Designing System-level Defenses against Cellphone Malware

Liang Xie* Xinwen Zhang† Ashwin Chaugule* Trent Jaeger* Sencun Zhu*
∗Department of Computer Science and Engineering, Pennsylvania State University, PA 16802
† Samsung Information Systems America, San Jose, CA 95134
{lxie, avc114, szhu, tjaeger}@cse.psu.edu xinwen.z@samsung.com

***Abstract*—Cellphones are increasingly becoming attractive targets of various malware, which not only cause privacy leakage, extra charges, and depletion of battery power, but also introduce malicious traffic into networks. In this work, we seek system-level solutions to handle these security threats. Specifically, we propose a mandatory access control–based defense to blocking malware that launch attacks through creating new processes for execution. To combat more elaborated malware which redirect program flows of normal applications to execute malicious code within a legitimate security domain, we further propose using artificial intelligence (AI) techniques such as Graphic Turing test. Through extensive experiments based on both Symbian and Linux smartphones, we show that both our system-level countermeasures effectively detect and block cellphone malware with low false positives, and can be easily deployed on existing smartphone hardware.**

***Keywords*-Cellphone Malware; Graphic Turing Test; Mandatory Access Control;**

I. Introduction

Mobile communication systems which support both voice and data services have become ubiquitous and indispensable. This advantage however comes with a price — mobile networks and terminal devices (e.g., smartphones and PDAs) are becoming attractive targets to attackers. Especially, the popularity of mobile services such as email and messaging, and their dependence on common software platforms such as Symbian and Linux, have made mobile devices ever more vulnerable. Situation gets worse as mobile devices quickly evolve [1]. According to F-Secure [2], currently there are more than 350 mobile malware (or viruses) in circulation. Examples of the most notorious threats to cellphones include Skull [3], Cabir [4], and Mabir [5] targeting at Symbian operating systems. The research organization IDC estimates that by 2008 the market for mobile security software will grow yearly by 70% [6].

In this paper, we refer to *cellphone malware* as malicious codes that exploit vulnerabilities in cellphone software and propagate in networks through popular services such as Bluetooth and messaging (SMS/MMS) services. Cellphone malware are devastating to both mobile users and network infrastructures. Users of compromised cellphones could be unconsciously charged for numerous messages delivered by malware and their phone batteries could be quickly drained. Other reported damages include loss of user data and privacy and software crashes. For example, a Trojan spy named Flexispy [7] monitors a victim's call history and contacts, and delivers these sensitive information to a remote server.

Research on quarantining cellphone malware has started recently. The straightforward defense is to mirror the strategy against PC viruses with the inception of security patches for cellphones. However, it is challenging for users to acquire signature files in a timely manner; besides, downloading often causes extra charges. Some recent work propose more active solutions, e.g., by designing algorithms to automatically identify compromised phones based on user interactions [8] and suggesting a proactive containment framework to quarantine those suspected devices, and by designing a collaborative virus detection/alerting system [9]. These approaches, however, rely on network or external agents to throttle worm spread. We believe that defense mechanisms deployed on cellphones are more effective both in detecting malware in a timely fashion and in preventing malicious messages from entering wireless networks. Indeed, on this line, a labeling technique [10] has been proposed to protect a phone against cross-service attacks from its PDA interface. However, it does not provide further solutions for containing automated malware [11].

Contributions: First, a comprehensive study on malware's attack strategies is important in designing effective countermeasures against them. Our categorization and analysis on major attack strategies reveal challenging issues to be addressed in effective defenses. Second, we propose a mandatory access control–based mechanism which controls program accesses to important system resources through enforcing a customized access control policy in the context of cellphones. This mechanism is effective in defeating malware which execute malicious codes in new processes. Third, to combat automated malware which gain controls of existing processes to execute malicious codes, we propose a more comprehensive defense which identifies and blocks malware using AI techniques such as Graphic Turing test (GTT). Using MMS as an example, through challenging a user, our approach differentiates whether a delivery of MMS message is legitimate (user-initiated) or illegal (malware-initiated). Moreover, this approach does not reply on known malware signatures. Fourth, we provide real implementations and evaluate our defenses on major smartphone platforms. Experimental results including benchmark and performance data demonstrate that our solutions are light-weight and effective in blocking various malware.

Outline: The rest of the paper is organized as follows. In Section II, we provide an overview of our defenses: an access control–based scheme and a GTT-based scheme. We present design and implementation details of the defenses in Section III and evaluate their effectiveness and performance through experiments in Section IV. We introduce related work in Section V and conclude in Section VI.

A. *Malware's Attack Strategies*

From a software perspective, we study two basic forms of attack strategies adopted by cellphone malware. The key difference between these two is whether a malware creates a new process to launch its attack.

Attack strategy I: *In this case, a malware always creates a new process to execute its malicious code, and to compromise the cellphone*. User operations are usually required in this case, e.g., a user installs a downloaded software, or opens a received message. The newly created process has its own descriptor, which describes address space, execution state, and security attributes (i.e., security context [1]). This security context is different from that of the parent process which invokes it. Due to simplicity, this strategy is widely adopted by most existing malware. Symbian and Windows Mobile users cannot change the kernel; Symbian programs register themselves within a platform and make system calls within framework APIs (provided by vendors) to use system services. Realizing this, cellphone malware launch attacks through legally installed applications. For example, Mabir [5] includes its malicious code in an SIS installation file [2] and attaches it to an infection vector (e.g., an MMS message). Once a recipient activates this SIS file, a malicious program is installed and executed, incurring damages to the phone. For instance, a program invokes messaging APIs to deliver numerous malicious messages, or steal user private information and send to a malicious server.

Attack strategy II: *In this case, a malware does not create a new process in the phone. Instead, it redirects the program flow of a legitimate application (e.g., messaging process) to execute its malicious code within a legitimate security context.* Such attacks mostly happen in cellphones with open-source platform OS and application frameworks, e.g., Linux-based smartphones. An automated malware adopts this strategy through exploiting software vulnerabilities such as stack buffer-overrun [12] in a cellphone software to 'hijack' its normal program flow and launch attacks. Note that the malicious code to be executed after the malware gains control of the program counter may exist in the original program's address space so that it is legal to OS. In some cases, these code may even invoke certain system calls to cause crashes. Compared with strategy I, this attack method is more elaborated and automated hence more challenging because it keeps the security context of the hijacked process while executing the malicious code and does not rely on user operations to propagate.

[1] For example in Linux, a process's security context typically consists of identity, role, and type/domain of the process.

[2] The format of an user-initiated installation file to distribute Symbian applications.

II. Overview of the Countermeasures

We assume that a malware launches attacks from application-level software, i.e., it could compromise phone applications such as email and MMS/SMS messaging, etc. However, it cannot break the kernel. Kernel level attacks such as rootkits have not been found on cellphones [2]. We also notice that a few techniques can be adopted to prevent malware's kernel-hacking in cellphones, for example, integrity measurement architecture (IMA) [13], [14] helps identify falsifications on kernel code, virtual machine techniques [15] helps isolate kernel-level attacks from a legal OS. Kernel protection is out of the scope of this paper.

We design a mandatory access control (MAC)-based protection scheme to defeat malware which launch attacks through creating new malicious processes (attack strategy I). To combat more elaborated malware which redirect program flows of normal applications (e.g., messaging and emailing) to execute malicious code within a legitimate security domain (attack strategy II), we adopt AI techniques such as Graphic Turing test (GTT) as the countermeasure. Note that this approach addresses both forms of attacks.

A. *Access Control–based Protection*

We first anatomize the MMS/SMS messaging, a major infection vector for malware. To deliver a message to the recipient, a messaging process invokes a sequence of key system calls to access important system resources (e.g., file, socket, and modem device) and access related system services. Using a Linux-based phone (e.g., Motorola A1200 and E680) as an example, to search for the email address or phone number of the recipient, a messaging process named $mmsclient$ invokes *open("address_book", O_RDONLY)* to access the contact list in the phone address book; to deliver a SMS/MMS message, $mmsclient$ calls *fd=open("/dev/ttyS0", O_RDWR)* to open the modem device [3] (i.e., serial device $ttyS0$) and invokes *write(fd, message, length)* to send a composed message $message$ to the modem, which eventually processes the message and transmits it to the air interface; to send an email messaging through Wi-Fi interface, a process named $smtpclient$ invokes *socket(AF_INET, SOCK_STREAM)* to create a stream socket through which it communicates with an SMTP server. These system calls are critical monitoring and authentication points in kernel, because malware's final goal is to invoke system calls as a normal process does and gain accesses

[3] Modem device is a separate phone unit, which contains radio frequency (RF) and base-band components, low-level coding-decoding software and protocol stacks.

to important system resources and launch attacks. Therefore, our goal with access control-based protection is to distinguish malicious processes from normal ones and give different permissions to them at these checking points.

We adopt a defense model that is similar to Security Enhanced Linux (SELinux) [16] to achieve strong protection in cellphones. SELinux employs Linux Security Modules (LSM) inside kernel to implement MAC policies, which assign different permissions to different processes based upon the least privilege principle. Specifically, SELinux associates security labels (contexts) of the form *user:role:type* to all subjects (processes) and objects (files/directories, programs, sockets, etc.). Within a security context, the `type` attribute represents the type of a subject or an object, e.g., *sshd_t* and *syslogd_t*. Instead of directly associating a *user* with *type*s, SELinux associates a *user* with a set of *role*s and a *role* with a set of *type*s. The *role* simplifies the management of users. This means that access control in SELinux is primarily enforced via a so-called Type-Enforcement [17] policy model.

B. GTT-based Protection

Using messaging malware as an example, to identify and throttle malware within a cellphone device, we first need to differentiate a malware-initiated messaging process from a normal user-initiated messaging process. This is relatively easy when a malware creates an unauthorized process to invoke system calls or messaging APIs to execute its malicious program, because this malicious process has its new security context which can be identified according to some predefined security policies. However, in the case when a malware launches attacks through hijacking the program flow of a registered application, it becomes a challenging task because the malicious process retains the security context of the registered application, which is still legitimate.

To detect such malware behavior, one approach is to use content-based filtering. For example, messages that are delivered with .SIS attachments and attractive titles are more suspicious than others. However, this often results in non-neglectable false positives and false negatives, as we experience in the context of email spam filtering. Moreover, this involves running complex machine learning tools, which do not fit for cellphones that are usually resource-constrained. Another approach is to let a user manually confirm every message that is leaving her phone. Specifically, when sending a message to the air interface, the messaging framework asks the user to choose YES/NO using either keypad or touch-screen to confirm the message delivery. This of course helps block some simple attacks. However, more elaborated malware could compromise the application-layer interface of the keypad/touch-screen driver and inject a false input to circumvent this simple protection.

We adopt artificial intelligence (AI) techniques to address this challenging issue. Our idea is to rely on AI techniques to authenticate an ongoing process and identify malware behaviors. Specifically, our countermeasure involves executing a Graphic Turing test (GTT) before a phone message framework eventually delivers a message to the air interface of the phone. GTT has a nice feature that a human being can always pass a Turing test while an automated malware cannot. In this case, even if an elaborated malware compromises the application-layer interface of the keypad driver, it is still unable to figure out the correct answer of a GTT challenge. This countermeasure can identify malware adopting both forms of attack strategies, because it does not simply verify the security context of an ongoing process; instead, it differentiates a malicious message originator from a normal user through authenticating their genuine natures. This approach also helps detect a wider variety of malware, including new malware that are unknown to security vendors (e.g., zero-day or polymorphic ones).

Although our GTT-based defense requires user operations during message deliveries, the rate of normal MMS/MMS messaging (on the order of 0$\sim$10.07 messages/hour [18]) is unlikely to be very frequent due to the limited resources of dedicated signaling channels, e.g., Stand-alone Dedicated Control Channels (SDCCHs) configured for SMS messaging in GSM [19] Base Stations. Moreover GTT takes only several seconds, which can be ignored when considering the time a user spends to compose a message. Note that a user is motivated to take GTT because she wants to prevent cellphone from sending numerous malicious messages which cause privacy leakage, extremely high service charges, and quick depletion of battery power Under certain circumstances, a user may choose to launch GTT on her cellphone using a *random-challenging mode*, in which the messaging framework of the phone randomly (e.g., with a probability of 10%) executes GTTs to authenticate message initiators. This is especially useful when the user needs to deliver a large amount of messages, for example, during holidays or birthdays. Note that also it does not provide perfect prevention, random-challenges are still very effective in detecting malware which tend to generate numerous malicious messages. We argue that one desire property of malware is stealthy to bypass IDS. With our technique, such malware will be eventually captured even if it transmits at a low rate. After that, the user can disinfect this phone by installing a security patch or sending it for professional maintenance. Overall, the random-challenging strategy nicely deals with a tradeoff between user convenience and cellphone security.

III. Deployment of the Countermeasures

A. Access control–based Protection

1) Achieving Access Control on Smartphones: Here we explain in details[4] how this scheme works in Qtopia phone

[4]Our technical report [20] shows the complete flow of applying access control on key system resources to contain cellphone malware.

edition 4.2 [21], the major application platform for Linux-based phones such as Motorola A1200, Sony Mylo, and many others [22]. Recall that our strategy is to authenticate and authorize accesses to key resources such as user address book, modem devices, and Wi-Fi interfaces on a mobile device. Therefore, we focus our control on the system calls that are invoked towards these resources, such as *open(”address_book”, O_RDONLY)*, *open(“/dev/ttyS0”, O_RDWR)*, and *write(fd, message, length)*. Fortunately, hooks have been defined by LSM in most places where we want to control system file openings and device accesses. Therefore, the main tasks in this scheme are to (1) define SELinux policy rules to specify process and object types and permissions to access critical system resources, (2) label corresponding programs to specific types thus enabling their legitimate permissions, and (3) limit type transitions via program invocations from any other type to those allowed to access protected resources. We explain these steps below.

The fundamental goal of our access control–based scheme is to only allow legitimate accesses to key resources while denying others. The key problem is to identify what kind of privileges are required for each program, such as to satisfy the least privilege principle. One feature of SELinux policy is that all *allow* rules in a *policy.conf* are positive; that is, a single rule always adds some permissions to a subject type. Therefore, to prevent any illegal subject type from having unexpected permissions, we define “private” types of our target resource objects such that they are only visible to subject types in the same *domain*. Here by domain we mean all trusted subjects that are allowed to access a target object. For example, in the Qtopia platform, Qt applications and plugins (e.g., *qtmail* and *qtmms*) are typically trusted to the address book object. Thus permissions from them to read/write user address book are allowed, while those permissions from other subjects are denied. To provide private object types, we leverage the recent SELinux Loadable Policy Module (LPM) mechanism, which offers a flexible way to create self-contained policy modules that are linked to an existing policy. Private object types can be defined in an LPM such that they can only be accessed by subject types defined in the same module, which provides a flexible and safe mechanism to define desired permissions for subjects. If a subject type needs to access object types defined in another module, an LPM can define *public* object types that can be accessed by *external* subject types via pre-defined interfaces, i.e., only pre-defined subjects can access them.

To correctly enforce an SELinux policy and confine application behaviors, another key issue is subject and object labelling. As SELinux is label-based access control, assigning appropriate labels to target subjects and objects, and controlling the permissions to change these labels are critical. Particularly, in the cellphone environment we have to solve two problems: (1) we need to label key resources and trusted subjects with appropriate labels in a cellphone OS, and (2) besides policy rules to enable trusted subjects’ permissions on target resources, we need to define rules for permission *labelfrom* and *labelto* to subjects and key resources such that only legitimate processes can get these permissions and re-label them. Significantly different from traditional OS where an object owner determines other program’s permissions to access the object, SELinux policy controls which domain can have these permissions.

2) A Sample Policy for Secure Messaging: Next, we use a concrete example to demonstrate how cellphone malware can be effectively contained with above techniques. Specifically, we show how to construct a policy module for securing messaging services in Linux-based phones that adopt Qtopia Phone 4.2 as their application platform. Qtopia provides a number of integrated messaging applications (e.g., SMS, MMS, and email client) to phone users. Here we use *qtmail* – the email messaging application as an example. We note that policy rules can be defined for other applications within this module or in other modules.In this example, we examine malware exploits on user address book. We show how to define SELinux policy rules and explain why such policy rules help protect the Qtopia messaging application against a malware that adopts attack strategy I. We first construct the following policy module, as shown in the table.

As previously discussed, we define two types: *qtmail_t* for *qtmail* client application, and *qtaddressbook_t* for user email address book object file. The first two allow rules specify that only *qtmail_t* subject can read and write the address book object, while any other subjects (*base_t*) can have the *getattr* permission of it, e.g., to get some meta-information of the object. Any other accesses to the address book object are denied by SELinux security module. The next two allow rules state that only *sysadm_t* subjects can label objects (including program file and data file) to/from *qtmail_t* and *qtaddressbook_t*; that is, only a system administrative program can make a program file to *qtmail_t* such as to read object file of *qtaddressbook_t*. This prevents the possibility that a malicious program can relabel an arbitrary program to *qtmail_t* which then can read the address book file. The last three allow rules define the domain transition that when a *qtmail_t* program is executed, the new process is transited from *base_t* to *qtmail_t* automatically. This ensures that process *qtmail_t* can only be created by executing a *qtmail_t* program file.

Hence, by enforcing this policy module, we can ensure that only *qtmail_t* process can read and write user address book, a *qtmail_t* process can only be created by executing a *qtmail_t* program file, and only system administrative process can label a program file to type *qtmail_t*. Therefore, without an explicit administrative change, any malicious process launched by a malware cannot be labelled as *qtmail_t* and its access to the user address book object will be denied.

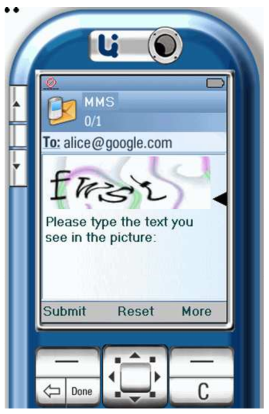
```
policy_module(qtmail, 1.0)
require {
  type base_t;
  type sysadm_t;
}
type qtmail_t;
type qtaddressbook_t;
allow qtmail_t qtaddressbook_t:file *;
allow base_t qtaddressbook_t:file getattr;
allow sysadm_t qtmail_t:{dir file}
  {relabelto relabelfrom};
allow sysadm_t qtaddressbook_t:{dir file}
  {relabelto relabelfrom};
allow type_transition base_t qtmail_t:
  process qtmail_t;
allow qtmail_t qtmail_t:file entrypoint;
allow base_t qtmail_t:process transition;
```

For attack II, as the legitimate *qtmail* process is hijacked, which is already labelled with *qtmail_t* when it is launched, our access control–based scheme cannot block its malicious access to the user address book. Therefore we need a more intelligent scheme as described in next subsection.

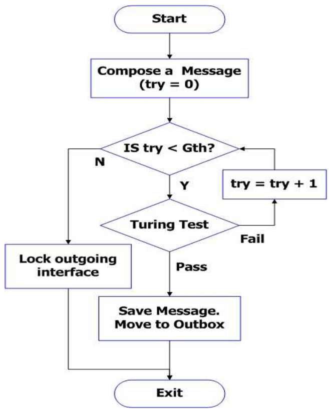
B. Graphic Turing Test on Smartphones

AI techniques such as GTT help people differentiate human beings from automated machines. Network servers adopt this strategy to filter automated messages generated by spammers or worms, e.g., in securing email account registrations [23] and fighting against Deny of Service (DoS) attacks [24]. We design a GTT-based mechanism which helps a cellphone identify malware-controlled program flow and take further countermeasure against it. Although here we use MMS-related malware to demonstrate the defense, our approach is equally effective in combating malware propagating via other vehicles such as Email, Bluetooth, and Wi-Fi. In the following section, we assume cellphones adopt a random-challenging mode of GTT.

1) Instantiating Graphic Turing Test: We choose to incorporate a visual CAPTCHA (Completely Automated Public Turing test to Tell Computers and Human Apart) [25] test into a mandatory point of each message delivery. CAPTCHA is a program that generates and grades visual tests that most humans can solve, but automated programs including malware cannot. Thus, for a normal cellphone user, she simply needs to pass an easy test (e.g., recognizing an image of distorted text) before sending her newly composed message to the output interface (usually a buffer named Outbox in cellphone). Note that decoding images of distorted text is well beyond the capability of cellphone malware. Therefore, a malware most likely fails this authentication and all its unauthorized out-going messages will be eliminated from the cellphone. As most cellphones are resource-constrained (typically with 220MHZ CPU and 32 MByte RAM), we cannot embed very complex visual tests in them. One effective yet simple realization of CAPTCHA is GIMPY [26], which concatenates an arbitrary sequence of letters to form a word and renders a distorted image of the word. In most variants of GIMPY (e.g., EZ-GIMPY [27]), a cluttered and textured background is also added to the text. The distortion and clutter are sufficient to confuse current OCR (optical character recognition) software. This is the fact that GIMPY relies on the guarantee that a cellphone user can recognize the text while existing automated programs cannot. Figure 1 shows a scenario of GIMPY test and a detailed flow of phone software. Here a user in messaging authenticates herself by entering an ASCII text in the same sequence of letters as that appear on the phone screen.



(a) GIMPY test in cellphone



(b) Flow of protection

Figure 1. **GTT-based protection in a cellphone.**

In choosing an appropriate instance of GTT, there is a tradeoff between a test's complexity (i.e., security) and its convenience to users. Normally users will not spend much time on a test. Otherwise, they would rather switch it off. However, a GIMPY test with few characters are likely to be broken by intelligent malware. Although for demonstration purpose we use GIMPY in this work, we can easily substitute it with a more secure instance of GTT, e.g., Animal-PIX, which asks users to choose among a pre-defined set of animals. This intuitive method also helps reduce user response time. One fortunate thing is that, malware cannot install complex machine-learning tools such as neural networks in resource-constrained cellphones to build classifiers and recognize images, although they have done it in PC systems. Indeed, even it is smart enough to bypass one test in a while, failure once will expose itself and will be removed by a user.

2) Embedding GTT in Smartphones: The next issue is where to embed GTT tests. In our experiments with Symbian smartphones, we incorporate a GIMPY test into an appropriate API in the messaging framework (e.g., used in Nokia S60 or Sony-Ericsson UIQ 3 SDKs), which must be invoked each time when an application accesses the modem interface to deliver a message. From software perspective, a modem is responsible for data coding/decoding and protocol processing in a cellphone, and any messaging request (including the message content itself) has to be transferred to modem in order to be transmitted to the air interface (using AT commands defined in GSM protocol 0707, 0705 [19]). Specifically, vendors such as Nokia and Sony-Ericsson can integrate a GTT test into a Symbian messaging library function named $CMmsClientMtm :: SendL()$. This function is invoked each time when a user application initializes a communication entity named $iMmsMtm$ and composes a new MMS message (see Figure 2 in Technical Report [20]). According to a typical messaging procedure, $iMmsMtm$ first sets the message content as well as possible attachment, the recipient's address, etc., and copies the message to a temporary buffer named Outbox (through calling $CMmsClientMtm :: SendL()$). At the same time, $iMmsMtm$ starts a timer using $wait \rightarrow start()$ and has the system scheduler deal with the final message delivery to the modem device. Through augmenting the library function $CMmsClientMtm :: SendL()$ into $CMmsClient Mtm :: NewSendL()$, we place the GIMPY test at the point right after the message content has been built and just before it is transferred to the buffer of the modem device. Thus, only authorized messages that pass authentications are accepted by the modem. Because GIMPY test is embedded in the messaging API, a user can conveniently update it through downloading a new function library from a vendor's website. For example, she can upgrade the software to execute another form of GTT, e.g, Animal-PIX [25].

We have demonstrated how to embed GTT in messaging libraries. However, in an open-source platform (e.g., Linux) where an intelligent malware tries to bypass the messaging framework and directly transfers malicious messages to the modem interface, GTT can be embedded in kernel level, e.g., in a kernel module. Comparing with the simple YES/NO scheme we mentioned in Section II-B, GTT is more robust and secure because it is unlikely to be broken by application-level malware. Even its interfaces with application views are compromised, a malware still cannot figure out the correct answers of a GTT test. However, in the YES/NO scheme, the keypad driver's interfaces with applications are always exposed to malware.

IV. Experiments and Evaluations

A. Experimental Settings

In our tests, we chose OMAP-5912OSK [28] as the development board for Linux-based phones. We chose Trolltech [21] Qtopia Phone Edition 4.2 as the application platform. Qtopia is currently running on a wide variety of Linux-based phones [22] (e.g., Motorola A1200 and OpenMoko), and it provides a complete set of C++ SDKs, user-friendly tools and APIs for application developers. Our source codes of malware and defenses were first built into PC executables and tested in a Linux-based emulator. Finally, these programs were cross-compiled into target executables for ARM-9 processor and deployed to the OMAP board.

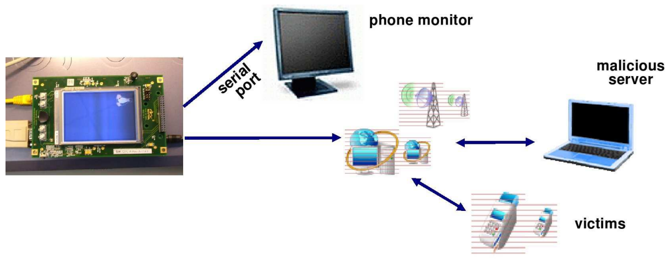


Figure 2. **Configuration of cellphone experiments (using OMAP-5912OSK platform as an example).**

Figure 2 shows the configuration of our smartphone experiments. An administrator can control the OMAP board and log its events through an external *Minicom* terminal (serial port). The OMAP board connects to a standard modem device through which it communicates with other smartphones within 2G/3G cellular networks. In addition, each board can access the Internet through its on-board network interface. We also implemented an external malicious server [29], which establishes connections (SSL) with the on-board malware and receives private user contact information stealthily gathered from the OMAP board. Therefore, besides the malware attacks inside the board, the malicious server itself can exploit disclosed user information and launch automated messaging attacks to vulnerable phones by executing its own messaging service such as *sendmail*. We implemented three representative malware: Cabir [4], CommWarrior [30], and Lasco [31] in both Linux and Symbian-based smartphones. These malware adopt attack strategy I. In addition, we implemented an automated malware which adopts attack strategy II in compromising smartphone devices. Refer to Section I-A for attack strategies.

B. Access control–based Defense

As SELinux LSM inserts hooks and checks access control policies in many system calls, it introduces overhead to main primitive functions and inter-process communications (IPCs). We studied the performance of our access control–based scheme with microbenchmark to investigate the overhead for various low-level system operations such as process, file, and socket accesses in phone devices. Our test includes two policies: our simplified policy for cellphone environment, and the original NSA example policy. We compared the results with those measured without SELinux involved (baseline).

Our benchmark tests were performed with the LMbench 3 suites [32]. Table I shows the measurements in microseconds. The measurements show the trend that security checks in null I/O and stat operations introduce more overhead percentage than others. However, the total time of security checks is quite small in these operations, i.e., less then 20 microseconds. Typical file related operations such as open/close and create/delete, and process related operations such as fork, exec, and sh, also have very small overhead.

Benchmark	Baseline	Our Policy	%	NSA Policy	%
null I/O	22.5	31.4	39	29.5	31
stat	48.4	66.5	37	67.9	40
open/close	1113	1179	6	1277	14
0KB create	2985.1	3257.3	9	3436.4	15
0KB delete	3174.6	3268.0	3	3546.1	11
fork	4169	4270	2	4281	2
exec	18K	19K	5	19K	5
sh	78K	83K	6	83K	6
pipe	264.2	319.6	20	322.8	22
AF_UNIX	460	529	15	511	11
UDP	574.1	648.3	13	817.9	12
TCP	771.4	858.6	11	1044	35

Table I
Benchmark result in OMAP-5912. Measurements are in microseconds. Smaller is better.

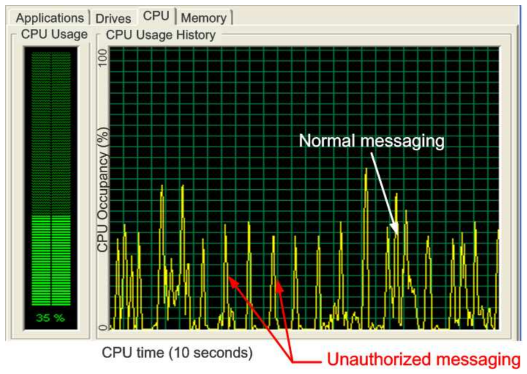


Figure 3. **Nokia 3230 (Symbian) compromised by malware adopting attack strategy II; average CPU occupancy exceeds 35%**

The average is around 5%. We note that as we use NFS root filesystem in the experiment, some of our measured performance data are much higher than that in desktop systems [16]. As a summary, using Linux kernel 2.6 as the platform OS, our access control-based defense is lightweight when compared with the baseline benchmark.

C. GTT-based Defense

To evaluate the effectiveness of the GTT-based protection scheme and test its overhead (e.g., CPU occupancy and memory usage) in real phone environments, we launched both forms of attacks on Symbian-based and Linux-based smartphones. The experiment result in Figure 3 demonstrates the system resource usages of a Nokia 3230 phone, which is compromised by a malware adopting attack strategy II to cause the same damages as CommWarrior [30] does. We can see that the malware automatically generates a malicious message in every 20 seconds (according to the timer it starts). This results in 35% CPU occupancy by average. We note that there is difference in CPU usage between a normal messaging and a malware-initiated messaging, because the latter is executed in the background hence does not involve Graphic User Interactions (GUIs) during its messaging process.

To find the overhead of the GTT-based scheme, we chose EZ-Gimpy [27], a CAPTCHA currently used by Yahoo! to screen out bots. As aforementioned before, we embedded GTT into the Symbian library function $CMMSClientMtm :: SendL()$, which is currently used in the messaging framework of both Nokia S60 phones and Sony-Ericsson UIQ phones. The experiment shows that our GTT-based scheme successfully identified and blocked the malware and thus prevented misuses of system resources. Our protection scheme reduces the average CPU occupancy to 6% or below. In addition, two GTT trials in the test incur low overhead on CPU and memory usages. Specifically, the maximum CPU occupancy is 5% and it only happens within less than one second, the memory space required for an EZ-GIMPY is in order of hundreds of KBytes. Another important thing is that each EZ-Gimpy takes a user 1.5$\sim$2.0 seconds to complete, hence it has a good time response compared with the time that a user spends in composing a message.

V. Related Work

Mulliner et al. [10] adopted a labelling technique to protect the phone interface against malware attacks which comes through the phone's PDA interface. Specifically, resources and codes are labeled based on the interfaces that they come from. A process can access a resource or invoke a code only when it has been labelled with the same label as the resource or code. or, if it is not labelled, it is labelled with the same label of the resource or code and then gets the access. Any process or resource created by a process is labelled with the same label as the creating process. The key difference between this and our access control–based scheme is that, our scheme is based on a well-studied TE policy model, which can be configured to implement different security requirements.

Bose et al. [8] proposed a proactive malware containment framework to quarantine suspected users. Cheng et al. [9] designed a collaborative virus detection and alert system named SmartSiren for securing smartphones. Smartsiren collects communication data from phones and performs joint analysis to detect abnormal behavior and alert users. Our work differs from these solutions in that we adopt system-level defenses to combat cellphone malware. Recently, Bose et al. [33] proposed adopting machine-learning techniques in cellphones to detect malware. Their method discriminates the malicious behaviors of malware from the normal behaviors of applications through training a classifier such as Support Vector Machines (SVMs). However, this training process requires input from known malware behavior, i.e., their approach still heavily replies on malware signatures. Similarly, Venugopal et al. [34] described a virus detection system for the Symbian platform which monitors the DLL functions used by applications. By using Bayesian decision theory and past virus samples, they try to check the behavior of applications to find matches with malicious activity. Anomaly-based IDS has also been proposed to detect malware that are energy-greedy [35]. Unlike the above schemes which are primarily intrusion detection based, our defenses (also in [36]) are more like intrusion prevention. It can not only block attacks at realtime, but also expects very low false positive and false negative rates because it does not rely on signatures.

Other related work include attacks on cellphone's batteries [37] and on cellular networks [18]. In [38], social network knowledge (the social ties among cellphone users reflected by their conversation frequency) is leveraged to rapidly distribute security patches [39] to cellphones under worm attacks.

VI. Conclusions and Future Work

Recent outbreaks of computer malware also remind us that devastating attacks in cellphones is approaching. Existing signature-based countermeasures and security updates are either non-realtime or heavily dependent on users' awareness. We have proposed two system-level defenses which consists of an access control-based scheme and a GTT-based scheme. We have showed through extensive cellphone experiments that these solutions are effective, lightweight, and easy to deploy. In spite of some remaining issues such as diversity of smartphone OS, new infection vehicles such as Wi-Fi, and potential OS kernel-level malware attacks, our solutions provide practical ways for containing malware within the context of mobile devices. Identifying compromised handsets from the network according to their traffic behavior by leveraging our previous techniques [40].

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by grants NSF-0643906 and NSF-0721579.

References

[1] "http://www.us-cert.gov/press_room/trendsandanalysisq108.pdf," .

[2] M. Hypponen, "State of cell phone malware in 2007, http://www.usenix.org/events/sec07/tech/hypponen.pdf," .

[3] E. Chien, "Security response: Symbos.skull, symantec, 2004," .

[4] P. Ferrie, P. Szor, R. Stanev, and R. Mouritzen, "Security response: Symbos.cabir," Symantec Corporation, 2004.

[5] E. Chien, "Security response: Symbos.mabir, symantec, 2005," .

[6] "http://www.idc.com/getdoc.jsp?containerid=206072," .

[7] "http://www.f-secure.com/v-descs/flexispy_a.shtml," .

[8] A. Bose and K. Shin, "Proactive security for mobile messaging networks," in *Proc. of WiSe*, 2006.

[9] J. Chen, S. Wongand, H. Yang, and S. Lu, "Smartsiren: Virus detection and alert for smartphones," in *MobiSys*, 2007.

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

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

[12] "http://en.wikipedia.org/wiki/stack_buffer_overflow," .

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

[14] T. Jaeger, R. Sailer, and U. Shankar, "Prima: Policy-reduced integrity measurement architecture," in *Proc. of SACMAT*, 2006.

[15] "http://www.virtuallogix.com/," .

[16] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings of USENIX Annual Technical Conference*, 2001.

[17] W. E. Boebert and R. Y. Kain, "A practical alternative to hierarchical integrity policies," in *Proceedings of the Eighth National Computer Security Conference*, 1985.

[18] W. Enck, P. Traynor, P. McDaniel, and T. La Porta, "Exploiting open functionality in sms-capable celluar networks," in *ACM CCS*, 2005.

[19] "http://www.3gpp.org/ftp/specs/archive," .

[20] "http://www.cse.psu.edu/˜ lxie/techrep/," .

[21] "http://trolltech.com/products/qtopia," .

[22] "http://trolltech.com/products/qtopia/qtopiainuse/qtopiadevices," .

[23] "http://www.pcmag.com/article2/0,4149,1306805,00.asp," .

[24] W. Morein, A. Stavrou, D. Cook, A. Keromytis, V. Misra, and D. Rubenstein, "Using graphic turing tests to counter automated ddos attack against web servers," in *ACM CCS*, 2003.

[25] L. Ahn, M. Blum, N. Hopper, and J. Langford, "CAPTCHA: Using Hard AI Problems for Security," in *EUROCRYPT*, 2003.

[26] "http://www.captcha.net/captchas/gimpy," .

[27] "http://www.cs.sfu.ca/˜ mori/research/gimpy," .

[28] "http://www.elinux.org/osk," .

[29] "http://www.cse.psu.edu/˜ lxie/snapshots/," .

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

[31] E. Chien, "Security response: Symbos.lasco.a, symantec, 2005," .

[32] "http://sourceforge.net/projects/lmbench/," .

[33] A. Bose and et al., "Behavioral detection of malware on mobile handsets," in *MobiSys*, 2008.

[34] Deepak Venugopal, Guoning Hu, and Nicoleta Roman, "Intelligent virus detection on mobile devices," in *PST '06: Proceedings of the 2006 International Conference on Privacy, Security and Trust*, 2006, pp. 1–4.

[35] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proc. of The International Conference on Mobile Systems, Applications, and Services*, 2008.

[36] L. Xie, H. Song, T. Jaeger, and S. Zhu, "Towards a systematic approach for cell-phone worm containment," in *in Proc. of WWW (poster paper)*, 2008.

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

[38] Z. Zhu, G. Cao, S. Zhu, S. Ranjan, and A. Nucci., "A social network based patching scheme for worm containment in cellular networks," in *Proceedings of IEEE Infocom*, 2009.

[39] L. Xie, H. Song, and S. Zhu, "On the effectiveness of internal patch dissemination against file-sharing worms," in *Proc. of ACNS*, 2008.

[40] L. Xie and S. Zhu, "Message dropping attacks in overlay networks: Attack dectection and attacker identification," in *Proc. of SecureComm*, 2006.