furthermore, signature-based detection can be evaded by simple obfuscation, polymorphism and packing techniques [34, 36], thus requiring a new signature for almost every single malware variant. These all limit the extent to which the signature-based approach can be deployed on resource-constrained handsets. Second, most published studies (e.g., [39, 54]) on the detection of Internet malware focus on their network signatures (i.e., scanning, failed connection, and DNS request). However, due to the high mobility of devices and the relatively closed nature of cellular networks, constructing network signatures of mobile malware is very difficult. In addition, the emergence of mobile malware that spread via non-traditional vectors (i.e., SMS/MMS messaging and Bluetooth [4, 14]) makes possible malware outbreak whose progress closely tracks human mobility patterns [29], hence requiring novel detection methods. Also, compared to traditional OSes, Symbian and other mobile OSes have important differences in the way file permissions and modifications to the OS are handled. Considering all these differences, we need a new lightweight classifier for mobile handsets that accounts for new malware behaviors. The goal of this work is to develop such a detection framework that overcomes the limitations of signature-based detection while addressing unique features and constraints of mobile handsets.

An alternative to the signature-based approach, *behavioral detection* [13], has emerged as a promising way of preventing the intrusion of spyware, viruses and worms. In this approach, the run-time behavior of an application (e.g., file accesses, API calls) is monitored and compared against malicious and/or normal 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 is more resilient to polymorphic worms and code obfuscation, because it assesses the effects of an application based on more than just specific payload signatures. For example, since encryption/decryption does not alter the application behavior, multiple malware variants generated via run-time packers (e.g., UPX[2]) can be detected with a single behavior specification. As a result, a typical database of behavior profiles should be much smaller than that needed for storing specific payload signatures for each variant of many different classes of malware. This makes behavioral detection particularly suitable for resource-limited handsets. Moreover, behavioral detection has potential for detecting new malware and zero-day [50] worms, because new malware are often constructed by adding new behaviors to existing malware [34] or replacing the obsolete modules with fresh ones, indicating that they share similar behavior patterns with existing malware.

However, there are two challenges in deploying a behavioral detection framework. The first is *specification* of what constitutes normal or malicious behavior that covers a wide range of applications, while keeping the rate of false positives 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 normal and malicious signatures. Our main contribution in this paper is to overcome these two challenges in the mobile operating environment. 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 thus far been reported in the wild. We specify an application behavior as a collection of system events and resource-access attempts made by 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 [21] and backtracking [28]. In our approach, we reconstruct higher-level behavior signatures on-line from lower-level API calls, much like 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 program behavior. The next step is fast and accurate reconstruction of these 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 (e.g., new worms that share only partial behaviors with known worms), we train a machining learning classifier called *Support Vector Machines* (SVMs) [8] with *both* normal and malicious behaviors, so that partial signatures for malicious behavior can be classified correctly 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 totally new behaviors are far fewer than new variants, the updates are not expected to be frequent.

The paper is organized as follows. We review the related literature in Section 2. Section 3 presents an overview of system architecture and design. Section 4 describes the construction of behavior signatures using the TLCK logic and shows examples from current mobile worms. Section 5 and 6 describes our implementation of a monitoring layer and the machine learning algorithm that we use to detect malicious behavior from normal behavior. Section 7 discusses possible limitations of our approach and their countermeasures. We evaluate the effectiveness of behavioral detection in Section 8 against real-world mobile worms and Section 9 concludes the paper.

2. RELATED WORK

Recently, several behavior-based malware analysis and detection techniques have been proposed in the desktop environments to overcome the limitations of traditional signature-based solutions. We first compare and contrast our approach with related work in the area of behavior-based malware detection. Besides the difference in the target environment (mobile vs. desktop environments), several important features also distinguish our work from previous research.

Early efforts, such as the one by Forrest *et al.* [21], are designed for host-based anomaly detection. These approaches observe the application behavior in the form of system call sequences and create a database of all consecutive system calls from normal applications. Possible intrusions are discovered by looking for call sequences that do not appear in the database. Later work improves the behavior profile by applying advanced mining techniques on the call sequences, e.g., rule learning algorithms [9], finite-state automata [41], and hidden Markov model [51]. All these share the same concept of representing a program's normal behavior with system calls and detecting anomalies by measuring the deviation from normal profiles. However, because these approaches ignore the semantics of the call sequences, one of their limitations is that they could be evaded by simple obfuscation or mimicry attacks [49]. Christodorescu *et al.* proposed static semantics-aware malware detection [34] that attempts to detect code obfuscation by identifying semantically-equivalent instruction sequences in the malware variants. They apply a matching algorithm on the disassembled binaries to find the instruction sequences that match the predefined template of malicious behaviors, e.g., decryption loop. By abstracting away the name of register and symbolic constants, this approach is resilient to several code obfuscation techniques. However, as it requires exact matching between the template and application instructions, attacks using the equivalent instruction replacement and reordering are still possible. Similarly, the approach proposed by Kirda *et al.* [12] also uses static analysis of application behavior to determine a spyware component in a browser. It statically extracts a set of Windows API calls invoked in response to browser events, and identifies the interactions between the component and the OS via dynamic analysis. A spyware-like behavior is detected if the component monitors user behavior and leaks this information via some API calls.

Our approach differs from those mentioned above in several ways. The first difference lies in the definition of application behavior. Our approach observes the programs' run-time behavior at a higher level (i.e., system events or resource-access) than system calls [21, 51] and machine instructions [34]. The higher-level abstraction allows the detection algorithm to incoporate more semantics of application behavior, thus improving the resilience to polymorphism and malware variants. Second, our approach employs a run-time analysis, which effectively bypasses the need to deal with code obfuscation, and also avoids the possible information loss of the static approach, since a static analysis often fails to reveal inter-component interaction information [46] and/or disassembly is not always possible for all binaries. Third, in contrast to Forrest's anomaly detection [21] which learns only normal applications' behavior or Christodorescu's misuse detection [34] which matches against only malicious templates, our approach exploits information on *both* normal programs' and malware's behaviors, and employs a machine learning (instead of exact matching) algorithm to improve the detection accuracy. Since the learning and classification are based on two opposite-side data sets, this approach conceptually combines the anomaly detection with misuse detection and therefore, could strike a balance between false positives and false negatives.

There are also several existing works that leverage on the run-time analysis for improving the detection accuracy. Lee and Mody collected a sequence of application events at run-time and constructed an opaque object to represent the behavior in rich syntax [46]. Their work is similar to ours in that both apply a machine learning algorithm on high-level behavior representations. However, their work focuses on clustering malware into different families using nearest-neighbor algorithms based on the edit distance between data samples, while we emphasize only on distinguishing normal from malicious programs. Moreover, we use a supervised learning procedure to make best of existing normal and malicious program information while clustering is a common unsupervised learning procedure. Ellis *et al.* present a novel approach for automatic detection of Internet worms using their behavioral signatures [13]. These signatures were generated from worm behaviors manifested in network traffic, e.g., tree-like propagation and changing a server into a client. Along the same line, NetSpy [24] performs behavior characterization and differential analysis on the network traffic to help automatically generate network-level signatures of new spyware. Our approach is different from the above two approaches in that we focus on the characterization of host-based behavior of mobile malware, incorporating a wide range of system events into behavior signatures.

Previous research we have discussed so far dealt primarily with the desktop environment and thus are not suitable for addressing malware in mobile settings which are capable of spreading via non-traditional vectors such as Bluetooth and SMS/MMS messages. To the best of our knowledge, this is the first attempt to construct a behavioral detection model for mobile environments. The most relevant to our work is the analysis of mobile viruses and worms [4, 35, 47]. Many well-known mobile viruses and worms, including some of the malware mentioned herein, have been analyzed in [4] and [47]. Morales *et al.* test virus detectors for handsets against windows mobile viruses and show that current anti-virus solution performs poorly in identifying virus variants [36]. There have also been recent studies to model propagation of such malware in cellular and ad-hoc (e.g., in Bluetooth piconets) networks [35, 52, 53]. For example, Mickens and Noble proposed the *probabilistic queuing* for modeling malware propagation in an ad-hoc Bluetooth environment [35]. Fleizach *et al.* evaluated the speed and severity of random contact worms in mobile phone networks under several scenarios (e.g., MMS and VoIP) [20], showing aggressive malware could quickly bottleneck the capacity of network links and launch denial-of-service attacks. Although our focus is primarily handset-based detection, analysis and modeling of mobile viruses and worms can help us devise appropriate behavior signatures and response mechanisms. There are also several recent efforts on detecting mobile malware based on their network characteristics. For example, Sarat and Terzis applied random moonwalks to identify mobile worms (focusing on the worms spreading in the wireless domains where mobility is caused by the physical movement of mobile hosts, such as laptops) and their origins. Cheng *et al.* proposed SmartSiren [7], a collaborative virus detection and alert system for smartphones. In SmartSiren, a centralized proxy collects the communication activities from a number of smartphones and performs a statistical analysis on the collected data to detect abnormal communication patterns such as excessive daily usage of SMS/MMS messages.

Applying machine learning algorithms in anomaly detection has also received considerable attention [22]. Recently, Support Vector Machines (SVMs) [40]—a supervised learning algorithm based on the pioneering work of Vapnik [48] and Joachims [26] on statistical learning theory—have been successfully applied in a number of classification problems. For example, Mukkamala *et al.* compared the performance of neural network-based and SVM-based systems for intrusion detection using a set of DARPA benchmark data [37]. Honig *et al.* describe Adaptive Model Generation (AMG) [23], a real-time architecture for implementing data-mining-based intrusion detection systems. AMG uses SVMs as one specific type of model-generation algorithms for anomaly detection.

3. SYSTEM OVERVIEW

Figure 1 illustrates the architecture of the proposed framework. The left side of the figure shows the pre-deployment step in a service provider's network. The handset manufacturer, in cooperation with a service provider, develops a behavior classifier trained by the normal behavior of typical services on handsets as well as malicious patterns for currently-known mobile malware. The classifier along with a behavior signature database is then deployed in the handsets, where the framework consists of three building blocks. The *monitor agent* collects the application behavior in the form of system events/API calls at multiple locations (e.g., Bluetooth and MMS) and reports aggregated behavior signatures to the *de-*

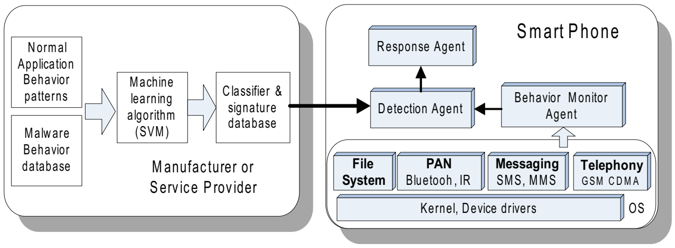

Normal Application Behavior patterns
Malware Behavior database
Machine learning algorithm (SVM)
Classifier & signature database
Manufacturer or Service Provider
Smart Phone
Response Agent
Detection Agent
Behavior Monitor Agent
File System
PAN Bluetooh, IR
Messaging SMS, MMS
Telephony GSM CDMA
Kernel, Device drivers
OS

Figure 1: System Overview

tection agent. Upon receiving the signature, the detection agent performs machine-learning classification with the pre-loaded classifier and labels the activity as malicious or innocent. This result could be used for activating response mechanisms (e.g., requesting the user's verification, rate-limiting, drop packets,etc.) against any detected malicious behavior. Additionally, since the classifier parameters and database need to be updated when new mobile threats are discovered, the service provider can update the handset detection agent with over-the-air updates. In what follows, we will detail this proposed behavioral detection framework.

4. MALICIOUS BEHAVIOR SIGNATURES

4.1 Temporal patterns

We define ***behavior signature*** as the manifestation of a specification of resource accesses and events generated by applications, including malware. We are interested in only those behaviors that indicate the presence of a malicious activity, such as damage to the handset operating environment (e.g., draining the battery or overwriting system files), installing a worm payload, sending out an infected message, etc. For this, it is not sufficient to monitor a single event (e.g., a file read/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 lifecycle 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—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()*) in Symbian OS. This is often used by applications for exchanging data or files among nearby handsets. On their own, any such calls will appear 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 [44] or Commwarrior [43]. 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 the specific implementation (e.g., infection vectors) of a malware. However, the *specification* of temporal manifestation of malicious behaviors can still be prescribed *a priori* by considering their effects on the handset resources and the environment. For this reason, behavioral detection is more resilient to code obfuscation and polymorphism than the signature-based detection.

A simple representation of malicious behavior can be given by ordering the corresponding actions using a vector clock [33] 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. Among the results of this research, the *temporal logic of causal knowledge* (TLCK) [38] 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 environment.

4.2 Temporal Logic of Malicious Behavior

This section describes how to specify malicious behavior in terms of system events, interposed by temporal and logical operators. The specification of malicious behavior is the first step of any behavioral detection framework. Though our presentation is primarily targeted at the Symbian OS, it can be extended for other mobile OSes 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*
- $\Diamond_t$ *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 $[t-k,\ t]$.

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

1. Time is represented by a 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 [3] 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 predicates 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, 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 approach as opposed to payload signatures, given the small memory and storage footprint of these devices.

4.3 Example: Commwarrior Worm

The Commwarrior worm [43] targets Symbian Series 60 phones and spreads 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. Once the installation of the SIS file is permitted by the user, the SIS file installer installs the payload and 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. 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 upon receiving the message. The daily window for replication via MMS is only from 00:00 to 06:59 hours. Commwarrior also resets the device on the 14-th day of each month, deleting all users' settings and data. Figure 2 graphically describes the Commwarrior's behavior. Our goal is to convert this graphical representation to a behavior signature using logical and temporal operators defined in Section 4.2.

Note that the specification of Commwarrior behavior requires the monitoring of a small number of processes and system calls ($N = 5$), 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 with 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.

InstallApp(f,files,dir): Install a SIS archive file f by extracting *files* and installing them in directory *dir*.

LaunchProcess(p,parent): Launch an application p by a *parent* process, 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.

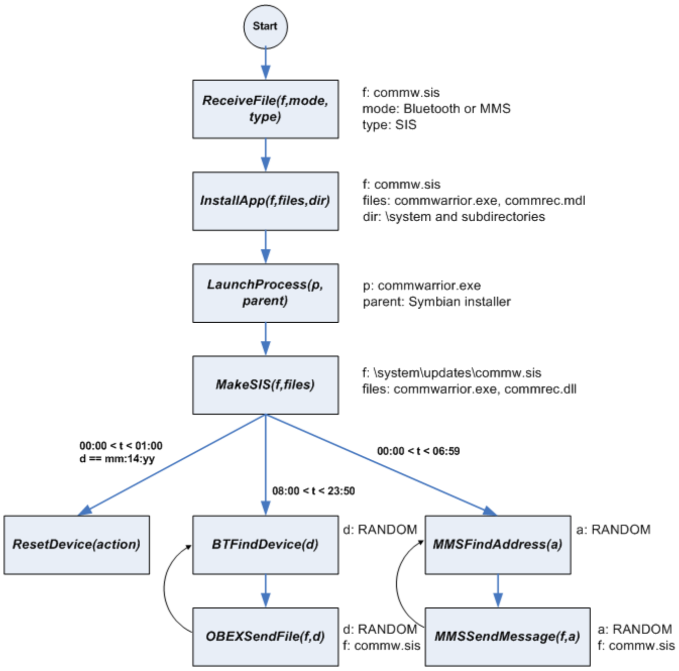


Figure 2: Behavior signature for Commwarrior worm

OBEXSendFile(f,d): Transfer a file f (with full 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 seven higher-level signatures that correspond to the major behavioral steps of the worm family. Notice that since the filename of a malware's payload and its activation time can be changed easily(as in case of many variants of commwarrior worms), when constructing high level signatures we exclude these detailed information to improve the generality of the behavior signatures. Four of these seven 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 an outgoing message. Later, in Section 6, we show that malicious behavior can be detected more accurately by training an SVM.

- $\odot_t(bt-transfer) = \Diamond_t(BTFindDevice(d)) \wedge (\odot_t(OBEXSendFile(f,d)))$

- $\odot_t(mms - transfer) = \Diamond_t(MMSFindAddress(a)) \wedge$
  $(\odot_t(MMSSendMessage(f,a)))$

- $\odot_t(init - worm) = \odot_t(ReceiveFile(mode = Bluetooth))$
  $\vee (\odot_t(ReceiveFile(mode = MMS)))$

- $\odot_t(activate - worm) = \Diamond_t(init - worm) \wedge$
  $(\odot_t(InstallApp) \wedge \odot_t(LaunchProcess))$

- $\odot_t(run - worm - 1) = \Diamond_t(activate - worm) \wedge$
  $(\odot_t(MakeSIS) \wedge \odot_t(VerifyDayofMonth) \wedge$
  $(\Diamond_{1:00}^{0:00}(SetDevice)))$

- $\odot_t(run - worm - 2) = \Diamond_t(activate - worm) \wedge$
  $(\odot_t(MakeSIS) \wedge (\Diamond_{23:59}^{8:00}(bt - transfer)))$

- $\odot_t(run - worm - 3) = \Diamond_t(activate - worm) \wedge$
  $(\odot_t(MakeSIS) \wedge (\Diamond_{6:59}^{0:00}(mms - transfer)))$

4.4 Generalized Behavior Signatures

In order to create generalized signatures that are not specific to each variant of malware, we studied more than 25 distinct families of mobile viruses and worms targeting the Symbian OS, including their *140* variants reported thus far. For each malware family, we generated propositional variables corresponding to its actions, identified the argument lists for each variable, and assigned TLCK operators to construct the behavior signatures 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. 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. Currently the behavior signatures are generated by hand and we plan to investigate automatic feature extraction and signature generation as a future work. We find that the malware actions can be naturally placed into three categories i.e., actions that affect ***User Data Integrity*** *(UDI)*, actions that damage ***System Data Integrity*** *(SDI)* and ***Trojan-like Actions***, 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 and write these data files.

Example: Acallno [18] 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 OS 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 InfraRed 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.[1] The variable returns true when *directory* matches against a hash table of precompiled list of critical system and application directories. At this point, because this is a potentially dangerous operation, the installation process can be suspended until the user permits to go ahead with the installation. However, since the user may be tricked by attacker's social engineering tactics and fail to recognize the malicious application, these generalized SDI signatures, together with other behavior signatures, will later go through the machine learning algorithm to identify malware behavior.

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 `\System`. The `install` directory contains installation and uninstallation information for applications. Many Symbian applications log error codes in `\System\bootdata` when they generate a panic. Without these directories, most handset applications become unusable.

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

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 `\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. 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 under `\System` (SDI action). The Trojan-like action of Cardblock is the locking of the MMC card 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.

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 [45]. 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 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.

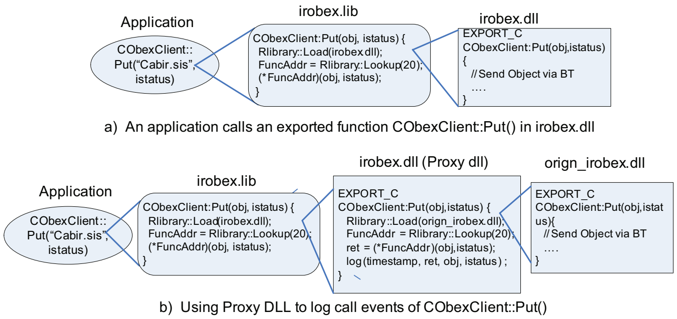
However, Symbian OS v9 is unlikely to completely stop mobile malware for the following reasons. First, given the vast number of mobile applications available on the Internet, one cannot expect all applications to be signed with the Symbian root certificate. In fact, most third party applications are installed using self-signed certificate and thus, cannot be free of malicious codes. Second, it may not prevent mobile worms that spread via SMS/MMS or Bluetooth and social engineering techniques. Third, there will increasingly be higher-level vulnerabilities (e.g., Web scripts) as handsets become Internet-capable and these vulnerabilities could be exploited by malware to propagate itself.

5. RUN-TIME CONSTRUCTION OF BEHAVIOR SIGNATURES

To build a malware detection system, the behavior signatures must be constructed at run-time by monitoring the target system events and API calls. We describe next the implementation of the monitoring layer in Symbian Operating System.

5.1 Monitoring of API Calls via Proxy DLL

Because Symbian is a proprietary OS and provides very little public information about either kernel monitoring APIs or system-wide hooks, intercepting API calls is very difficult, if not impossible. Fortunately, the Symbian SDK is accompanied with a Symbian OS emulator, which accurately emulates almost all functions of a real handset. The emulator implements the Symbian APIs in the form of Dynamic Link Libraries (DLLs). This is the feature that we were able to exploit to build the monitoring system. Specifically, we use a technique called *Proxy DLL* to collect the API traces of applications running in the emulator. Proxy DLL is a popular technique used by anti-virus tools, e.g., to hook into Winsock's I/O functions and network data for virus signatures [27].



```
CObexClient::
Put(“Cabir.sis”,
istatus)
```

```
CObexClient:Put(obj, istatus) {
 Rlibrary::Load(irobex dll);
 FuncAddr = Rlibrary::Lookup(20);
 (*FuncAddr)(obj, istatus);
}
```

```
EXPORT_C
CObexClient:Put(obj,istatus)
{
  //Send Object via BT
  ….
}
```

a) An application calls an exported function CObexClient::Put() in irobex.dll

```
CObexClient::
Put(“Cabir.sis”,
istatus)
```

```
CObexClient:Put(obj, istatus) {
 Rlibrary::Load(irobex.dll);
 FuncAddr = Rlibrary::Lookup(20);
 (*FuncAddr)(obj, istatus);
}
```

```
EXPORT_C
CObexClient:Put(obj,istatus) {
  Rlibrary::Load(orign_irobex.dll);
  FuncAddr = Rlibrary::Lookup(20);
  ret = (*FuncAddr)(obj,istatus);
  log(timestamp, ret, obj, istatus) ;
}
```

```
EXPORT_C
CObexClient:Put(obj,istat
us){
  //Send Object via BT
  ….
}
```

b) Using Proxy DLL to log call events of CObexClient::Put()

Figure 3: Proxy DLL to capture API call arguments

A proxy DLL is a shim between the application and the real DLL file. It is named exactly the same as the original DLL and gets loaded in the run time by the application. The proxy DLL loads the real DLL and passes all API calls from the application to the original DLL. This allows the proxy DLL to intercept and record ev-

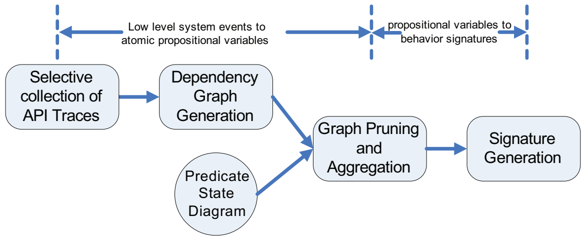

Low level system events to atomic propositional variables
propositional variables to behavior signatures
Selective collection of API Traces
Dependency Graph Generation
Predicate State Diagram
Graph Pruning and Aggregation
Signature Generation

Figure 4: Major components of the monitoring system

ery detail about the API call events from the application while still maintaining its correct operation. Figure 3 illustrates an example of a proxy DLL that we implemented in the Symbian OS emulator to log CObexClient::Put() API call (exported by irobex.dll), a function commonly used by mobile worms to transfer infected payload to nearby devices. The *.lib* file, which is called *import library*, is statically linked to the application and invokes the exported functions in the original DLL with functions' ordinal number.[2] Without the proxy DLL (Figure 3(a)), when CObexClient::Put() is called, the import library loads the original irobex.dll, searches for the entry point of the function by its ordinal number (i.e., 20) and invokes the function. On the other hand, in Figure 3(b), the original irobex.dll is replaced with a proxy DLL which also exports CObexClient::Put(), but is instrumented with logging functionalities. When the application calls CObexClient::Put(), the proxy DLL will be loaded into the memory and spy the application's call events (e.g., timestamp, caller ID, arguments, etc.). Meanwhile, the proxy DLL passes the function call to the original DLL to ensure the normal operation. 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, i.e., a small set of functions that constitute the atomic proposition variables described in Section 4. The number of function calls to be monitored may increase in future as new malware families emerge.

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 4.2), and then by graph pruning and aggregation, a set of behavior signatures.

5.2 Stage I: Generation of Dependency Graph

Using the proxy DLL, our monitoring agent logs a sequence of target API calls invoked by running processes. The next step is to correlate these API calls using the TLCK logic described in Section 4.2, and build the behavior signatures (see Section 4). To efficiently represent the interactions and correlation among processes, we construct a dependency graph from logged API calls by applying the following rules to the captured API calls.

Intra-process rule: API calls that are invoked by the same process 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.

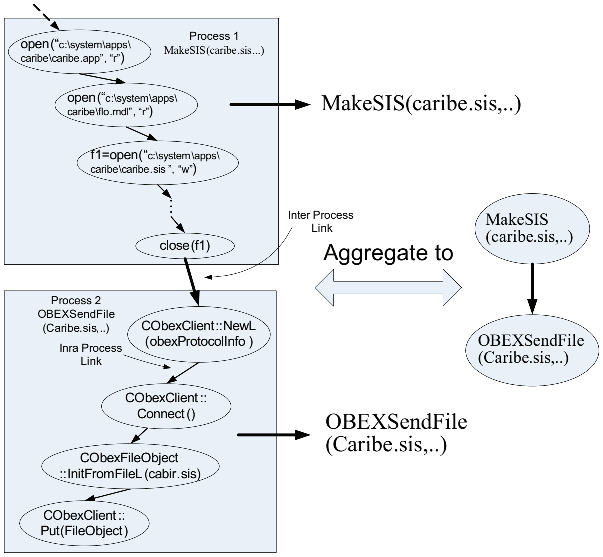

Process 1
MakeSIS(caribe.sis...)
open("c:\system\apps\caribe\caribe.app", "r")
open("c:\system\apps\caribe\flo.mdl", "r")
f1=open("c:\system\apps\caribe\caribe.sis ", "w")
close(f1)
MakeSIS(caribe.sis,..)
Inter Process Link
Aggregate to
MakeSIS (caribe.sis,..)
OBEXSendFile (Caribe.sis,..)
Process 2
OBEXSendFile (Caribe.sis,..)
Inra Process Link
CObexClient::NewL (obexProtocolInfo)
CObexClient:: Connect ()
CObexFileObject ::InitFromFileL (cabir.sis)
CObexClient:: Put(FileObject)
OBEXSendFile (Caribe.sis,..)

Figure 5: Dependency graphs for constructing atomic propositional variables

Inter-process rule: Because malware behavior often involve multiple processes, we define following rules to identify inter-process relationships. **(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 [28]. Figure 5 shows an example of the inter-process dependency rule, where Process 1 packages some files into a SIS file (caribe.sis), and subsequently, Process 2 reads the file and sends it to in-range devices via Bluetooth. This constructs a larger signature: $MakeSIS(caribe.sis,..) \wedge OBEXSendFile(caribe.sis,..)$.

5.3 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 grow very large. A simple expiration policy is to destroy the call-chain graph of a process upon its 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. 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 specify 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:

[2] Each function exported by a DLL is identified by a unique ordinal number. To facilitate programming, each DLL is often accompanied by an import library which allows the application to invoke DLL functions by function names instead of their ordinal numbers.

1. The process didn't have inter-process dependency relationships with any other process (i.e., it was independent);
2. Its graph doesn't *partially* match with any malicious behavioral signatures;
3. It didn't create or modify any file or directory in the list of directories maintained in a hash table of critical user and system directories (see Section 4.4); 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 one or more atomic propositional variables. The advantage of encoding atomic variables 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 propositional variables along its state transition graph. The outcome of the two stages is a behavior signature that is to be classified either malicious or harmless by the detection system.

6. BEHAVIOR CLASSIFICATION BY MACHINE LEARNING ALGORITHM

The behavior signatures for the complete life-cycle of a malware, such as those developed in Section 4, are placed in the behavior database for run-time classification. However, if we have to wait until the complete behavior signature of a malware is constructed, 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 eventually manifesting as malicious behavior. These partial signatures (e.g., *sms-transfer* and *init_worm* in Section 4.3) 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. Moreover, the behavior-detection system can detect even new malware or variants of existing malware, whose behavior is only partially matched with the signatures in the database. Therefore, instead of exact matching, we need a mechanism to classify partial (or incomplete) malicious behavior signatures. We use a learning method for classifying these partial behavior signatures from the training data of *both* normal and malicious applications. In what follows, we describe a particular machine learning approach called *Support Vector Machines* (SVMs) that we applied for the binary classification of partial behavior signatures.

6.1 Support Vector Machines

SVMs, based on the pioneering work of Vapnik [48] and Joachims [26] 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. 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 generalization of the algorithm which measures its ability to learn future unknown data without error. The flexible generalization ability of SVMs makes it suitable for real-world applications with a limited amount of training data. 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 Θ. 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., the scalar distance) between data points $\mathbf{x}$ and $\bar{\mathbf{x}}$ in Θ:

$$k : \Theta \times \Theta \mapsto \Re(\mathbf{x}, \bar{\mathbf{x}}) \rightarrowtail k(\mathbf{x}, \bar{\mathbf{x}}). \quad (1)$$

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

$$k(\mathbf{x} \cdot \bar{\mathbf{x}}) = \Sigma_{i=1}^{d} (\mathbf{x})_i \cdot (\bar{\mathbf{x}})_i. \quad (2)$$

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

$$k(\mathbf{x}, \bar{\mathbf{x}}) \rightarrow K(\mathbf{x}, \bar{\mathbf{x}}) := (\Phi(\mathbf{x}) \cdot \Phi(\bar{\mathbf{x}})) \quad (3)$$

There are many choices for the kernel functions, such as polynomials, radial basis functions, multi-layer perceptron, splines and Fourier series, leading to different learning algorithms. We refer to [8] 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 (σ is the width of the Gaussian):

$$K(\mathbf{x}, \bar{\mathbf{x}}) = \exp(-\frac{\|\mathbf{x} - \bar{\mathbf{x}}\|^2}{2\sigma^2}). \quad (4)$$

With these definitions, the two basic steps of SVM can be written as: (i) map the training data into a higher-dimensional feature space via Φ, 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 Θ. 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:

$$\begin{aligned} y = f(\mathbf{x}) &= sgn\left(\sum_{i=1}^{m} y_i\,\alpha_i \cdot (\Phi(\mathbf{x}) \cdot \Phi(\mathbf{x}_i)) + b\right) \\ &= sgn\left(\sum_{i=1}^{m} y_i\,\alpha_i \cdot K(\mathbf{x}, \mathbf{x}_i) + b\right). \quad (5) \end{aligned}$$

The Lagrange multipliers α_i's are found by solving the following optimization problem:

$$\textit{maximize} \quad W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{6}$$

subject to the following constraints:

$$\sum_{i=1}^{m} \alpha_i y_i = 0, \;\; \alpha_i \geq 0, i = 1, 2, \cdots, m \tag{7}$$

where $\mathbf{x}_i$'s denote the training data of the behavior vectors. Note that only those data that have non-zero α_i contribute to the hyperplane equation. These are termed *Support Vectors* (SVs). If the data points are linearly separable in the feature space, all the SVs will lie on the margin and therefore, the number of SVs are typically small. This means that the hyperplane will be determined by only a small subset of the training set and the other points in the training set will have no effect on the hyperplane, so the size of the resulting signature database will be small and suitable for resource-constrained handset environments. 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:

$$b = \frac{1}{2} \sum_{i=1}^{m} \alpha_i y_i [K(\mathbf{x}_i, \mathbf{x}_r) + K(\mathbf{x}_i, \mathbf{x}_s)] \tag{8}$$

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* [8], resulting in well-generalizing classifiers in this case.

7. POSSIBLE EVASION & LIMITATIONS, AND THEIR COUNTERMEASURES

We now discuss (i) several possible ways attackers may evade our detection framework as well as a few limitations associated with it, and (ii) possible countermeasures against them.

A malware writer who has learned our detection system may try to evade it by modifying existing malware or creating new malicious behaviors. Similar to code obfuscation [34], program behavior can be obfuscated by behavior reordering, file or directory renaming, normal behavior insertion and equivalent behavior replacement. For behavior reordering, attackers cannot arbitrarily reorder the program behaviors, since some temporal constraints must be satisfied in order to maintain the correct functionality. For example, the temporal ordering of ReceiveFile and InstallApp cannot be reversed, because the installation of malware depends on first receiving the malware file. Similar temporal ordering must hold within a Bluetooth or MMS transfer. On the other hand, the ordering between Bluetooth and MMS transfers can be changed arbitrarily. Our approach is resilient to reordering in that we use TLCK to capture only those inherent constraints and construct high-level behavior signatures (Section 4), thus providing strong protection against the reordering obfuscation. File/directory renaming is done by assigning different names to malware executables or installation directories so that exact matching will fail to detect the variants. To resist such obfuscation, we abstract away the file/directory name and keep only file types, e.g., installation type (.SiS) and system directory in the behavior signatures (`\system`). The third type of obfuscation (i.e., normal behavior insertion) inserts useless or innocent behavior sequences that do not influence the program functionality among malicious behaviors. Our approach is resilient to this obfuscation because it does not seek any exact match of some malicious template with program behavior. Instead, the signature is constructed by validating each behavior propositional variable and then classified via a learning algorithm. Moreover, in our framework, most behavior variables on their own are normal, but they together, when connected with temporal relationships, become strong indicators of malicious activities. Hence, insertion of useless or normal behavior cannot evade the detection. Finally, equivalent behavior substitution replaces groups of behaviors with other sequences that have the same functionality but are not captured with our signatures. Alternatively, attackers may try to circumvent the detection by mimicry attacks [49], i.e., disguising its behavior as normal sequences while having the same effects on the system. Although our approach cannot completely handle this type of obfuscation, it makes substitution or mimicry more difficult. First, the high-level definition of behavior signature hides implementation details. Finding equivalent behavior sequences is more difficult than that for machine instruction sequences, where a rich instruction set is available [34]. Second, even if the monitor layer misses some malware behaviors due to lack of specification for equivalent behavior sequences, the machine learning algorithm may still be able to make correct classification based on the captured partial signatures that match the existing malicious behavior.

However, our approach also has a few limitations. First, since the current set of behavior signatures is defined based on the existing mobile malware, the detection might fail if most behaviors of a mobile malware are completely new or the same as normal programs (this is equivalent to the case when attackers manage to substitute most of their malicious behaviors with equivalent sequences that are not detectable by our system). This is a fundamental limitation of *any* behavioral approach that detects unseen anomalies based on their similarities from existing training data. Fortunately, in most cases, new malware share a great deal of similarity with their predecessors for the following reasons. First, due to the growing complexity and modularization of current malware, addition of new behaviors to existing malware is a common technique used by malware writers [34]. Creation of truly new malware is very rare. Second, runtime packers (e.g., UPX, MEW, FSG, [5] etc.) are one of the most widely-used techniques for generating malware variants (e.g., over 92% of malware files in wildlist 03/2006 are packed [5]). These packed malware runs by unpacking the original executable codes and then transferring control to them. Hence, the run-time behavior of packed variants is the same as the original malware. A recent report by Symatec [10] also confirms our observation that most new malware are variations of existing ones. Second, like all host-based mechanisms, our system can be circumvented by malware that can bypass the API monitoring (e.g., install rootkit, place hook deeper than the monitor layer) or modify the framework configuration (e.g., disable the detection engine). Countermeasures have been proposed for desktop environments, such as the rootkit revealer [1] and the tiny hypervisor to protect kernel code integrity [42]. These approaches have the potential to be applied in mobile settings with the growing capability and power of handset devices.

In summary, while there are several ways an attacker could attempt to evade detection, our approach, as demonstrated in the next section, is still effective in detecting many mobile malware variants and thus raises the bar significantly high for mobile malware writers.

8. EVALUATION

The main components of the proposed framework include the monitor agent and the behavior detection agent. Because the monitor agent is platform-dependent, we implemented and tested it on the Symbian Emulator using the Proxy DLL technique described in section 5.1. On the other hand, the high-level detection agent, which translates API traces collected from monitor agent into behavior signatures and performs SVM classification algorithm, is not necessarily coupled with specific OS platform. As a result, we currently implemented it separately on the desktop machine running Symbian Emulator and used the specific SVM implementation called libsvm [6]. However, since the detection algorithm is OS-independent, it is easy to be ported to the handset environment (e.g., Symbian OS).

8.1 Methodology

Since the monitor agent is implemented in the Symbian OS Emulator, which is a windows application simulating the phone hardware and OS, application executables for a real smartphone (ARM platform) cannot be directly executed in the emulator. In order for an application to run in the emulator, it has to be recompiled from its source code for the emulator platform. Due to limited access to the source codes of worms and normal applications, we evaluate the proposed behavioral detection framework first by emulating program behavior and then testing it against real-world worms whose source codes are available to us. First, we wrote several applications 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 reproduced the infection state machine, especially the resource accesses and system events that these malware trigger in the Symbian OS. We also included variants of each malware based on our review of the malware family published by various anti-virus vendors. For most malware, this required addition of different variations in malware lifetime, number and contents of messages, file name, type and attachment sizes, different installation directories for the worm payload, etc. We also built 3 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 OS that creates an SIS archive file from a given list of files.

These 8 (5 worms and 3 legitimate) applications contain many execution branches corresponding to different behavior signatures that can be captured by the runtime monitoring. We run these applications repeatedly so that most branches are executed at least once. Each run of an application results in a set of behavior signatures captured by the monitoring layer. Depending on the time window over which these behavior signatures are created from the monitoring logs, we obtain partial/full signatures of various lengths. Next, we remove all repeated signatures 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 and calculate expected averages of classification accuracy, false positive and negative rates. Next, we use the training data to train the SVM model and classify each signature in the test data to determine the classification accuracy.

8.2 Accuracy of SVC

We first evaluate the accuracy of SVC in detecting existing or known worms (the capability of detecting unknown worms will be discussed in the next subsection). To evaluate the effectiveness of SVC in capturing known worms, we vary the size of the training set (in all cases, around two thirds of the training set consists of the normal signatures and the remaining one third are worm signatures) to determine its effect on the classification error. Notice that the signatures of same worm may appear in both training and testing set. However, because of the different running environments and execution paths, the test set also contains many signatures that do not appear in the training set. Table 1 shows the classification accuracy, number of false positives and false negatives for a test set of 905 distinct 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 trend and results.

Training Set Size	number of Support Vectors	Accuracy	False Positive	False Negative
22	21	82.1%	0	16
47	22	97.9%	1	18
56	20	97.5%	0	22
74	34	98.4%	0	14
92	29	99.4%	0	5
122	30	99.5%	0	4
142	51	99.2%	0	7
153	38	99.6%	0	3
256	48	100%	0	0
356	82	99.7%	0	2
462	61	100%	0	0
547	95	99.8%	0	1
628	106	99.8%	0	1
720	68	100%	0	0
798	186	99.8%	0	1

Table 1: Classification accuracy.

Table 1 also shows the number of Support Vectors (SVs) for each training set, which indicate the size of the SVM model (Section 6) 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 5 worms we studied, on average, about 50 SVs are included in the SVM model for run-time detection. Each SV corresponds to a signature in the training dataset and therefore, the

Training Set (“known") worms	Testing Set (“unknown" worms)				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 2: Detection accuracy (%) for unknown worms

number of signatures needed for classification for hundreds of variants of these worms is relatively small.

8.3 Generality of Behavior Signatures

A major benefit of behavioral detection is its capability of detecting new malware based on existing malicious behavior signatures if the new malware, as is commonly the case, share some behavior with the existing malware signatures. In case of payload signature-based detection systems, their signature database must be updated to detect the new malware. In order to evaluate the effectiveness of 'generalization' in our behavior detection algorithm, we divide the 4 worms (Cabir, Mabir, Lasco, and Commwarrior (CW)) into 2 groups. The signatures of the first group ("known worms") are placed in the malicious behavior signature database and used to train the SVM model. The worms in the second group ("unknown worms") are then executed in the emulator; their signatures are captured by the monitoring layer and comprise the test dataset. The resulting detection rates for different combinations of known and unknown worms are summarized in Table 2. The results show that the combination of TLCK-based signature generation and SVC methodology is able to detect previous unseen worms, especially for malware that share similar behavior with existing ones. For example, when the training set contains 3 malware, the detection achieves very high accuracy for the remaining unknown malware, whose behavior is likely to be covered by existing malware. Therefore, the size of the malicious signature database could remain small as new strains of malware targeting handsets are discovered.

8.4 Evaluation with Real-world Mobile Worms

To confirm the effectiveness of our behavior-based detection, we tested it against real-world mobile malware. We were able to collect the source codes of 2 Symbian worms, Cabir and Lasco. Cabir [15] replicates over Bluetooth by scanning to discover Bluetooth devices in its range and sending copies of infected worm payload (SIS file). Lasco [16] propagates via Bluetooth in the same manner. It is also capable of inserting itself into other SIS files in the devices, so that Lasco will automatically start when the injected files are installed.

We collected the behavior signatures for these worms by compiling and running them on the Symbian emulator. Considering the fact that the dynamic analysis results may depend on the run-time environment, we ran each malware sample 10 times with different environmental settings such as running time, number of neighboring devices, number of failed Bluetooth connections, etc. For example, in one setting, the number of neighboring devices is zero, thus making the worm continuously search for new devices. This generates varying-size signatures that describe the worm behavior in each specific environment. We apply the trained classifier (with training set size 92 as in Table 1) on each captured signature. SVC was found to achieve 100% detection of all worm instances.

To test our framework's resilience to the variations and obfuscation, we modified the source codes and implemented worm variants based on F-Secure mobile malware descriptions [17]. Since we did not find any information for Lasco variants, we mainly focus on creating variants for Cabir. Cabir has 32 variants (Cabir.A-Z, AA, AB, AC,AD, AE, AF), most of which are minor variations of the original Cabir worm. For example, Cabir.Z differs only in the infected SIS file name (i.e., file-renaming obfuscation) from Cabir.B, which, in turn, differs trivially from the original Cabir worm by displaying a different message on the screen. Since our behavioral detection abstracts away the name details, these variants are easily detectable.[3] As a result, we only implement 3 major types of variations. First, the original Cabir has an implementation flaw that makes it lock on the device found first and never search for the others, which slows down its spreading speed. One major variation (e.g., Cabir.H) is to fix this bug by enabling the worm to search for new targets when the first device is out of range. We modified the replication routine in the source code and implemented this variation. The second major variant is Cabir.AF, which is a size-optimized recompilation of the original Cabir. We implemented this variation by incorporating the compression routine found in Lasco source code, which utilizes the zlib library to compress the SIS file. Third, we implemented a synthetic behavior-reordering obfuscation. The original Cabir worm always prepares an infected SIS file before searching for nearby Bluetooth devices. In contrast, the new variant finds an available device first, then creates a SIS file, and finally transfers it via Bluetooth. We collected behavior signatures for these variants by running each of them 10 times in different environments and apply the trained classifier. Again, SVC is found to be resilient to these obfuscation, and successfully detects all the variants.

8.5 Overhead of Proxy DLL

The major overhead of our monitoring system comes from the Proxy DLL that logs API call events in real time. To estimate this overhead, we measure the execution time of functions before and after they are wrapped by Proxy DLL. Average overheads (over 10,000 repeated executions) for some typical API calls are: 564.2 μs (establish a session with the local Bluetooth service database), 670 μs (display a message on the screen), 625.8 μs (SMS messaging library call) and 608.5 μs (allocate new objects). The overhead is, on average, 600 microseconds. We conjecture that this is primarily due to disk accesses. Since we only need to monitor a small subset of API calls (e.g., Bluetooth device search, OBEX transfer, SMS/MMS send/receive) which are usually invoked sporadically during an application's lifetime, the overall overhead is expected to be low. We confirm this by measuring the running time of a Bluetooth file exchange application before and after enabling the monitor agent. The result shows that to transfer a 10K byte file, the original application spends, on average, 10.36 seconds. On the other hand, the same process takes 10.6 seconds with the monitor agent enabled, which represents only 3% overhead.

8.6 Summary and Discussion of Results

Overall, the behavior-based approach is found to be highly effective in detecting mobile malware and variants thereof. However, we also noticed the limitation of current evaluation: the monitoring layer is implemented in a Symbian emulator, rather than in a real handset, due to the restricted access of the Symbian OS kernel information, which is only available to their business partners or licensed users. This keeps us from testing the framework against a wide range of normal applications whose source codes are not available. Thus, we have to resort to the emulator to accurately reproduce the programs' real behavior. Nevertheless, the synthetic traces could overestimate the detection accuracy and/or underestimate the framework overhead. These limitations prevent us from comparing the proposed behavior-based approach against existing

[3] Although the signature-based approach is also resilient to simple renaming obfuscation, some variants (e.g., Cabir.AA), besides renaming the infected file, modify and recompile the source code, thus resulting in different binary images from the original worm. Hence, to detect this variant, a signature-based approach may require additional signatures, while a single behavior signature for the original worm will suffice for the behavioral detection.

signature-based solutions. Although signature-based solutions are known to have a very low false positive rate, they cannot detect new malware or variants of existing malware whose signatures are not in the database (e.g., through simple obfuscation). By contrast, behavior-based approaches like ours are usually more resilient to variations of malware in the same family (since they share similar behaviors) and can thus respond to new malware outbreaks in a timely manner. We are currently collaborating with a major mobile phone manufacturer to implement the proposed framework on real handsets. Despite these limitations, our evaluation results on real mobile worms still indicate that the behavioral detection offers a good alternative to signature-based detection, because of its smaller database and capability of detecting new malware variants.

9. CONCLUSIONS

We have presented a behavioral detection framework for viruses, worms and Trojans that increasingly target mobile handsets. The framework begins with extraction of key behavior signatures of mobile malware by applying TLCK on a set of atomic steps. 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 and can thus be placed on a handset. A behavior signature also has the advantage of describing behavior for an entire malware family including their variants. This eliminates the need for frequent updates of the behavior signature database as new variants emerge. We have implemented the monitoring layer on the Symbian emulator for run-time construction of behavior signatures. In order to identify malicious behavior from partial signatures, we used SVM to train a classifier from normal and malicious data. Our evaluation of both emulated and real-world malware shows that behavioral detection not only results in high detection rates but also detects new malware which share certain behavioral patterns with existing patterns in the database.

Acknowledgement

The work reported in this paper was supported in part by the National Science Foundation under Grant No. CNS 0523932, Samsung Electronics, and Intel Corporation. We would like to thank our shepherd Dr. Anthony D. Joseph and the anonymous reviewers for their constructive comments and helpful advice.

10. REFERENCES

[1] Rootkitrevealer 1.71. http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx.

[2] Upx: the ultimate packer for executables. http://upx.sourceforge.net/.

[3] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] A. Bose and K. G. Shin. On mobile viruses exploiting messaging and Bluetooth services. *SecureComm*, 2006.

[5] T. Brosch and M. Morgenstern. Runtime packers: The hidden problem? Black Hat USA 2006.

[6] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[7] J. Cheng, S. H. Wong, H. Yang, and S. Lu. Smartsiren: virus detection and alert for smartphones. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 258–271, New York, NY, USA, 2007. ACM.

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

[9] W. W. Cohen. Fast effective rule induction. In *Proc. of the 12th International Conference on Machine Learning*, 1995.

[10] S. Corp. Symantec internet security threat report trends. http://www.symantec.com/business/theme.jsp?themeid=threatreport.

[11] K. Corporation". Kaspersky Anti-Virus Mobile. http://usa.kaspersky.com/products_services/antivirus-mobile.php.

[12] E.Kirda, C.Kruegel, G.Banks, G.Vigna, and R.Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

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

[14] 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*, 2005.

[15] F-secure. Cabir. http://www.f-secure.com/v-descs/cabir.shtml.

[16] F-secure. Lasco. http://www.f-secure.com/v-descs/lasco_a.shtml.

[17] F-secure. Mobile detection descriptions. http://www.f-secure.com/v-descs/mobile-description-index.shtml.

[18] F-Secure. SymbOS.Acallno Trojan description. http://www.f-secure.com/sw-desc/acallno_a.shtml, Aug 2005.

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

[20] C. Fleizach, M. Liljenstam, P. Johansson, G. M. Voelker, and A. Mehes. Can you infect me now?: malware propagation in mobile phone networks. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 61–68, New York, NY, USA, 2007. ACM.

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

[22] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *ID'99: Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring*, 1999.

[23] 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.

[24] H.Wang, S.Jha, and V.Ganapathy. Netspy: Automatic generation of spyware signatures for nids. In *Proceedings of Annual Computer Security Applications Conference*, 2006.

[25] T. M. Incorporated. Trend Micro mobile security. http://www.trendmicro.com/en/ products/mobile/tmms/, 2006.

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

[27] Y. Kaplan. Api spying techniques for windows 9x, nt and 2000. http://www.internals.com/articles/apispy/apispy.htm.

[28] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems (TOCS)*, 2005.

[29] J. Kleinberg. The wireless epidemic. *Nature*, 449(20):287–288, September 2007.

[30] K. Lab. Kaspersky security bulletin 2006: Mobile malware. http://www.viruslist.com/en/analysis?pubid=204791922.

[31] K. Lab. Mobile malware evolution: An overview, part 2. http://www.viruslist.com/en/analysis?pubid=201225789.

[32] K. Lab. Mobile threats - myth or reality? http://www.viruslist.com/en/weblog?weblogid=204924390.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[34] M.Christodorescu, S.Jha, S.A.Seshia, D.Song, and R.E.Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[35] J. W. Mickens and B. D. Noble. Modeling epidemic spreading in mobile environments. In *2005 ACM Workshop on Wireless Security (WiSe 2005)*, September 2005.

[36] J. A. Morales, P. J. Clarke, Y. Deng, and B. M. G. Kibria. Testing and evaluating virus detectors for handheld devices. *Journal in Computer Virology*, 2(2):135–147, November 2006.

[37] S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vectormachines. *Intl. Joint Conf. on Neural Networks, 2002*, 2, 2002.

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

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

[40] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.

[41] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society.

[42] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, 2007. ACM.

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

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

[45] Symbian. Symbian Signed platform security. http://www.symbiansigned.com.

[46] T.Lee and J.J.Mody. Behavioral classification. http://www.microsoft.com/downloads/details.aspx?FamilyID =7b5d8cc8-b336-4091-abb5-2cc500a6c41a&displaylang=en,2006.

[47] S. Töyssy and M. Helenius. About malicious software in smartphones. *Journal in Computer Virology*, 2(2), 2006.

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

[49] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems, 2002.

[50] 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.

[51] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[52] G. Yan, , and S. Eidenbenz. Bluetooth worms: Models, dynamics, and defense implications. In *Computer Security Applications Conference, 2006.*

[53] G. Yan, H. D. Flores, L. Cuellar, N. Hengartner, S. Eidenbenz, and V. Vu. Bluetooth worm propagation: mobility pattern matters! In *Proceedings of the 2nd ACM symposium on Information, computer and communications security, 2007.*

[54] C. C. Zou, W. Gong, D. Towsley, and L. Gao. The monitoring and early detection of Internet worms. *IEEE/ACM Transactions on Networking*, 13(5):961–974, 2005.