ROOTKITS ON SMART PHONES: ATTACKS, IMPLICATIONS, AND ENERGY-AWARE DEFENSE TECHNIQUES

BY JEFFREY EARL BICKFORD

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Vinod Ganapathy and Liviu Iftode

and approved by

_____

_____

_____

New Brunswick, New Jersey

January, 2012

ABSTRACT OF THE THESIS

Rootkits on Smart Phones: Attacks, Implications, and Energy-Aware Defense Techniques

by Jeffrey Earl Bickford

Thesis Director: Vinod Ganapathy and Liviu Iftode

Smart phones are increasingly being equipped with operating systems that compare in complexity with those on desktop computers. This trend makes smart phone operating systems vulnerable to many of the same threats as desktop operating systems.

In this dissertation, we focus on the threat posed by smart phone rootkits. Rootkits are malware that stealthily modify operating system code and data to achieve malicious goals, and have long been a problem for desktops. We use four example rootkits to show that smart phones are just as vulnerable to rootkits as desktop operating systems. However, the ubiquity of smart phones and the unique interfaces that they expose, such as voice, GPS and battery, make the social consequences of rootkits particularly devastating.

The rapid growth of mobile malware and the four rootkit attacks developed, necessitates the presence of robust malware detectors on mobile devices. However, running malware detectors on mobile devices may drain their battery, causing users to disable these protection mechanisms to save power. This dissertation studies the security versus energy tradeoffs for a particularly challenging class of malware detectors, namely rootkit detectors. We investigate the security versus energy tradeoffs along two axes: attack surface and malware scanning frequency, for both code and data based rootkit detectors. Our findings, based on a real implementation on a mobile handheld device, reveal that protecting against code-driven attacks

is relatively cheap, while protecting against all data-driven attacks is prohibitively expensive. Based on our findings, we determine a sweet spot in the security versus energy tradeoff, called the balanced profile, which protects a mobile device against a vast majority of known attacks, while consuming a limited amount of extra battery power.

Acknowledgements

The initial idea for this thesis was motivated by my advisors Vinod Ganapathy and Liviu Iftode. Developing rootkits for smart phones began as an undergraduate independent study and with their support, continued into graduate school. Vinod and Liviu's guidance and direction over the past several years has made this thesis possible and I thank them for all the help they have given me.

I would also like to thank Arati Baliga for her guidance in developing kernel-level rootkit attacks and support on the Gibraltar rootkit detector. Porting Gibraltar to the Xen hypervisor was an extension of Arati's dissertation and would not have been possible without her help.

Investigating the security versus energy tradeoff was done in collaboration with AT&T Research in Florham Park, NJ. I would thank my mentors H. Andrés Lagar-Cavilla and Alexander Varshavsky who both provided guidance throughout my summer internship and beyond. Andés' virtualization expertise and experience helped significantly while porting Gibraltar to the Xen hypervisor. He also provided code and implementation details for the Patagonix rootkit detector. Without his help, this work would not have been possible.

Lastly, I would like to thank my parents for supporting me throughout my time at Rutgers. Without them, none of this would be possible. Thank you!

Dedication

To my parents

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Thesis

Due to the energy constrained nature of mobile devices, energy-aware malware defense techniques must be developed.

In support of this thesis, this dissertation identifies a new class of threats for smart phones, namely *kernel-level rootkits*, previously used by attackers targeting desktop and server class machines. Due to the increasing complexity of smart phone operating systems, they are now vulnerable to these same types of attacks. We motivate the need for energy-aware defense techniques by developing four novel rootkit attacks for smart phones. Though previous rootkit detection techniques exist, we demonstrate that they cannot simply be ported to a mobile device environment due to their energy constraints. To solve this problem, this dissertation proposes a framework to study the security versus energy tradeoffs of malware defenses. We use this framework to develop energy-aware rootkit defenses for use on a mobile device.

1.2 Mobile Malware

We have come to rely on mobile devices as an integral part of our everyday lives. We entrust our smart phones, netbooks, and laptops with personal information, such as email, friend lists, current location, and passwords to online banking websites. The future holds an even greater role for mobile devices, *e.g.,* as interfaces for wireless payments [13] or smart home control [12]. Mobile devices are thus swiftly becoming prized bounties for malicious entities: while the quantity and diversity of mobile malware available today pales in comparison with

malware available for desktops, the incentives available to attackers point to a large and thriving future underground economy based on infected mobile devices. This has motivated recent research on both attacks against and defenses for mobile devices [19, 22, 26, 28, 35, 42, 46, 49].

Over the last several years, the decreasing cost of advanced computing and communication hardware has allowed mobile phones to evolve into general-purpose computing platforms. Over 115 million such *smart phones* were sold worldwide in 2007 [7] with 54.3 million smart phones sold in the first quarter of 2010 alone [2]. These phones are equipped with a rich set of hardware interfaces and application programs that let users interact better with the cyber and the physical worlds. For example, smart phones are often pre-installed with a number of applications, including clients for location-based services and general-purpose web browsers. These applications utilize hardware features such as GPS and enhanced network access via 3G or LTE. To support the increasing complexity of software and hardware on smart phones, smart phone operating systems have similarly evolved. Modern smart phones typically run complex operating systems, such as Linux, Windows Mobile, Android, iOS, and Symbian OS, which comprise tens of millions of lines of code.

The increasing complexity of smart phones has also increased their vulnerability to attacks. Recent years have witnessed the emergence of *mobile malware*, which are trojans, viruses, and worms that infect smart phones. For instance, F-Secure reported an almost 400% increase in mobile malware within a two year period from 2005-2007 [33]. This trend continues today: a recent presentation by Kapersky labs reports that about 25 new variants (on average) of mobile malware were released each month in the first five months of 2010 [40]. By 2011, McAfee reports that there are over 1300 unique mobile malware samples in the wild [37]. Mobile malware typically use many of the same attack vectors as do malware for traditional computing infrastructures, but often spread via interfaces and services unique to smart phones, including Bluetooth, SMS and MMS. The Cabir worm, for instance, exploited a vulnerability in the Bluetooth interface and replicated itself to other Bluetooth enabled phones. Today, malware is frequently packaged inside of popular apps and downloaded by users via various app stores. Mobile malware is typically used to send premium messages for financial profit or steal personal information stored on the device.

1.3 Rootkits

The term "rootkit" originally referred to a toolkit of techniques developed by attackers to conceal the presence of malicious software on a compromised system. During infection, rootkits typically require privileged access (*e.g.,* root privileges) to infect the operating system. Even on operating systems that do not run applications with root privileges, an attacker may exploit vulnerabilities in application programs, such as web browsers (*e.g.,* drive-by-download attacks) and the operating system, to obtain elevated privileges to install rootkits.

By compromising the operating system, rootkits may be used to hide malicious user space files and processes, install Trojan horses, and disable firewalls and virus scanners. Rootkits can achieve their malicious goals stealthily because they affect the operating system, which is typically considered the trusted computing base. The stealthy nature of rootkits allows them to retain long-term control over infected devices, and to serve as a stepping stone for other attacks such as key-loggers or backdoors. It is no surprise then that a 2006 study by McAfee Avert Labs [10] reported a 600% increase in the number of rootkits in the three year period from 2004-2006. The explosive growth of rootkits continues; McAfee's 2010 threat predictions report also contains several examples of rootkit-aided Trojan horses that were used to commit bank fraud [11].

As this dissertation will show, the increasing complexity of the hardware and software stack of mobile devices, coupled with the increasing economic value of personal data stored on mobile devices, point to an impending adoption of rootkits in the mobile malware arena. The recent development of proof-of-concept rootkits for Android-based phones [46] and the iPhone [42] only reinforces these predictions.

1.4 Summary of Thesis Contributions

This dissertation has two main contributions in the area of mobile device security, which were published in the *Proceedings of the 11th International Workshop on Mobile Computing Systems and Applications* [19] and *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* [20].

We show the feasibility of rootkit attacks on mobile devices and their social consequences [19]. We use four example rootkits to show that smart phones are just as vulnerable to rootkits as desktop operating systems. The ubiquity of smart phones and the unique interfaces that they expose, such as voice, GPS and battery, make the social consequences of rootkits particularly devastating. We have demoed these rootkit attacks to live audiences and through the media [21].

Due to the increasing threat of mobile malware and the social consequences of the rootkit attacks developed in the work above, we identify the necessity of malware detectors on mobile devices. Because running malware detectors on mobile devices may drain their battery, causing users to disable these protection mechanisms to save power, we study the security versus energy tradeoffs for mobile malware detection. Specifically, we study and optimize two rootkit detection systems in order to measure the feasibility of protecting mobile devices against this challenging class of attacks [20]. This work also provides a framework to develop a balanced security profile, which protects a mobile device against a vast majority of attacks, while consuming a limited amount of extra battery power.

1.5 Contributors to the Thesis

The following is a list of people who co-authored papers from which material was used in this dissertation. Ryan O'Hare helped develop code used in two out of four rootkit attacks. Specifically, he developed a function to decode SMS messages received by the device. The attacks were designed with the help of Arati Baliga, who also supplied code and support for the Gibraltar rootkit detection system. Investigating the security versus energy tradeoff was done in collaboration with Andrés Lagar-Cavilla and Alexander Varshavsky, both members of AT&T Research in Florham Park, NJ at the time. Andrés provided code and implementation details for the Patagonix system, which was used to measure the security versus energy tradeoff.

1.6 Organization of the Thesis

This dissertation is organized as follows. Chapter 2 describes four rootkit attacks for a smart phone device. We show how each attack contains social consequences and motivate the need to detect these types of attacks on mobile devices. Chapter 3 investigates the security versus

energy tradeoffs for two rootkit detectors to provide a framework for optimizing mobile malware defenses with energy in mind. In Chapter 4, we report measurements and results of our experiments based on this framework. Finally, Chapter 5 concludes the dissertation.

Chapter 2

Rootkits on Smart Phones: Attacks and Implications

The increasing complexity of smart phone operating systems makes them as vulnerable to rootkits as desktop operating systems are. However, these rootkits can potentially exploit interfaces and services unique to smart phones to compromise security in novel ways. In this chapter, we present four proof-of-concept rootkits that we developed to illustrate the threat that they pose to smart phones. They were implemented by the first two authors, with only a basic undergraduate-level knowledge of operating systems. Our test platform was a Neo Freerunner smart phone running the Openmoko Linux distribution [5]. We chose this platform because (a) Linux source code is freely available, thereby allowing us to study and modify its data structures at will; and (b) the Neo Freerunner allows for easy experimentation, *e.g.,* it allows end-users to re-flash the phone with newer versions of the operating system.

All our rootkits were developed as Linux kernel modules (LKM), which we installed into the operating system. However, during a real attack, we expect that these LKMs will be delivered via other mechanisms, *e.g.,* after an attacker has compromised a network-facing application or via a drive-by-download attack. Figure 2.1 presents the lines of code needed to implement each attack, and the size of the corresponding kernel module. This figure shows the relative ease with which rootkits can be developed. It also shows that the small size of kernel modules allows for easy delivery, even on bandwidth-constrained smart phones.

Although our implementation and discussion in this section are restricted to the Neo Freerunner platform, the attacks are broadly applicable to smart phones running different operating systems. For example, Android is a platform derived from Linux and can support loadable kernel modules; consequently, our proof-of-concept rootkits can potentially be modified to work on the Android platform (and the phones that run it, such as the Droid and the Nexus One). Since

Attack	LOC	Size of kernel module
GSM	116	92.8 KB
GPS	428	101.7 KB
Battery	134	87.2 KB
Call Forward	128	56.2 KB

Figure 2.1: Size of kernel modules that implement each of the four attacks

the attacks modify OS-specific data structures, they must be re-implemented for other platforms, such as Windows Mobile and Symbian OS; we expect that doing this will be relatively easy.

In the following four sections, we will describe in detail the four rootkits we developed. For each rootkit, we will present the goal of the attack, the attack description and its social impact.

2.1 Attack 1: Spying on Conversations via GSM

Goal

The goal of this attack is to allow a remote attacker to stealthily listen into or record confidential conversations using a victim's rootkit-infected smart phone.

Attack Description

The Freerunner phone is equipped with a GSM radio, which is connected via the serial bus and it is therefore available to applications as a serial device. During normal operation of the phone, user-space applications issue system calls to the kernel requesting services from the GSM device. The GSM device services the request allowing the application to access the telephony functionality provided by the device. GSM devices are controlled through series of commands, called AT (attention) commands, that let the kernel and user-space applications invoke specific GSM functions. For example, GSM devices support AT commands to dial a number, fetch SMS messages, and so on. To maliciously operate the GSM device, *e.g.,* to place a phone call to a remote attacker, the rootkit must therefore issue AT commands from within the kernel.

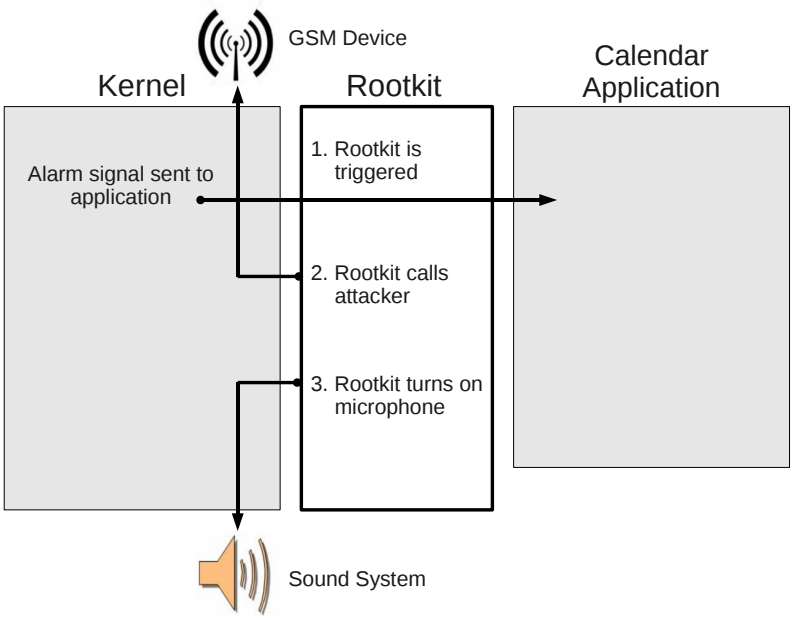


Figure 2.2: Sequence of steps followed in the conversation snooping attack

Most smart phones today contain calendar programs, which notify users when scheduled events occur. Our prototype rootkit operates by intercepting these notifications set by the user. As shown in Figure 2.2, a notification is displayed by a user-space program to notify the user of an impending meeting. The rootkit intercepts this notification and activates its malicious functionality. The attack code stealthily dials a phone number belonging to a remote attacker, who can then snoop or record confidential conversations of the victim. The phone number dialed by the rootkit can either be hard-coded into the rootkit, or delivered via an SMS message from the attacker, which the rootkit intercepts to obtain the attacker's phone number. Alternatively, the rootkit could be activated when the victim dials a number. The rootkit could then stealthily place a three-way call to the attacker's number, thereby allowing the attacker to record the phone conversation.

To trigger the rootkit, we used a simple alarm clock program to simulate calendar notifications on the Openmoko (we did so because the Openmoko phone does not have any released calendar programs). In an uninfected kernel, when an alarm is signaled, a specific message is

```
 1.   char *atcommand1 = "AT command1";
 2.   char *atcommand2 = "AT command2";
 3.   ...
 4.   mm_segment_t saved_fs = get_fs();
 5.   set_fs(KERNEL_DS);
 6.   fd = sys_open("/dev/ttySAC0", O_RDWR | O_NONBLOCK, 0);
 7.   sys_write(fd, atcommand1, sizeof(atcommand1));
 8.   sys_write(fd, atcommand2, sizeof(atcommand2));
 9.   ...
10.   sys_close(fd);
11.   set_fs(saved_fs);
```

Figure 2.3: Pseudocode of the conversation snooping attack

delivered via the `write` system call. In our "infected" kernel, the rootkit hooks the system call table and replaces the address of the `write` system call with the address of a malicious `write` function implemented in the rootkit. The goal of the malicious `write` function in our prototype rootkit is to check for the alarm notification in the `write` calls. Once an alarm message is identified, the malicious functionality is triggered.

When triggered, our rootkit places a phone call by emulating the functionality of user-space telephony applications. Typically, user-space applications (such as the Qtopia software stack [6], which ships with the Openmoko Linux distribution on the Freerunner phone) make calls by issuing a sequence of system calls to the kernel. Specifically, applications such as Qtopia use `write` system calls to issue AT commands to the GSM device (these commands are supplied as arguments to the `write` system call). The number to be dialed is located in the AT command.

Our prototype rootkit calls the attacker by issuing the same sequence of AT commands from within the kernel. We obtained the sequence of AT calls that must be issued to place a phone call by studying the Qtopia software stack. The AT commands issued by the rootkit activate the telephony subsystem and successfully establish a connection to the attacker's phone.

When a system call is issued, the Linux kernel first checks that the arguments to the call are within the virtual address-space of a user-space application. While this check is important when system calls are issued by user-space applications (*e.g.,* to ensure that an application cannot maliciously refer to kernel data in a system call argument), it will cause system calls issued by the kernel to fail. To issue the sequence of AT calls from kernel mode, the rootkit first modifies the boundaries of the data segment to point to kernel-addressable using the `get_fs`/`set_fs`

call sequence. This sequence allows us to issue system calls (such as `sys_open`, `sys_write` and `sys_close`) from within the kernel. The rootkit simply opens the GSM device, places a sequence of `write` system calls with appropriate arguments (*e.g.,* AT commands) and closes the device. Figure 2.3 shows the pseudocode of this process. The rootkit first prepares a set of AT commands (lines 1-3), modifies the boundaries of the data segment (lines 4-5), and writes AT commands to the GSM device (lines 6-10).

Though the AT commands successfully establish a connection to the attacker's phone, the microphone must still be activated. The microphone is controlled via the sound subsystem of the kernel. On the Openmoko, the sound system is controlled by the Advanced Linux Sound Architecture (ALSA) [1]. The user-space program, `alsactl`, is used to control settings in the ALSA sound card drivers. To activate the microphone, we programmed the rootkit to issue the following command from kernel mode:

`alsactl -f /usr/share/OpenMoko/scenarios/gsmhandset.staterestore`.

Social Impact

Snooping on confidential conversations has severe social impact because most users tend to keep their mobile phones in their proximity and powered-on most of the time. Rootkits operate stealthily, and as a result, end users may not even be aware that their phones are infected. Consequently, an attacker can listen-in on several conversations, which violates user privacy, ranging from those that result in embarrassing social situations to leaks of sensitive information. For example, an attack that records the conversations at a corporate board meeting can potentially compromise corporate trade secrets and business reports to competitors. Similarly, several automated phone-based services often require a user to enter (via voice or key presses) PIN numbers or passwords before routing the call to a human operator; an attacker snooping on such calls may financially benefit from such information.

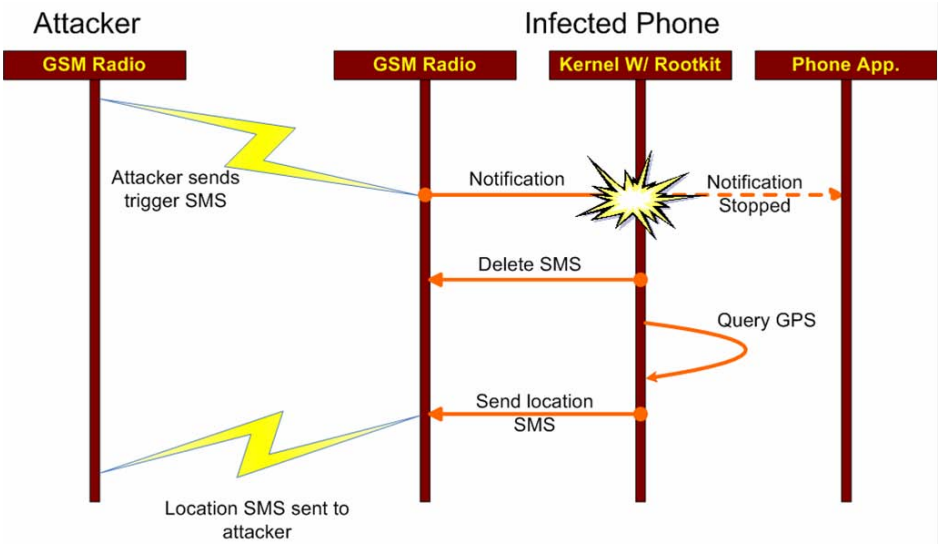


Figure 2.4: Sequence of steps followed in the location privacy attack

2.2 Attack 2: Compromising Location Privacy using GPS

Goal

The goal of this attack is to compromise a victim's location privacy by ordering the victim's rootkit-infected smart phone to send to the remote attacker a text message with victim's current location (obtained via GPS).

Attack Description

As with the GSM device, the GPS device is also a serial device. The kernel maintains a list of all serial devices installed on the system. A rootkit can easily locate the GPS device. Every serial device contains a buffer in which the corresponding device stores all outgoing data until it is read by a user-space application. Our prototype rootkit uses this buffer to read information before it is accessed by user-space applications. This allows us to monitor and suppress incoming SMS messages and also query the GPS for location information.

A rootkit that compromises location privacy as described above must implement three mechanisms. First, it must be able to intercept incoming text messages, and determine whether a text message is a query from a remote attacker on the victim's current location. Second, the

rootkit must be able to extract location information from the GPS receiver. Last, it must generate a text message with the victim's current location, and send this information to the attacker. An overview of this attack is shown in Figure 2.4.

Our prototype rootkit intercepts text messages by monitoring and changing data in the GSM device buffer. To monitor the GSM buffer, we hook the kernel's `read` and `write` system calls. This is achieved by modifying the corresponding entries in the system call table to point to rootkit code. Consequently, the rootkit identifies when user-space applications are accessing the GSM device by checking the file descriptor that is passed into `read` and `write`. When this occurs, our rootkit scans the GSM for certain incoming AT commands.

When an incoming message arrives, the rootkit will check whether this is a query from the attacker. This is done by sending the AT command to read messages. An attacker's message will be a certain phrase or set of words that the rootkit can check for. If the message is a query from the attacker, the rootkit's malicious operation is executed. The rootkit can be written to enable different functionality for different messages.[1] The attacker can also enable/disable the rootkit's malicious functionality via text messages, in effect allowing the attacker to remotely control the rootkit.

Importantly, the rootkit, must also suppress the notification of the attacker's message, to ensure that the user does not learn about it. The rootkit deletes the message from the SIM card by sending another AT command to the GSM device.

New incoming text message notifications are caused by the "`+CMTI=simindex`" command, where "`simindex`" is the serial number of the text message. To suppress a new message notification we find the `+CMTI` substring in the GSM buffer. It is important to note that information in the buffer has not yet reached user space. Upon finding this substring, the rootkit writes random characters to this location to suppress the new message notification.

To determine whether it contains an instruction from the attacker (in some pre-determined format), we must parse the message at location `simindex`. A command `AT+CMGL=4` is issued to the GSM device to list all messages currently on the SIM card. Each message contains a

[1]This mechanism is also useful in Attack 1—instead of hard-coding the number that the rootkit must dial, an attacker can transmit the number that the infected phone must dial via a text message. We can also trigger a phone call via a text message instead of a calendar notification.

status bit that determines whether the message is new or has already been read. The attack code finds the message matching the index number and marks it as read in order to hide from user space applications. The text of the message is parsed and decoded. The attacker encodes the message in a certain predefined format in order to trigger malicious functionality. If the message does not belong to the attacker, the rootkit code changes the status bit of this message back to *unread*, so user space applications can process this message as usual.

Once the rootkit intercepts a message to query a user's current location, it attempts to obtain location information from the GPS device. As before, the rootkit can easily obtain location information from the buffer of the GPS device. The rootkit can obtain location information even if the user has disabled the GPS. This is because the rootkit operates in kernel mode, and can therefore enable the device to obtain location information, and disable the device once it has this information. If the user checks to see if the GPS is enabled during this time, it will appear that the GPS device is off.

Having obtained location information, the rootkit constructs a text message and sends the message by sending an AT command to the GSM device. The attacker will now receive a user's current location.

Social Impact

Protecting location privacy is an important problem that has received considerable recent attention in the research community. By compromising the kernel to obtain user location via GPS, this rootkit defeats most existing defenses to protect location privacy. Further, the attack is stealthy. Text messages received from and sent to the attacker are not displayed immediately to the victim. The only visible trace of the attack is the record of text messages sent by the victim's phone, as recorded by the service provider.

2.3 Attack 3: Denial of Service via Battery Exhaustion

Goal

This attack exploits power-intensive smart phone services, such as GPS and Bluetooth, to exhaust the battery on the phone. This rootkit was motivated by and is similar in its intent to a

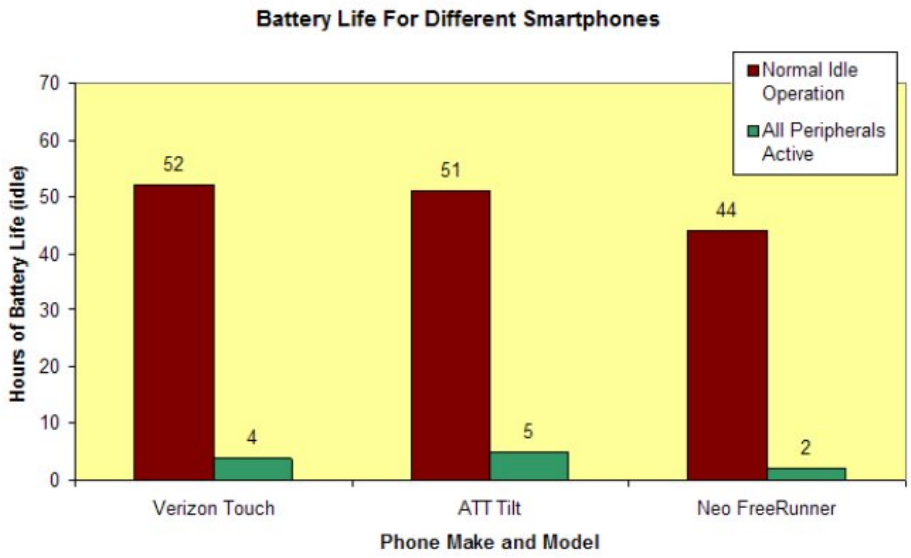


Figure 2.5: Battery life study for battery exhaustion attack

previously proposed attack that stealthily drains a smart phone's battery by exploiting bugs in the MMS interface [50]. However, the key difference is that the rootkit achieves this goal by directly modifying the smart phone's operating system.

Attack Description

The GPS and Bluetooth devices can be toggled on and off by writing a "1" or a "0," respectively, to their corresponding power device files. The rootkit therefore turns on the GPS and Bluetooth devices by writing a "1" to their corresponding power device files. To remain stealthy, the rootkit ensures that the original state of these devices is displayed when a user attempts to view their status. Most users typically turn these devices off when they are not in active use because they are power-intensive.

When a user checks the status of a GPS or Bluetooth device, the user-space application checks the power device file for a "1" or "0". To do this, it calls the `open` system call on the file and then reads it. The rootkit monitors the `open` calls by overwriting kernel function pointer for the `open` system call in the system call table, making it point to rootkit code. When an `open` system call is executed, the rootkit examines if the file being opened corresponds to the power

device files of the GPS or Bluetooth devices. If so, it ensures that the original states of the devices are displayed to the user. The rootkit continuously checks the status of these devices; if the devices are turned off by the user, the rootkit turns them back on. Consequently, the devices are always on, except when the user actively queries the status of these devices.

Social Impact

This attack quickly depletes the battery on the smart phone. In our experiments, the rootkit depleted the battery of a fully charged and infected Neo Freerunner phone in approximately two hours (the phone was not in active use for the duration of this experiment). In contrast, the battery life of an uninfected phone running the same services as the infected phone was approximately 44 hours (see Figure 2.5). We also simulated the effect of such a rootkit on the Verizon Touch and ATT Tilt phones by powering their GPS and Bluetooth devices. In both cases, battery lifetime reduced almost ten-fold. Because users have come to rely on their phones in emergency situations, this attack results in denial of service when a user needs his/her phone the most.

Although our prototype rootkit employs mechanisms to hide itself from an end user, this attack is less stealthy than Attacks 1 and 2. For example, a user with access to other Bluetooth-enabled devices may notice his smart phone is "discoverable," causing him to suspect foul play. Nevertheless, we hypothesize that the vast majority of users will suspect that their phone's battery is defective and replace the phone or its battery.

2.4 Attack 4: Identity Theft via Call Forwarding

Goal

In this attack, a rootkit forwards a trusted phone call to an attacker, allowing the attacker to pose as a trusted party and coerce a user give away personal information.

Attack Description

Smart phone user's inherently trust that when they dial a specific number, they will be connected to the proper party. For example, when a bank customer calls their bank's number, the customer

observes the number dialing on his/her screen and assumes a call is being made to their bank. When connected, the bank employee verifies the credentials of the customer by asking secret security questions or asking for personal information. Personal information such as a mother's maiden name or social security number is frequently used to steal a person's identity. This common situation has a security flaw since a customer never validates the bank employee they are speaking to.

In this attack, when a user calls a specific number (a bank's phone number), we forward this call to the attacker without changing the number displayed on the screen. From here, the attacker can pose as the assumed recipient and retrieve personal information from the user. The user will never know he/she has called a wrong number unless they check for this on their phone bill at the end of the month.

Our prototype rootkit must trigger malicious functionality when a user dials a specific phone number. Like in previous rootkits, we hook the write system call in order to monitor the AT commands sent from user space. When a user dials a phone number, `ATD` commands are sent via write system calls. We monitor all `ATD` commands waiting for our hard-coded number (i.e., the bank's number). When the user dials the number, the rootkit changes the number passed to the attacker's number. To do this, we modify the buffer passed to the system call following which a call is placed to the attacker.

Once a call is initiated, the screen displays the number a user is trying to dial. Since we do not want the user to know the call is being forwarded, the rootkit must display the original number the user tried to dial. By studying the system calls during a phone call, we discovered that the number displayed on the screen is sent through the write system call. After the rootkit modifies the phone number, it awaits a specific `CPI` call that is used to display the number on the screen. When found, the rootkit modifies the number located in the buffer to the number originally dialed by the user. The call is placed to the attacker and the phone number displayed to the user is the original number.

Social Impact

Once a call is forwarded, an attacker can pose as a trusted service employee without the knowledge of the smart phone user. Since it is common practice is to verify the credentials of the bank

customer, but not the bank employee, the attacker can ask for personal information which the user willingly gives. The attacker can then use this information for financial gain by stealing a person's identity.

2.5 Rootkit Delivery and Persistence

To effectively infect a smart phone using the rootkits discussed above, attackers must also develop techniques to deliver rootkits and ensure that their functionality persists on the phone for an extended period of time.

2.5.1 Delivery

Rootkits can be delivered to smart phones using many of the same techniques as used for malware delivery on desktop machines. A study by F-Secure showed that nearly 79.8% of mobile phones infections in 2007 were as a result of content downloaded from malicious websites. Bluetooth connections and text messages were among the other major contributors to malware delivery on smart phones. Rootkits can also be delivered via email attachments, spam, illegal content obtained from peer-to-peer applications, or by exploiting vulnerabilities in existing applications.

The Neo Freerunner phone used in our experiments ran the Openmoko Linux distribution, which directly executes applications with root privileges. Therefore, unsafe content downloaded on this phone automatically obtains root privileges. Smart phone operating systems that do not run applications with root privileges can also have vulnerabilities (just as in desktop machines). Such vulnerabilities are not uncommon even in carefully engineered systems. For example, a recent vulnerability in Google's Android platform allowed command-line instructions to execute with root privileges [3]. Though root exploits are typically developed for users to gain full control of their own devices, there is a recent trend by malware authors to package root exploits within malicious apps [52, 57–60]. Once an app is downloaded and executed, the application exploits a vulnerability, gains root access on the device, and then can install a rootkit via a malicious kernel module, to complete the attack.

2.5.2 Persistence

To be effective, rootkits must retain long-term control over infected machines. Rootkits typically achieve this goal by replacing critical operating system modules, such as device drivers, with infected versions. While this approach ensures that the rootkit will get control even if the operating system is rebooted, it also has the disadvantage of leaving a disk footprint, which can be detected by malware scanners.

Rootkits can avoid detection by directly modifying the contents of kernel memory, thereby avoiding a disk footprint. Although such rootkits are disabled when the operating system is rebooted, they can still retain long-term control over server-class machines, which are rarely rebooted. However, this technique is not as effective on smart phones, because phones are powered off more often (or may die because the battery runs out of charge). Consequently, rootkits that directly modify kernel memory can only persist on smart phones for a few days. In such cases, an attacker can re-infect the phone. For example, a rootkit that spreads via Bluetooth can re-infect victims in the vicinity of an infected phone.

However, the social consequences of smart phone rootkits mean that they can seriously affect end-user security even if they are effective only for short periods of time.

2.6 Summary

In this chapter, we demonstrated four proof of concept rootkit attacks. Each attack has drastic social consequences, from compromising a user's privacy by sniffing their phone conversations and GPS location to forwarding phone calls and draining a user's battery. Because rootkits compromise the integrity of the operating system, they are stealthy and undetectable by current anti-malware solutions deployed on smart phones. For this reason, this chapter motivates the need for techniques to detect these types of attacks on mobile devices.

Chapter 3

Energy-Aware Rootkit Defenses for Mobile Devices

Conventional wisdom holds that executing malware detectors on resource-constrained mobile devices will drain their battery [44], causing users to disable malware detection to extend battery life, and in turn exposing them to greater risk of infection. In this chapter, we present a framework to quantify the degree of security being traded off when prolonging battery life, and the ways in which such tradeoffs can be implemented. Specifically, we study security tradeoffs along two axes: (1) the surface of attacks that the malware detector will cover, and (2) the frequency with which the malware detector will be invoked.

Some emerging proposals for malware detection have sought to sidestep the energy constraints that we formalize and quantify in this study using *offloaded architectures* [14,25,45,49], in which the malware detector itself executes on a well-provisioned server and monitors mobile devices. Unfortunately, malware detection offload either incurs significant power expenditures [49] due to data upload, or has limited effectiveness because it is best suited to traditional signature-based scanning. Such signature scanning is easily defeated with encryption, polymorphism and other stealth techniques. For this reason, there is growing consensus that signature-based scanning must be supplemented with powerful host-based agents that, for example, employ behavior-based detection algorithms [24]. Host-based detectors execute on and share resources such as CPU time and battery with the host device, thereby making the *security versus energy tradeoff* germane to the design of such detectors.

In this chapter, we focus on security versus energy tradeoffs for host-based *rootkit detection*. As previously mentioned, rootkits are a class of malware that infect the code and data of the operating system (OS) kernel. By infecting the kernel itself, they gain control over the layer that is traditionally considered the trusted computing base (TCB) on most systems. Rootkits can therefore be used to evade user-space malware detectors (including most commercial solutions

that employ signature-based scanning). Further, rootkits enable other attacks by hiding malicious processes, and allow attackers to stealthily retain long-term control over infected devices. The previous chapter has argued that the increasing complexity of mobile device OSes offers a vast attack surface of code and data that makes rootkits a realistic threat. As a consequence of the variety of ways in which a kernel can be exploited, and rootkit detection implemented, we show that rootkit detectors can be *modulated* to explore a rich space of configuration options. Varying these configurations allows us to explore, in a general manner, the tradeoff between the security provided by the detection agent and the energy consumption of the host.

3.1 Related Work

In this section, we discuss prior work on detecting malware on resource-constrained mobile devices. Although these works have developed new detection approaches tailored for mobile devices, some of which are resource-aware, none have quantified the security versus energy tradeoff.

3.1.1 Offloading detection.

As discussed in the introduction of Chapter 3, one way to sidestep the security versus energy tradeoff for detecting certain kinds of malware is to offload detection to a well-provisioned machine. Maui [25] and CloneCloud [14] approach general cloud offloading, while Paranoid Android [49] focuses on security. The latter performs user-space operation record and replay, at the granularity of system calls and signals. Replay on well-provisioned servers allowed offloading of security checks, as they are executed on a conceptually identical environment. However, host-based operation record, and the uploading of these operations to a server, resulted in an energy overhead of 30%.

3.1.2 Collaborative and behavior-based detection.

In keeping with the recent interest on behavior-based techniques for malware detection, researchers have investigated techniques tailored for mobile phones. The work of Bose *et al.* [22]

and Kim *et al.* [35] are two such examples, which use a host-based agent that observes activities on the phone and reports anomalies such as forwarding SMS messages to external phone numbers or the deletion of important system files. The work of Kim *et al.* is an interesting complement to the security versus energy tradeoff studied in our work. They proposed a security tool that generates power signatures for applications running on a handheld device to detect energy-greedy anomalies caused by mobile malware such as Bluetooth worms.

3.1.3 Smart phone app security.

Recent research advocates for *preemptive* approaches that aid distributors verify the security of smart phone apps before they are deployed. Although not a panacea to the security problem, preemptive certification as implemented by Kirin [26] and ScanDroid [28] can detect and discard a large fraction of malware before it reaches the phone. Such techniques can complement host-based detectors, which can run using conservative security versus energy profiles when "trusted" apps are downloaded and executed.

3.2 Attack vectors

Rootkits remain stealthy by compromising the integrity of entities that belong to the trusted computing base (TCB) of victim devices. On most devices, these include OS code and data, as well as key user-space processes and files. We briefly survey the evolution of rootkit attack vectors, from those that are easiest to detect to those that are most challenging to detect.

- *System utilities.* Early rootkits attempted to hide the presence of malicious processes by compromising system utilities that are used for diagnostics. For example, a rootkit that replaces the `ls` and `ps` binaries with trojaned versions can hide the presence of malicious files and processes. Such rootkits are easy to detect by an uncompromised TCB that certifies the integrity of user-space utilities with checksums.

- *Kernel code.* The next generation of rootkits attempted to evade detection by affecting the integrity of kernel code. Such corruption is most usually achieved by coercing the system into loading malicious kernel modules. Once a rootkit has gained kernel execution privileges, it can mislead all detection attempts from user- or kernel-space. Successful

detection of such rootkits is achieved instead by components located outside the control of the infected kernel. The two main approaches involve use of external hardware which scans the kernel memory using DMA (*e.g.,* [15, 16, 34, 61]), or introspection from the vantage point of a different virtual machine (*e.g.,* [29, 38, 47]).

- *Kernel data structures.* A large majority of rootkits in use today corrupt *kernel control data* by modifying function pointers in data structures such as the system call table or the interrupt descriptor table. This attack technique allows rootkits to redirect control to attacker code when the kernel is invoked. For example, the Adore rootkit [39] hides user-space processes from reporting tools like `ps`, by hijacking the function pointer for `readdir()` in the root inode of the `/proc` file system. More recently, research has shown that attacks against *non-control kernel data* are realistic threats [17, 23]. For example, a rootkit can subvert key cryptographic routines by affecting kernel parameters controlling pseudo-random number generation.

3.3 Defenses

In this section, we discuss the design of two prior techniques to rootkit detection. The two techniques are representative of the algorithms used in most rootkit detectors, and complement each other. The first technique, based on Patagonix [38], detects rootkits by monitoring code integrity; the second technique, based on Gibraltar [15, 16], monitors kernel data integrity.

Both tools use hypervisors to achieve isolation from the kernels they monitor. The hypervisor guarantees isolation between a monitored system (the *untrusted guest domain*) and the monitoring tool (the *trusted domain*); functional correctness of such guarantees has been formally proven [36]. The hypervisor and the trusted domain therefore comprise the TCB of the system. When the trusted domain detects a compromise, the TCB is capable of taking over the UI to alert the user and provide containment options – the specifics of this mechanism are outside the scope of this dissertation.

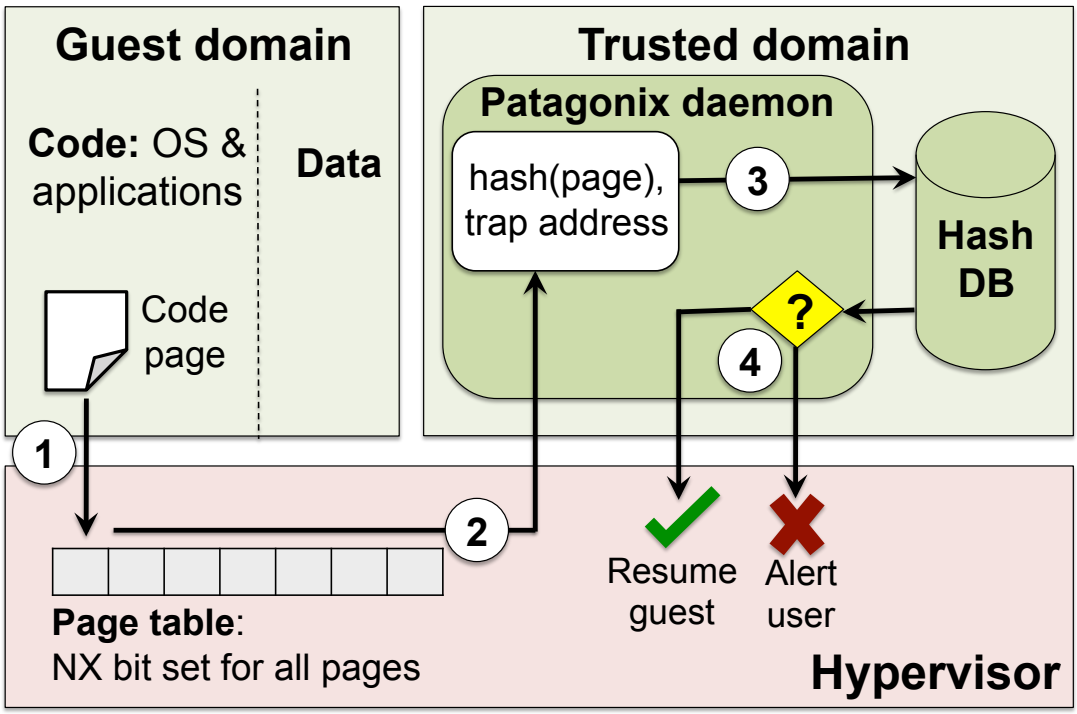


Figure 3.1: Patagonix architecture: checking code integrity (subsection 3.3.1)

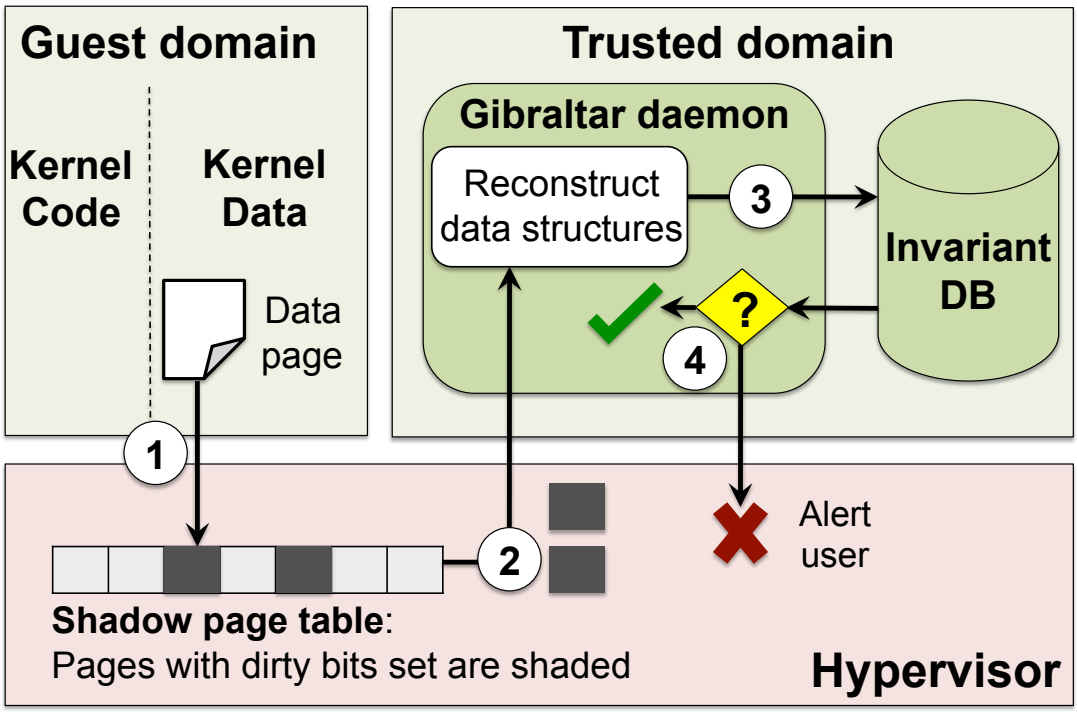


Figure 3.2: Gibraltar architecture: checking data integrity (subsection 3.3.2)

3.3.1 Checking code integrity

Patagonix [38] is a rootkit detection tool whose design typifies that of most code integrity monitoring systems. It provides mechanisms to ensure that all code executing on a system belongs to a whitelist of pre-approved code. Rootkits that modify system utilities or kernel code can be detected if the modified code is not in the whitelist. Patagonix can also detect certain data-modifying rootkits, *e.g.,* those that modify kernel data pages that should not be modified during normal operation. Figure 3.1 presents the design of Patagonix.

When a code page in the guest is first scheduled for execution, it results in a trap to the hypervisor and suspends the guest. Next, the hypervisor forwards this page to the Patagonix daemon in order to hash the page and authorize it. If the execution of the code page is authorized, Patagonix informs the hypervisor, which resumes execution of the guest; otherwise, Patagonix raises an alert.

Patagonix uses the capabilities of the hypervisor and the non-executable (NX) page table bit to detect and identify all executing code, including kernel code, system utilities, and other user processes. It modifies the code in the hypervisor to first set the NX-bit on all pages in the guest domain. When a page is first scheduled for execution, the NX bit causes a processor fault. The hypervisor receives the fault, pauses the guest domain and places information about the fault in a shared page that can be accessed by the Patagonix daemon executing in the trusted domain. The daemon hashes and compares the executing code to a whitelist of known software, comprised of the hashes of all approved code pages.

Patagonix enforces the **W⊗X** principle: pages are either modifiable, or executable. The hypervisor manipulates permission bits to enforce mutual exclusion between the two states. Pages will thus always be re-checked after modification. However, for code pages that are kept resident in the system and never change, Patagonix will not need to perform any further work. Thus, beyond an initial bootstrapping phase, the kernel working set and long-lived processes represent no additional work for Patagonix.

Patagonix uses optimizations to ensure fast verification of code pages. It remembers pages of code that have been blessed and have not changed. Thus, short-lived but recurring processes (*e.g.,*`grep`) will result in hypervisor work as new page tables are created, but no daemon work,

due to reuse of resident unmodified code pages. Patagonix knows the entry point of each binary in its whitelist – the first trap on a new binary should match an entry point in the whitelist. For approved binaries, it stores the associated address space (defined by the base address of the current page table) and the segment of the address space the binary occupies: pages within the same segment should only match pages of the same binary.

Though Patagonix is not representative of *all* code-integrity monitoring systems, its design is similar to several state-of-the art rootkit detection tools that have recently been proposed in the research literature. For example, NICKLE [51] and SecVisor [53] are similar in overall design to Patagonix. Grace *et al.*'s paper on commodity operating system protection [30] implements a subset of techniques used by Patagonix [55]. Our results on security versus energy tradeoffs for Patagonix will therefore also be applicable to these tools.

3.3.2 Checking data integrity

Rootkits that modify arbitrary kernel data structures are challenging to detect because of two reasons. First, the kernel manages several thousand heterogeneous data structures, thereby providing a vast attack surface. Second, unlike code, kernel data is routinely modified during the course of normal execution. Distinguishing benign modifications from malicious ones requires intricate specifications of data structure integrity. In this section, we describe Gibraltar [15, 16], a tool that monitors the integrity of kernel data structures to detect malicious changes.

Figure 3.2 shows the design of Gibraltar: a daemon executes on the trusted domain, and periodically fetches data pages from the untrusted guest kernel. The daemon reconstructs kernel data structures in a manner akin to a garbage collector. It starts at a set of kernel *root symbols* whose memory locations are fixed. Using the OS type definitions, it identifies pointers in these root symbols, and recursively fetches more pages that contain data structures referenced by these pointers.

Once data structures have been reconstructed, *data structure invariants* that specify kernel integrity constraints are verified. Some invariants are simple to verify: the values of function pointers must be addresses of known functions; the entries of the system call table should

remain constant during the execution of the kernel. Other more complex invariants span sophisticated data structures, *e.g.,* each process that is scheduled for execution must have an entry in the linked list of active processes on the system.

Data structure invariants can be specified by domain experts [48], but this approach can be labor-intensive. Instead, Gibraltar leverages the observation that a large number of data structure invariants can be automatically inferred by observing the execution of an uninfected kernel. Such inference is performed during a controlled *training* phase, when a clean OS executes several benign workloads. Prior work [15, 16] shows that high-quality invariants can be obtained with a relatively short training phase using the Daikon invariant inference tool [27].

Rootkits that affect kernel data integrity (and the corresponding detection tools) are a relatively recent development in contrast to rootkits that affect code integrity. The overall design of Gibraltar substantially resembles those of other data integrity monitoring tools, such as SBCFI [47] and Petroni *et al.*'s specification-based rootkit detection architecture [48]. HookSafe [56] prevents data-oriented rootkits (whereas Gibraltar can only detect them), but only protects a proper subset of Gibraltar's detection space. Other systems that detect rootkits by checking data invariants, such as OSck [31] and Co-Pilot [34], check for simpler invariants and thus may miss rootkits that Gibraltar can detect.

3.3.3 Shadow page table optimization

Gibraltar ran the daemon on a physically isolated machine and fetched memory pages from the monitored machine via DMA (using an intelligent NIC [15, 16]). For the study in this chapter, we adapted Gibraltar to execute on a hypervisor, which allowed us to implement novel performance optimizations. Notably, we implemented a *shadow page table optimization* that allows the Gibraltar daemon to focus the application of integrity constraints on just those data pages that were modified by the guest. This optimization relies on the use of shadow page tables by modern hypervisors, which grant to the TCB fine-grained control over the permission bits of virtual-to-physical memory translation. In particular, they can be used to cause faults on the first attempt to modify a page. The hypervisor catches these faults and records them in a "log-dirty" bitmap. The Gibraltar daemon consults this bitmap and only focuses on pages

whose dirty bits are set, and are known to contain data-structures of interest subject to integrity constraints.

The shadow page table optimization has a substantial effect on the number of checks Gibraltar has to perform. In experiments using the lmbench [41] workload executing for 144 seconds, 25 rounds of checks are performed by the optimized version of Gibraltar, as opposed to 5. By avoiding unnecessary checks to unmodified data, Gibraltar asserts the integrity of the kernel data structures 5 times more frequently, for the same power-budget and the same length of a user workload. We observed similar benefits in the other workloads employed in this work.

3.4 The Security/Energy Tradeoff

Security mechanisms have traditionally focused on well-provisioned computers such as heavy-duty servers or user desktops. Mobile devices present a fundamental departure from these classes of machines because they are critically resource-constrained. While advances throughout the last decade in mobile processor, GPU and wireless capabilities have been staggering, the hard fact is that mobile devices utilize batteries with a limited amount of stored power.

In this context, some fundamental tenets of security mechanism design need to be reconsidered. Without the limit of resource constraints, security mechanisms will check *everything they can, all the time*. In a mobile device, aggressively performing checks on large sets of security targets will inexorably lead to resource exhaustion and the inability to carry on useful tasks. Arguably, a certain amount of engineering could be added to any given security mechanism to make it marginally more efficient in terms of resource usage. We have just shown one such example with our shadow page table-based optimizations for Gibraltar. But we counter-argue that nothing short of a fundamental transformation will make security monitors palatable for mobile environments because energy must be a core consideration when designing tools for such environments.

The primary contribution of our work is in acknowledging that security needs to be traded off for battery lifetime in a mobile device, and in providing a framework to classify the choices a designer will face when modulating her security mechanism for a battery-constrained environment. Furthermore, we provide means for measuring the amount of security being traded

off. To the best of our knowledge, neither such a framework nor such metrics were deemed necessary before the widespread adoption of smart phones and other mobile devices.

3.5 What to check and when to check it

Energy-oblivious security mechanisms will check everything they can as frequently as they can. In a mobile setting, one must decide upon the attack surface to monitor (*i.e.,* what to check) and the frequency with which to perform monitoring (*i.e.,* when to check). These two factors must be incorporated as design parameters of the security mechanism itself to allow the mobile device to flexibly navigate the security versus energy tradeoff. We apply these concepts to the two rootkit detection systems discussed in the previous section.

3.5.1 What to check?

Operating system kernels provide a vast attack surface and a rootkit detection mechanism that monitors the entire attack surface will soon exhaust the mobile phone's battery. Both Patagonix and Gibraltar can be configured to check various subsets of the attack surface.

The Patagonix system can be configured to check (a) only the execution of kernel code pages; (b) kernel code pages and the execution of key binaries, such as root processes; and (c) kernel code, root processes, and selected kernel data structures, such as the system call table. Option (c) provides the highest security, whereas option (a) is the most efficient. The Patagonix daemon can trivially differentiate between kernel and user-space code due to the virtual addresses used: in all commodity OSes the kernel resides, on all address spaces, in a high band of virtual addresses. Root processes are manually classified and tagged in the whitelist; this does not prevent Patagonix from checking libraries linked in the address space of a root process (*e.g.,* OpenSSL for sshd).

Gibraltar also offers a wide variety of configurations, ranging from checks on selected kernel data structures, such as those that store control data (including function pointers), to checks on all kernel data structures. In between these two extremes, we can tune Gibraltar to check additional classes of data structures: static data; the process list and runnable queue, which are

a common target of attacks that hide the existence of a malicious user-space process [39]; all linked lists beyond the previous two; and more.

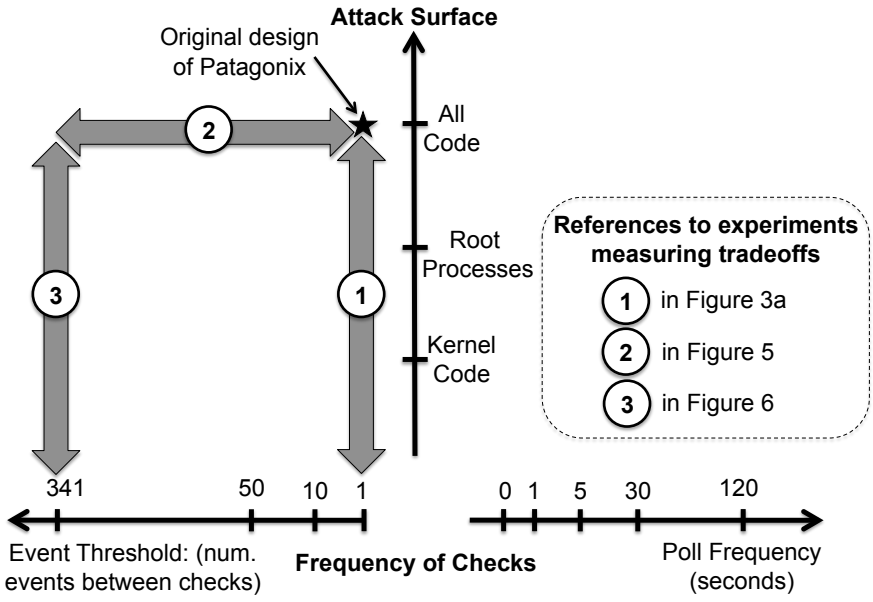
3.5.2 When to check?

Independently of the size of the attack surface being checked, one must tune how often to perform the checks. The design space for the frequency of checks ranges from an approach that uses periodic polling (where the period is configurable) to one that uses event-based or interrupt-based notifications to trigger the rootkit detector. Choosing the appropriate approach is a fairly well-understood dichotomy prevalent in systems design. With proper hardware support, one can implement event-based checks relatively efficiently, preventing the use of busy loops that burn too much CPU, or sleep timers that ignore momentous events.

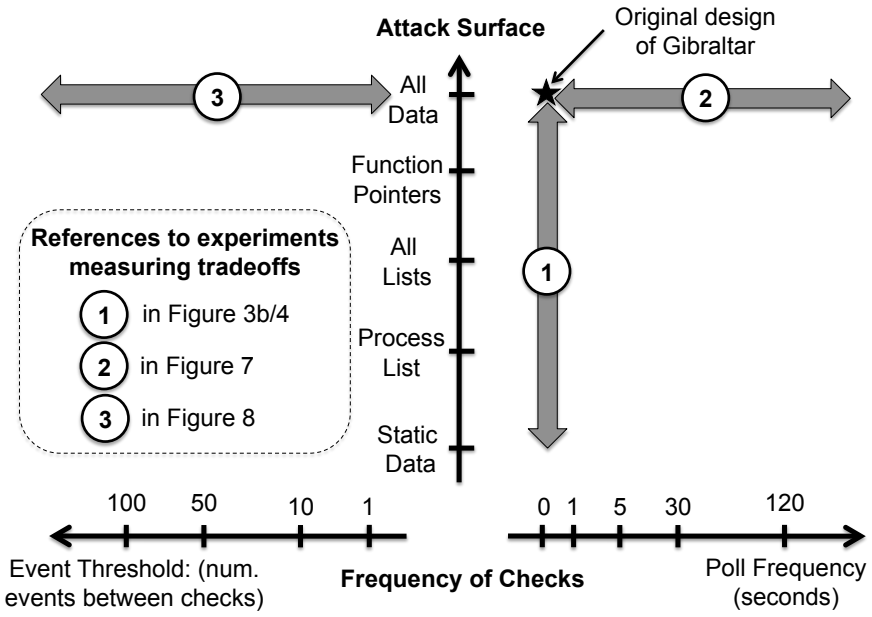
The original design of Patagonix uses an event-based approach to pause the guest domain each time a new page is scheduled for execution and check the page. However, Patagonix can also be modified to batch and perform these checks *en masse*. The former option detects and prevents malicious code execution in an online fashion, but may frequently pause the guest domain. In contrast, the latter option may detect malicious code only after it has executed on the system, but is likely to be more efficient.

The Gibraltar daemon traverses the guest OS's data pages in rounds, pausing for an interval of time after each round. During this interval, Gibraltar does not scan the kernel's data structures. The frequency of this traversal impacts energy efficiency and security. Frequent traversal minimizes the vulnerability of the guest operating system, while infrequent traversal conserves energy. The original version of Gibraltar has a T of zero as it continuously scanned all kernel pages: once the daemon completed traversal of all relevant kernel data structures, it immediately started a new round of traversals.

Our implementation of Gibraltar for mobile devices also incorporates an event-based mechanism in which the hypervisor interrupts the Gibraltar daemon when the guest has modified a certain number of pages, N, so that the data structure checks for these pages can be batched and performed *en masse*. We added a new interface to allow the Gibraltar daemon to instruct the hypervisor about which pages are relevant and which are not. The daemon prepares a bitmap indicating pages in which data structures of importance reside. The hypervisor will only wake



3.3(a) Design space for code integrity checks



3.3(b) Design space for data integrity checks

Figure 3.3: The security versus energy tradeoff

up the daemon once *N* relevant pages in the bitmap have changed. This prevents the hypervisor from accounting for frequent stack or page cache modifications as relevant.

Figure 3.3 summarizes these concepts. The *y*-axis of each figure shows various subsets of the attack surface, while the *x*-axis considers the parameters used to decide the frequency of checks. The shaded portions of each figure show the portions of the design space that we explored in our experiments to quantify the security versus energy tradeoffs for mobile device rootkit detection.

3.6 Measuring the security we give away

Reducing the attack surface monitored or the frequency of checks introduces the possibility of evasion. A rootkit could evade detection by infecting the kernel between checks or by modifying unmonitored data structures. Therefore, the security provided to a system is intimately related to the frequency of checks and the attack surface monitored.

3.6.1 Impact of attack surface size

It is challenging to measure the impact of varying the attack surface on the security of the system. This is because (1) different entities in the attack surface impact system security to varying degrees; and (2) not all entities in the attack surface can be compromised with equal ease. With rootkits, attacks that modify kernel code, static data, and data structures that hold control data (*e.g.,* the system call table) are more abundant and easy to program than attacks that modify arbitrary kernel data structures.

For lack of a good metric, we default here to manual expert curation. Prior studies have shown that: (1) kernel code is far more important to rootkit detection than user-space code [38, 51,53]; (2) among rootkit-based attacks that modify kernel data, function pointers are the prime target [47] as opposed to other data structures. A 2007 study of 25 popular rootkits by Petroni *et al.* [47] showed that 24 of these rootkits modified function pointers to achieve their malicious goals. Rootkits that do not use function pointers as an attack vector typically either modify different data structures [17] or inject malicious code into the kernel [31].

3.6.2 Impact of check frequency

Constant vigilance is likely more effective than daily overnight checks at catching exploits before they have done much harm. To quantify how the frequency of checks impacts the security provided by a detection system, we introduce the concept of *window of vulnerability*.

The window of vulnerability for a given object is defined as the time elapsed between two consecutive checks on that object. For example, if we check the kernel system call table every two seconds, a rootkit has a maximum of two seconds to hijack the system call table, steal user information written to a file via `write()`, and optionally restore the table to its pristine state to avoid detection. Our window of vulnerability is therefore two seconds. For security systems that check multiple components, the window of vulnerability metrics of each component (each code page or each data structure in our case) can be statistically aggregated into a system-wide value (*e.g.,* as the window of vulnerability averaged over all components).

The window of vulnerability is the time period during which a system is vulnerable to attack. The greater the period of time between checks, the more time an attacker has to perform a sophisticated attack. For example, with a large window of vulnerability, a rootkit might have time to steal and transmit a user's personal information, *e.g.,* gathered during a secure browsing session using a key logger, to a malicious server. In general, a smaller window of vulnerability will expose fewer user interactions to the rootkit. For instance, with a smaller window of vulnerability, it may be possible to detect the presence of a rootkit and raise an alert before the user completes the secure browsing session, thereby protecting at least some of the user's personal information.

Periodic polling systems have a clearly defined set of windows of vulnerabilities that they expose for each object they check. For a polling period T, the average window of vulnerability will be *at least* T, plus the processing time involved within each round. However, event-based systems can provide a greater degree of assurance. If the hardware, can immediately alert the monitor of a potential threat even before it is allowed to happen, then the system can provide an effective window of vulnerability of zero. Doing so effectively requires the system to react to a potentially large volume of events. For this reason, it is common for an event-based system to perform event merging or coalescing, *e.g.,* interrupt batching for a processor, or signal handling

for UNIX processes. In this case the window of vulnerability widens again depending on the amount of merging performed.

3.7 Mitigating a new class of timing attacks

Resourceful and knowledgeable adversaries will immediately recognize a new opportunity. By learning the timing mode and parameters of the system, they can craft attacks that break in, exploit, and clean up within the period of time during which security checks are inactive. We mitigate this by randomizing the timing parameters. For example, if Gibraltar is to be configured to check data structures every T seconds, we instead trigger checks at intervals pulled from a uniform distribution in the interval (T-M,T+M], with $M \leq T$. To generate a proper uniform distribution, the hypervisor can tap from sources of entropy that are protected from the guest kernel, such as the count of hardware interrupts. Because the checking intervals are uniformly distributed in the interval (T-M,T+M], windows of vulnerability and checking overhead will converge to the same values as if a fixed period of T seconds had been chosen.

For event-based timing modes, we can apply the same randomization to the number of events that will trigger security checks. However, we have to further augment the approach with an explicit timeout (which itself could be randomized, if necessary). The reason for this is that the system may enter a steady state in which the selected threshold of events (e.g. page executions or modifications) is not reached, thus granting the attacker an unlimited window of vulnerability.

In this work we are focused on measuring and characterizing the tradeoffs between security checking and energy footprint. For those reasons, we use fixed intervals and event thresholds throughout the evaluation section. This removes an additional layer of experimental noise from our measurement goals.

A similar concern arises if we reduce the surface of coverage for our checks. Similarly, we might choose to catch the attacker off-guard by randomly triggering coverage of a wider attack surface.

3.8 Summary

In this chapter, we introduced two previously developed rootkit detectors, namely Gibraltar and Patagonix. Gibraltar is a rootkit detector which monitors kernel data structure invariants while Patagonix, on the other hand, is a rootkit detector which checks the integrity of code executing within a system. Using these detectors as an example, we described the security versus energy tradeoff on two axes, the attack surface and the frequency of checks. In the next chapter, we will measure the impact on energy and security while varying both the attack surface and the frequency of checks.

Chapter 4

Measuring the Security versus Energy Tradeoff

In this chapter, we measure the impact on security and energy usage while varying parameters within our framework, namely the attack surface and the frequency of checks. We also describe the experimental platform and the workloads employed during our study. Our goal is to illuminate various aspects of the security/energy tradeoff for host-based rootkit detection:

- *Impact of attack surface size.* If a malware detector provides greater security by monitoring a larger attack surface (*i.e.,* classes of attack), its detection algorithm will likely be more complex, CPU-intensive, or will take longer to execute. How does the size of the monitored attack surface impact battery life?

- *Impact of malware scanning schedule.* Malware detectors can be configured to be "always-on" tools that continuously monitor for malicious activity, or can periodically scan the mobile device. The former option provides increased security while the latter option improves battery life. How does the schedule of scanning impact energy consumption?

In section 4.4 we report our findings relating to the above questions. In section 4.9 we build upon our results to further address:

- *Adaptation.* Given the conventional wisdom that executing malware detectors reduces battery life, can we develop a strategy that maximizes security while minimizing battery consumption?

- *End user involvement.* Can we further expose such strategy and its inherent tradeoffs and options to end users? Can we do so in an intelligible manner similar to that used with traditional performance versus power-savings strategies?

4.1 Platform

We used a Viliv S5 mobile device [8] as our experimental platform. It is equipped with an Intel Atom Z520 1.33 GHz processor rated at 1.5 W, 4.8" touch screen, 32GB hard drive, 1GB of memory, WiFi, Bluetooth, a 3G modem, GPS, and a battery rated at 24,000 mWh. Since our rootkit detection tools are dependent on running in a virtualized environment, the ability to install a hypervisor on the device was a key requirement. The limited availability of mobile virtualization options dictated our platform choice. With its x86 Atom processor, the Viliv supports Xen paravirtualization [18], and is one of few such devices most resembling a smart phone that we could purchase in North America. Other virtualization platforms either require VT extensions [43], which are available only on a few higher-powered Atom models; are not available commercially and/or in open-source form (*e.g.,* VMware Mobile [9]); or cannot be installed on commodity smart phones available today (*e.g.,* the Xen port to the ARM platform [32] and the OKL4 Microvisor [4]). In spite of its slightly larger form-factor, the Viliv is functionally equivalent to a phone. Further, the Menlow chipset used by the Viliv is the precursor to the to-be-released Moorestown platform for Intel-based smart phones such as the LG GW990.

On the device, we used the Xen 3.4.2 hypervisor. Xen relies on a trusted domain (*i.e.,* dom0) to manage VM lifecycles and execute device drivers, which in our case was a Fedora 12 stack running a version of Linux 2.6.27.42 with appropriate Xen patches. We enhanced the hypervisor on the device with support for Patagonix and Gibraltar, and added the respective daemons to the dom0 stack. Our guest domain ran Linux 2.6.27.5 with Xen paravirtualization patches under a CentOS 5.5 distribution.

To measure power, we used a Tektronix TDS-3014 oscilloscope with a Hall effect current probe. When performing power measurements, we disconnected the battery from the Viliv S5 device and supplied power directly from a 5V source. We used this approach to ensure that the current we measure is directly powering the device. The current probe was attached to the charging cord and a laptop connected to the oscilloscope recorded the current readings over the time of an experiment.

4.2 Workloads

Experimental workloads that have traditionally been used to evaluate the performance of security tools, such as members of the SPEC family, often fail to capture the dynamics of the mobile experience. We therefore created our own workload for our evaluation. This workload aims to replicate standard mobile usage by loading a series of popular web pages and checking email.

Our workload is driven by a script that starts up the Firefox browser by pointing it to the desired site via a command line argument. It then monitors the CPU usage of the browser until it settles into reasonably low utilization; many popular sites employ Flash animations that never quite stop consuming resources. Once the site has quiesced, the script discards the Firefox instance, and moves on to the next site on the list. By pointing the browser to a Youtube clip, Flash playback will prevent quiescing of the browser throughout the duration of the clip, thus allowing full playback. The script similarly launches an email client and discards it after email checkout has finished and the process has quiesced.

Our workload is highly customizable and independent of a specific platform, needing just the ability to launch browser and email client instances from a script. We plan to augment our workload with fetching and uploading data to a social networking site and release it to the mobile computing community.

Throughout the experiments in this paper, we loaded `google.com`, `cnn.com`, `gmail.com` using an open account, `youtube.com` pointing to a 60-second video, and Thunderbird configured to check email from one IMAP account with several hundred messages in its inbox. We ran this workload on the Viliv using both 3G and WiFi connectivity, and for simplicity refer to the results respectively as "3G Browsing" and "WiFi Browsing."

For completeness, we also used lmbench [41], a CPU intensive workload designed to measure OS performance. We used the first six stages of lmbench because it thoroughly exercises multiple OS interfaces, thereby stressing our rootkit detectors.

Operation	Energy (mWh)
Send/Receive Phone Call	1.1 ± .03 / second
Send/Receive 160-char SMS	6.3 ± 1 / SMS
Send/Receive 5-char SMS	6.2 ± 1.2 / SMS

Table 4.1: Energy spent for common mobile phone operations

4.3 Rootkit detector configuration

To generate a database of invariants, Gibraltar must first execute a training phase. Since lmbench modifies many data structures in the operating system, we trained Gibraltar against multiple complete executions of lmbench. The result is a database of 131,201 invariants across 2209 data structure types with a size of 7MB.

Patagonix, requires a database of hashes for all binaries running on a system. To generate this database, we generated an ELF parsing tool to output a hash of each code page. We parsed all binaries located in the official CentOS repository, resulting in a database size of 36 MB. The database stores 10929 different binary files and 509709 hashes. We also store a database of 627 kernel code pages resulting in a size of less than 1 MB.

4.4 Experimental Results

In this section, we present experiments that illustrate the security versus energy tradeoff faced by kernel code-integrity and data-integrity monitors. In each experiment, we report the total energy dissipated by the Viliv as it executed one of three workloads (lmbench, 3G and WiFi Browsing) and the value of the corresponding security metric (attack surface or window of vulnerability). Unless otherwise noted, we report the average and standard deviations obtained from three experiment runs for each data point.

We start this section by quantifying the overhead of executing the hypervisor on the Viliv. We compare the execution of our workloads on a native bare-metal kernel, to the execution of our workloads inside a virtual machine, with no host-based rootkit detectors activated. As measured by total energy dissipation, the overhead is negligible in all cases. We observed the maximum overhead in the 227-second 3G browsing workload, with the energy footprint going from 333 to 335 mWh in virtualized mode (1.29%).

	No Security		Patagonix		Gibraltar		r
	Time (s)	Energy (mWh)	Time (s)	Energy (mWh)	Time (s)	Energy (mWh)	
lmbench	217.5 ± 0.2	261.39 ± 3.7	243.5 ± 2.1	303.31 ± 1.3	374.6 ± 13.4	473.56 ± 5.2	0.9995
3G Browsing	227.2 ± 9.9	333.84 ± 12.9	269.7 ± 5.8	375.36 ± 10.7	317.5 ± 14.1	479.95 ± 13.1	0.9779
WiFi Browsing	144.3 ± 5.3	141.08 ± 5.8	180.9 ± 8.9	187.84 ± 8.5	262.1 ± 7.5	230.38 ± 2.9	0.9707

Table 4.2: Runtime versus Energy correlation for security checks.

Table 4.1 presents the energy dissipated by two operations that are common to mobile phones: placing/receiving a 60-second phone call, and sending/receiving SMS messages. We observe that varying the SMS message size had relatively no effect on the total energy required to send the message. In the rest of this section, we will refer back to Table 4.1 to place the energy overheads in context by comparing them to the cost of common phone operations.

4.5 Impact of security on energy and runtime

The introduction of host-based rootkit detection has an immediate impact on the time to completion for a given workload, and a strongly correlated impact on the total energy consumed by the workload. Table 4.2 shows the measurements for our three workloads; we compare the original implementations of Patagonix and Gibraltar to runs without security checks. Pearson correlation coefficients between energy and time overheads exceed 0.97 for all workloads.

Primarily, host-based rootkit detection competes for CPU cycles with the workload, prolonging the time to completion. Patagonix's contention is a function of the amount of different code executed, and for our workloads the resulting overhead does not exceed 33%. Gibraltar is constantly asserting kernel data integrity, and thus constantly contending for CPU cycles. The overhead becomes dependent on the hardware in use. With a CPU-bound workload (lmbench) the overhead nears 100%. With network IO involved, there is no workload slowdown during periods in which the system stalls waiting for IO – Gibraltar occupies otherwise unused CPU cycles. With faster IO hardware (WiFi), there are fewer stall periods, and the relative overhead is higher (64% versus 43% for 3G). With slower and less-energy efficient hardware (3G), the longer fractions of IO stall occupied by Gibraltar yield a higher absolute overhead (146 mWh versus 89 mWh for WiFi).

The strong correlations between energy footprint and workload runtimes observed here also hold throughout the experiments in this chapter. We focus on energy measures and do not report runtimes due to space considerations.

4.6 Modifying the attack surface

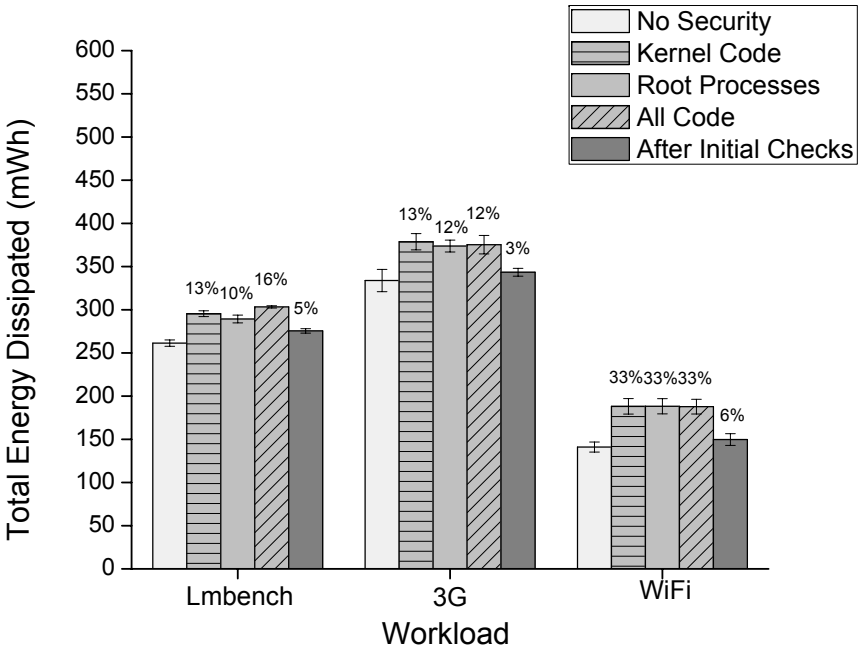
The energy dissipated by a security tool can be reduced by decreasing the fraction of the attack surface monitored. We quantify this observation considering the attack surface of code and data.

4.6.1 Impact on code integrity.

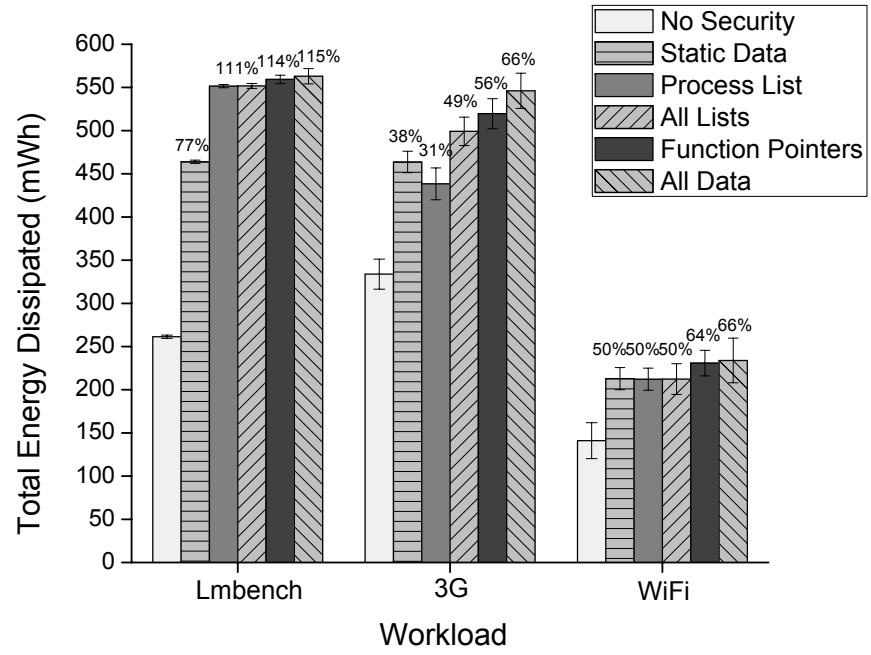
We configured Patagonix to monitor three subsets of the attack surface: (a) kernel code only; (b) kernel code and root processes; and (c) all code on the system, including kernel code, root and non-root processes. We set up Patagonix to check each code page as soon as it was scheduled for execution. On average, the Patagonix daemon verified 309 pages of kernel code as it executed the WiFi and 3G Browsing workloads; this number rose to 749 code pages when we included user-space code as well, 90 pages of which corresponded to root processes. During the execution of the lmbench workload, Patagonix verified 301 kernel code pages and 1602 user-space code pages, 11 of which belonged to root processes.

Figure 4.1(a) illustrates the energy dissipated by each of the three workloads. We present results for each subset of the attack surface considered. For each workload, the leftmost column is the baseline, which shows the energy dissipated by the workload executing in an environment with no security checks enabled (*i.e.,* in a hypervisor without Patagonix). Percentages reported above columns represent the extra energy dissipated (over the baseline value) when monitoring the corresponding subset of the attack surface. Comparing these results to Table 4.1, the extra energy dissipated by Patagonix when eagerly checking all code for the 144-second WiFi workload is the same as placing a 38 second phone call or sending 7 SMS messages.

Once Patagonix verifies the integrity of a code page, if the running process remains resident in memory and the code is not modified, Patagonix will never need to verify this page again, and it will therefore incur in no further overhead. This is particularly true for the kernel, which after boot remains resident and unchanged (save for module additions). The rightmost column in Figure 4.1(a) ("After Initial Checks") depicts the Patagonix overhead for the common case of recurring processes after bootstrap: the energy measurements were obtained by running the workloads a second time, after the initial execution. In this case, extra hypervisor

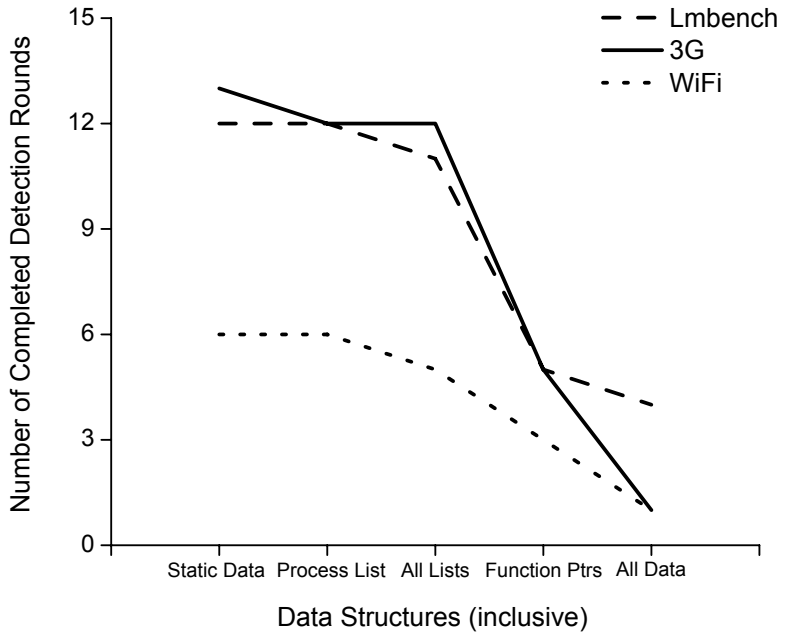


4.1(a) Code-integrity checks

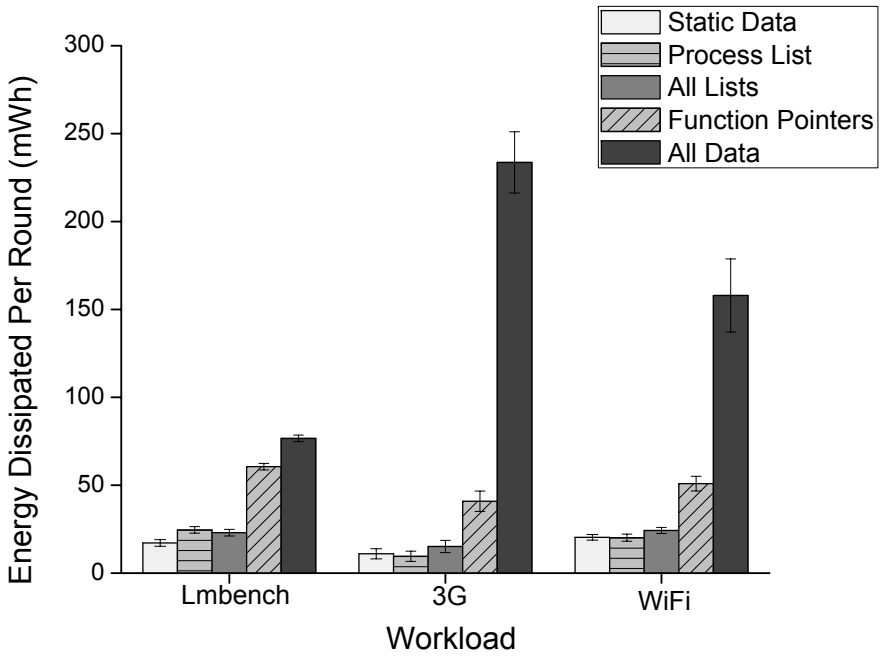


4.1(b) Data-integrity checks

Figure 4.1: Impact of varying the attack surface versus total energy dissipated



4.2(a) Number of rounds.



4.2(b) Energy dissipated per round.

Figure 4.2: Impact of varying the attack surface for kernel data integrity checks

work is necessary to enforce the **W⊗X** principle on the new page tables created. However, no additional daemon work is needed because the resident code pages have already been checked. The hypervisor overhead is small, and equivalent to a 10 second phone call or sending 2 SMS messages.

4.6.2 Impact on data integrity.

We configured Gibraltar to monitor five classes of kernel data, containing: (a) static kernel data, *i.e.,* data that is initialized during kernel boot-up and persists throughout the execution of the kernel; (b) data structures representing the process list; (c) all linked lists; (d) all kernel data structures that store function pointers; and (e) all data structures. Each class is inclusive, *i.e.,* data structures verified in each class also include the previous class as a subset. We set up Gibraltar to continuously monitor data integrity, and we refer to one complete traversal of the kernel's data segment as a *detection round*.

Figure 4.1(b) illustrates the total energy dissipated while Gibraltar monitors each of our three workloads. Our first observation is that the energy dissipated by Gibraltar is significantly higher than in the Patagonix case. As explained before, the Gibraltar daemon continuously contends for CPU cycles with the user workload. Our second observation is that the energy dissipated by Gibraltar varies with the attack surface being monitored. This is despite the fact that irrespective of the attack surface, Gibraltar executes continuously without any periods of dormancy. We hypothesize that cache pollution effects determine the overhead differences: with larger attack surfaces to cover, Gibraltar traverses a larger volume of data, thus effectively behaving as an adversarial workload in terms of memory locality. Decreased memory locality impacts processor cache performance efficiency, and thus energy efficiency. We plan to further investigate this effect, to potentially adjust Gibraltar's behavior to cache pollution rates.

Figure 4.2 presents the number of detection rounds Gibraltar completed throughout a workload, as well as the energy overhead per detection round. Both metrics are essentially locked in a zero-sum game: as the surface of attacks covered increases, more energy is spent in proportionally fewer rounds. When monitoring a smaller attack surface, data structures are checked more frequently (see Figure 4.2(a), each data class includes the previous class), presenting a

smaller window of vulnerability to attackers. The energy per round spent during the verification of kernel static data, linked lists and function pointers is significantly lower than that spent checking all data (see Figure 4.2(b)). This is fact is especially evident for the 3G and WiFi Browsing workloads, which dissipate approximately 3×-5× less energy than when Gibraltar monitors all data structures. This result is significant because a recent study of 25 rootkits [47] shows that 24 operate by violating the integrity of static data, linked lists or function pointers. As a consequence, Gibraltar can protect against most known attacks with modest energy dissipation per round: in the next section we show the limited amount of security we trade off by spacing these rounds and preventing continuous checking (and energy dissipation).

The number of data structures is one of two parameters that determine Gibraltar's coverage. The other is the number of invariants that are checked on these data structures. Decreasing the number of invariants from the original 131,201 to zero resulted in virtually no energy savings per round. We conclude that the dominant factor in the overhead per round for Gibraltar is the cost of reconstructing data structures.

4.7 Modifying the frequency of checks

Rootkit detection can be *event-based*, as in the original design of Patagonix, or *polling-based*. Patagonix can be adapted to batch events, while Gibraltar can be configured to poll kernel memory in different ways. In this section, we explore the effects of changing the frequency of checks, while measuring the security that we give away.

4.7.1 Impact on code integrity.

In the Patagonix experiments we observed that there were, on average, 50050 hypervisor notifications for lmbench, 13803 for 3G Browsing, and 15825 for WiFi Browsing. Each notification triggers a context switch to the trusted domain, where the page of code attempting to execute is checked. To decrease the number of context switches, Patagonix can be configured to add pages to a queue maintained in the hypervisor, notifying the daemon in the trusted domain only when the queue is full. Recall from subsection 3.3.1 that the hypervisor places information about a faulting executable page (page number, address space, and instruction address) on a

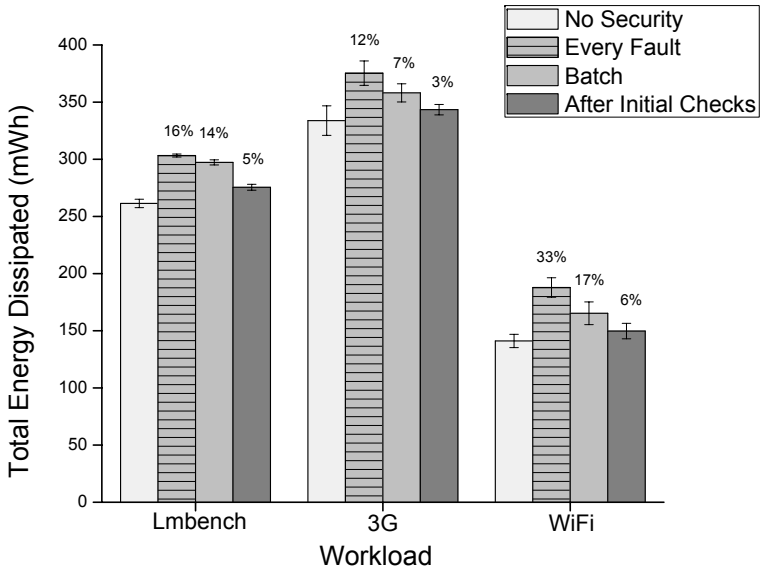


Figure 4.3: Impact of varying the frequency of code integrity checks

page shared with the trusted domain. The maximum number of entries a single 4 KB x86 memory page can hold is 341, thus dictating the size of our queue.

Figure 4.3 shows that batching code integrity checks results in a net decrease in energy dissipation for Patagonix. We attribute this primarily to the decrease in context switches. We observed 440 context switches while executing lmbench, 75 while executing the 3G Browsing workload, and 70 while executing the WiFi Browsing workload. This 99% decrease in the number of context switches yields the most impact for the WiFi workload: the Patagonix overhead decreases from the equivalent of placing a 38 second phone call to the equivalent of placing a 22 second phone call. Figure 4.4 shows that these results hold as we vary the coverage surface of our code integrity checks. Finally, as in the original case, subsequent executions of the workloads require no additional code verifications by the daemon, resulting in decreased energy expenditures up to a minimum of 3% for 3G browsing.

Batching code execution notifications fundamentally alters the security guarantees of Patagonix. By design, the original version of Patagonix offers a zero window of vulnerability: no code executes without prior inspection. Batching allows code to execute for a period of time

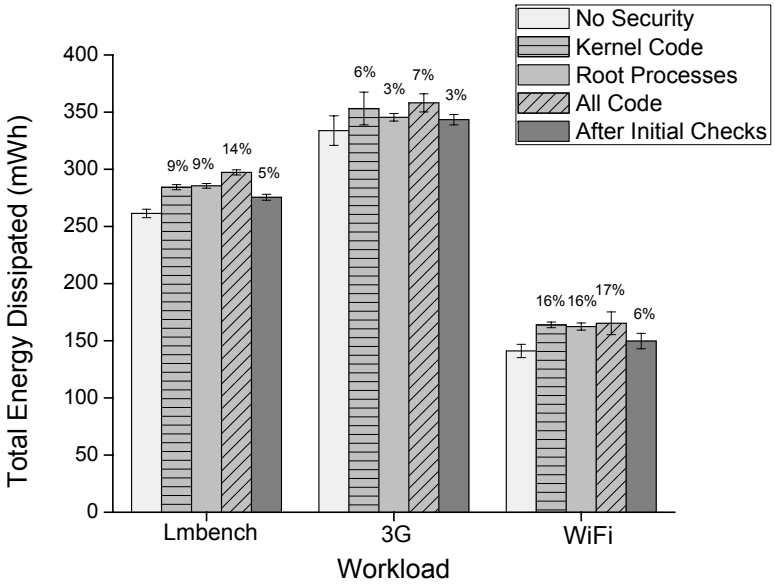


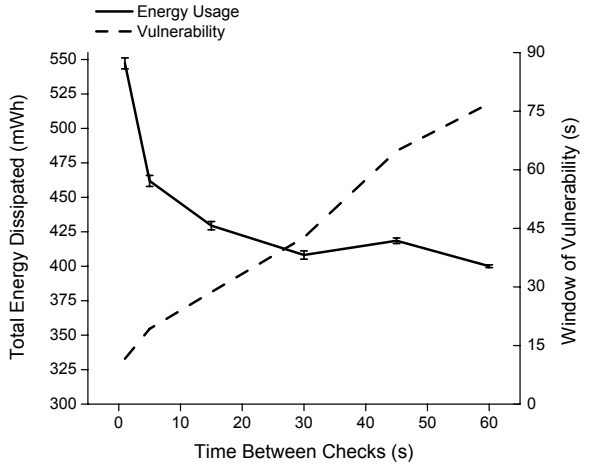
Figure 4.4: Impact of batching code integrity checks for various attack surfaces

Workload	Window of Vulnerability (s)
lmbench	2.5066 ± 3.39
3G Browsing	0.8766 ± 1.63
WiFi Browsing	0.7233 ± 1.79

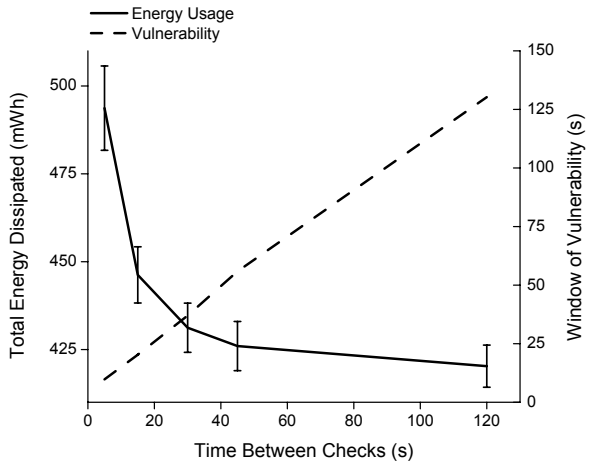
Table 4.3: Window of vulnerability for batched code integrity checks

without being modified, opening up a window of vulnerability. Table 4.3 shows that the windows of vulnerability are fairly small (under a second for browsing workloads), although quite variable because the rate at which new code pages execute is not at all uniform.

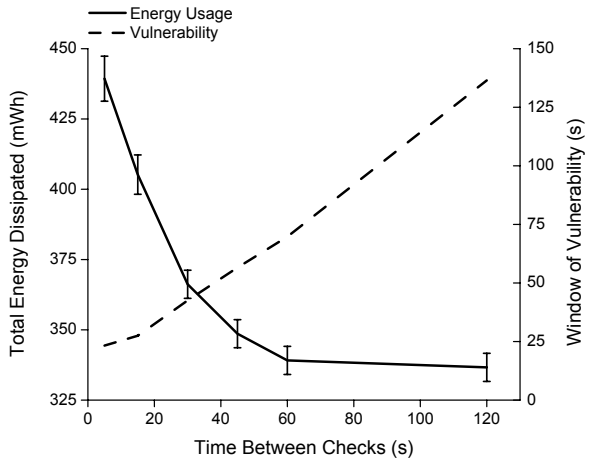
As discussed in Section 3.7, event-based queues need to be complemented with a timeout. Otherwise, the queue may never completely fill up, allowing in our case for rootkit code to remain undetected for arbitrarily long. We have not addressed this in this work as our focus was in studying mechanisms in isolation. From Table 4.3 we conclude that a timeout of five seconds would suffice.



(a) **Lmbench**

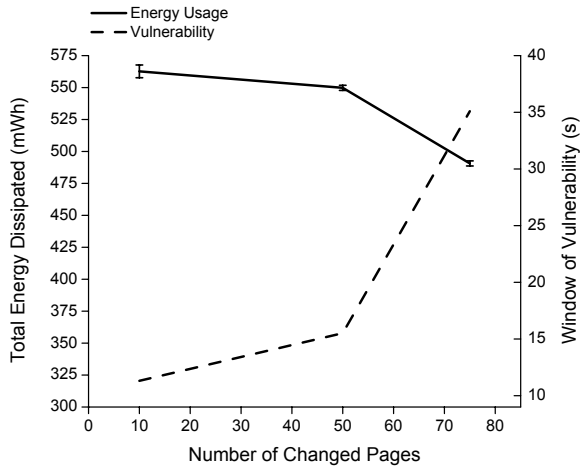


(b) **3G Browsing**

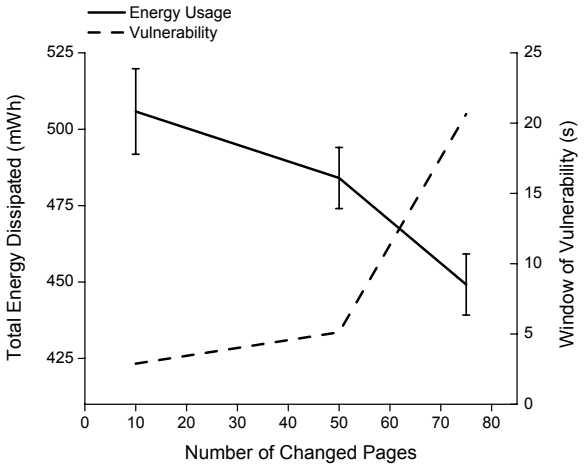


(c) **WiFi Browsing**

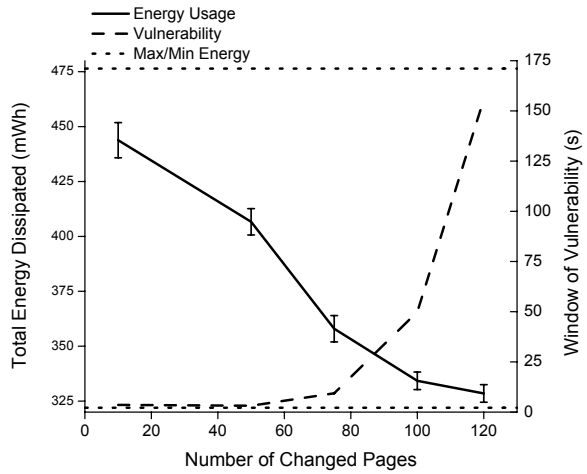
Figure 4.5: Security/Energy tradeoff when varying the period between data integrity checks



(a) **Lmbench**



(b) **3G Browsing**



(c) **WiFi Browsing**

Figure 4.6: Security/Energy tradeoff when varying page count for data integrity checks

4.7.2 Impact on data integrity.

Gibraltar can be configured to poll kernel data structures in one of two ways. The first configuration option uses a *polling period* T (in seconds) between detection rounds – the Gibraltar daemon starts a fresh traversal of kernel data structures T seconds after finishing the previous traversal. The second configuration option is event-based: the Gibraltar daemon is woken up after the guest kernel has modified N pages containing data structures of importance.

Figure 4.5 succinctly captures the security versus energy tradeoff. The solid lines represent the energy dissipated by the workloads executing in a guest domain monitored by Gibraltar, with different polling periods T varying between 0, 5, 15, 30, 45, 60, and 120 seconds. The broken lines represent the average window of vulnerability in the system. Increasing T results in less frequent rounds of verification – it increases battery life but decreases the overall security of the system, opening up wider windows of vulnerability. Recall that the average window of vulnerability is the mean of the times elapsed between consecutive integrity checks for each kernel data structure. The lower bound for the window of vulnerability is thus T, plus a small quantity derived from the time spent within each verification round.

Figure 4.6 presents the result of using the second configuration option to vary the frequency of checks. We configured Gibraltar to trigger integrity checks after N data pages have been modified, with N varying between 10, 50, 75, 100 and 120 pages. Both the lmbench and 3G Browsing workloads do not trigger a detection round for N=100 and N=120 pages. This is because these workloads repeatedly modify the same set of 75 to 99 kernel data pages. As the value of N increases, the amount of time between detection rounds also increases, and we observe the same phenomenon as in the polling case: energy overhead is traded off for an increase in the window of vulnerability in the kernel.

4.8 Cloud-offload Feasibility Study

Given the increasing popularity of cloud-offload, we conducted a feasibility study to investigate whether rootkit detection can be similarly offloaded. Instead of running the host-based rootkit detection logic locally, we perform a straight-forward partition. The same hypervisor logic executes in the device, selecting the same kernel code and data pages for checking. However,

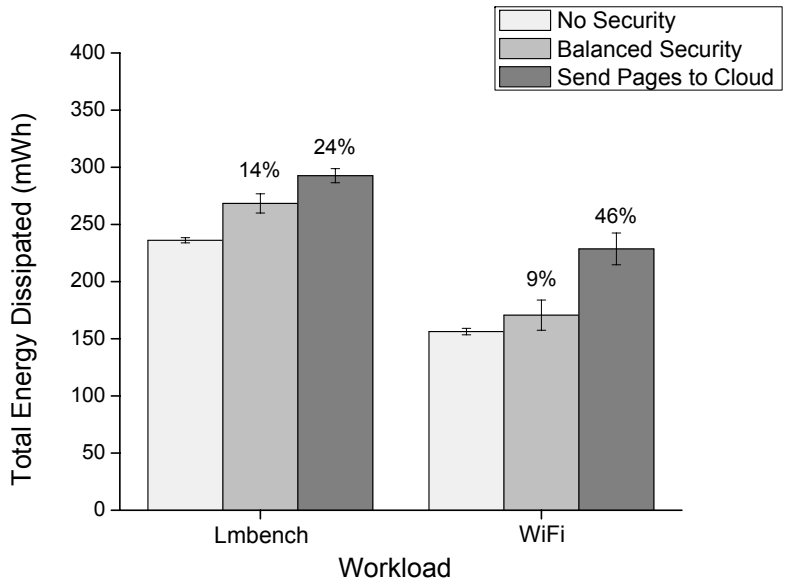


Figure 4.7: Cloud-based feasibility experiment

these pages are sent to a well-provisioned cloud server, which is idealized in two aspects. First, it replies immediately to the client: processing an arbitrary amount of rootkit detection logic consumes zero CPU cycles on the server. Second, we placed the server in the same LAN as the WiFi access point, resulting in small Internet RTT latencies.

Figure 4.7 compares the energy dissipation of the offloaded architecture with that of the balanced profile, and the case in which no security is enabled. We elided 3G from these experiments because: (a) previous work on cloud offload points to marginal energy gains at best using 3G [14, 25], and (b) 3G RTTs are substantially higher than WiFi RTTs in a LAN. For the browsing workload, cloud-offload presents a substantially higher energy overhead, due to the high frequency and volume at which kernel pages are sent. In spite of the idealized speed of the cloud server, network latency results in no gains in terms of windows of vulnerability. The results lead us to conclude that cloud-based rootkit detection is in principle more expensive than host-based detection, barring a fundamentally different approach.

4.9 Security/Energy Profiles

This section discusses how the results of the experiments from the previous section can be used to construct profiles that end-users can leverage to make educated decisions on how best to protect their mobile platforms. In that regard, security versus energy tradeoffs must be similar to performance/energy tradeoffs, which have existed since early laptop models and are therefore familiar to end-users. Such performance/energy tradeoffs are typically expressed succinctly, as a set of pre-defined power management profiles, in keeping with the conventional wisdom that a vast majority of end-users will steer away from both too much data and too many options.

Consider, for instance the power management profiles available on an iPhone running iOS 4.0. The only options exposed are: (1) screen brightness in standard mode; (2) the ability to automatically dim the screen brightness if inactive (but not parameters such as the dim gradient), and (3) the timeout period before the phone locks and the screen is turned off. Standard Windows 7 installations expose two power-management profiles, a *balanced* and *power saver* profile. Inquisitive users can find a third *high performance* profile. Further control is allowed by tailoring user-specific profiles. For security management to be useful, similar profiles must be made available to users. Power management profiles representing the extremities of the security versus energy tradeoff are, of course, easy to synthesize. In the rest of this section, we explain the reasoning behind a "best compromise" profile.

The results from the previous section show that energy cost association with checking code integrity is much lower than the cost of checking data integrity. The cost of Patagonix reduces further after code pages have been verified once and the system settles into a relatively stable working set of code pages. However, checking the integrity of kernel code alone is not sufficient to detect rootkits. Rootkits can perform their nefarious activities without installing new code to do so, *e.g.,* by using existing binary streams [54] or directly modifying kernel data structures by exploiting kernel buffer overflows. Static checking of code, as performed by Patagonix, cannot prevent potential hijacking of JiT code regions. Ultimately, control-flow is often governed by data structures such as function pointers, whose tampering could lead to subtle compromises.

Figure 4.2 shows that the power consumption of Gibraltar rises sharply when one increases its coverage to include all kernel data. However, checking the integrity of kernel data objects

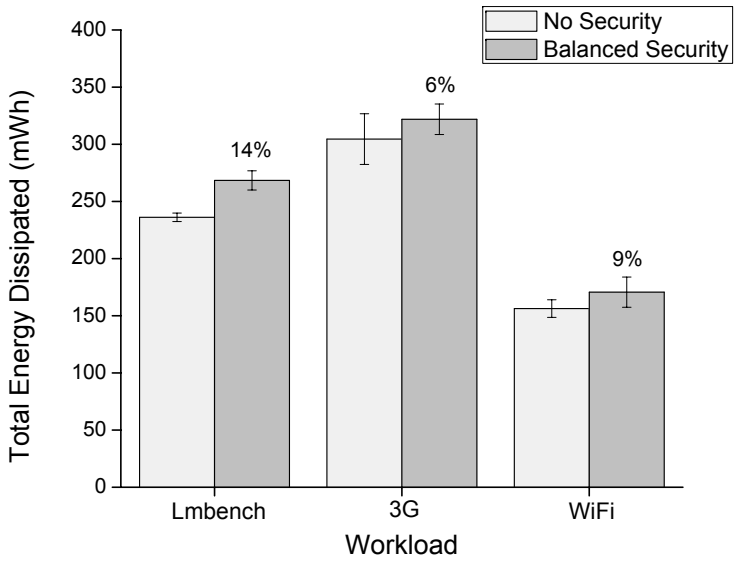


Figure 4.8: Energy dissipated for balanced security profile

that are typically attacked by rootkits (function pointers, process list, plus static objects) is three to five times cheaper energy-wise. The decision on which set of data structures to check is based on typical rootkit behavior [47] and is independent of the workload used in our experiments. Further, Figure 4.5 shows that a reasonable tradeoff between energy efficiency and window of vulnerability can be achieved by observing the intersection point between the corresponding curves. At the intersection point, there is a balance between energy consumption and the window of vulnerability of the system. Generally, this intersection point will be dependent on the workloads used to determine the tradeoff. From our browsing workloads, we determine the "sweet spot" as a polling mode with a period of $T = 30$ seconds. It is worth pointing out that for three fairly different workloads (from a system point-of-view), the intersection point lies in the neighborhood of $T = 30$ seconds. For the case of checking code integrity, Figure 4.3 shows that batching integrity checks reduces the energy overhead when using Patagonix.

Using this evidence, we construct a *balanced profile* for moderate energy consumption with a high degree of assurance against most rootkit attacks. This profile combines batched checks of kernel code pages, with polling-based integrity checks of static kernel data, linked lists and data structures containing function pointers, using the $T = 30$ second period identified as the sweet spot.

Figure 4.8 presents the energy dissipation of this balanced profile. Windows of vulnerability for kernel code vary between 2.5 and 0.7 seconds, while windows of vulnerability for kernel data structures monitored are on average 40.74 seconds. The energy overhead remains manageable at a maximum increase of 14%. Web browsing over 3G or WiFi incurs less overhead at 6% and 9%, respectively. The latter overhead is equivalent to a 15 second phone call or 2 SMS messages, for a workload originally taking 144 seconds. We note that the lower-energy mode into which Patagonix transitions after checking the kernel working set and resident processes is not included here.

The key shortcoming of predefined profiles (*e.g.,* the balanced profile) is that they rely on a set of assumptions about past rootkit behavior. If such profiles were to become standard in a software distribution, a vast user population will be bound to well-known profiles. Anecdotal observation indicates that most users never switch energy/performance profiles, pointing to a similar behavior for security versus energy profiles. Malware writers will thus be given the gift of a high-payoff and easy to study target. To complement the balanced profile we can use recent collaborative and behavior-based detection techniques described in Section 3.1 to design an *adaptive security versus energy profile*.

An adaptive rootkit detection tool could leverage the techniques mentioned in this dissertation to transition security states based on the perceived risk of user interactions. For example, a user browsing the internet might be considered a high risk scenario compared to a user placing a typical phone call. In this case, the tool can use this web browsing activity to automatically transition to a high security state to protect against malicious websites. During a normal phone call, the state may be transitioned to a low security mode. By automatically transitioning between power-saving and high-security modes, such an adaptive approach can protect against threats while also conserving battery power. It also provides the added benefit of not binding the device to fixed security versus energy profiles.

4.10 Summary

In this chapter, we measured the security versus energy tradeoff for two rootkit detectors by varying the attack surface and frequency of checks. In summary, we find that protecting against

code attacks is relatively cheap compared to data attacks, which require a significant amount of energy to protect against all attacks. From our results, we identified a sweet spot within the security versus energy tradeoff that minimizes both energy consumption and the window of vulnerability for attacks. In fact, this balanced profile protects against 97% of attacks while incurring a maximum of 14% energy overhead.

Chapter 5

Conclusion

The popularity of the mobile platform has already attracted attackers, who have increasingly begun to develop and deploy viruses, trojans, and worms that target these platforms. This dissertation demonstrated, through four proof-of-concept attacks, that kernel-level rootkits can exploit smart phone operating systems, often with serious social consequences. Rootkits evade detection by compromising the operating system, thereby allowing them to defeat user-space detection tools and operate stealthily for extended periods of time. As there was no previously available technique to detect rootkits on smart phones, we were motivated to explore techniques to effectively and efficiently detect rootkits on mobile devices. Because of the energy constrained nature of mobile devices, we show that detection techniques require a modification of design to work effectively.

In this regard, this dissertation explored the tradeoff between security monitoring and energy consumption on mobile devices. We studied security versus energy tradeoffs for host-based detectors, focusing on rootkits. We proposed a framework to investigate security versus energy tradeoffs along two axes, attack surface and malware scanning frequency, and to measure the security being traded off. We applied our framework to complementary hypervisor-based code- and data-based rootkit detectors on a phone-like device. Our results show that protecting against code-driven attacks is relatively cheap, while protecting against all data-driven attacks is prohibitively expensive. We identified a sweet spot in the security versus energy tradeoff, one that minimizes both energy consumption and the window of vulnerability opened as a result. This *balanced profile* is able to detect a vast majority of known attacks, which work against code and selected kernel data structures, while consuming a limited amount of battery power. We conclude by motivating the need for new mechanisms to enable cloud-offload of

rootkit detection, and by proposing the use of mobile-specific behavior-based anomaly detectors to transition between power-saving and high-assurance security modes.

References

[1] Alsa (advanced linux sound architecture). alsa-project.org.

[2] Gartner says worldwide mobile phone sales grew 17 percent in first quarter 2010. http://www.gartner.com/it/page.jsp?id=1372013.

[3] Google fixes android root-access flaw. www.zdnetasia.com/news/security/0,39044215,62048148,00.htm.

[4] OKl4 microvisor. http://www.ok-labs.com/products/okl4-microvisor.

[5] Openmoko Neo FreeRunner. wiki.openmoko.org/wiki/Neo_FreeRunner.

[6] Qtopia software stack (Qtextended.org). qtopia.net.

[7] Smartphones will soon turn computing on its head. news.cnet.com/8301-13579_3-9906697-37.html.

[8] Viliv S5 Real Pocket PC. http://www.myviliv.com/v4/product/s5/s5.asp.

[9] VMware Mobile Virtualization Platform. www.vmware.com/technology/mobile/.

[10] Rootkits, Part 1 of 3: A Growing Threat. http://download.nai.com/Products/mcafee-avert/whitepapers/akapoor_rootkits1.pdf, April 2006. MacAfee AVERT Labs Whitepaper.

[11] 2010 threat predictions. http://www.mcafee.com/us/resources/reports/rp-threat-predictions-2010.pdf, December 2009. MacAfee AVERT Labs Whitepaper.

[12] ABC News. Use Your Cell to Monitor Your Smart Home. http://abcnews.go.com/Technology/video/monitor-home-cell-phone-9887403.

[13] AT&T. AT&T, T-Mobile and Verizon Wireless Announce Joint Venture to Build National Mobile Commerce Network. http://www.att.com/gen/press-room?pid=18767&cdvn=news&newsarticleid=31369&mapcode=corporate|financial.

[14] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proc. of the 6th Conference on Computer Systems*, pages 301–314, April 2011.

[15] A. Baliga, V. Ganapathy, and L. Iftode. Automatic Inference and Enforcement of Kernel Data Structre Invariants. In *Proc. 24th Annual Computer Security Applications Conference*, pages 77–86, December 2008.

[16] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5):670–684, September 2011.

[17] A. Baliga, P. Kamat, and L. Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *Proc. 28th IEEE Symposium on Security and Privacy*, pages 246–251, May 2007.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. 19th Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[19] J. Bickford, R. OHare, A. Baliga, V. Ganapathy, and L. Iftode. Rootkits on Smart Phones: Attacks, Implications and Opportunities. In *Proc. 11th Workshop on Mobile Computing Systems and Applications*, pages 49–54, February 2010.

[20] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode. Security versus Energy Tradeoffs in Host-Based Mobile Malware Detection. In *Proc. 9th Conference on Mobile Systems, Applications, and Services*, pages 225–238, June 2011.

[21] C. Blesch. Rutgers Researchers Show New Security Threat Against âĂŸSmart PhoneâĂŹ Users. `http://news.rutgers.edu/medrel/news-releases/2010/02/rutgers-researchers-20100222`, February 2010.

[22] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral Detection of Malware on Mobile Handsets. In *Proc. 6th Conference on Mobile Systems, Applications, and Services*, pages 225–238, June 2007.

[23] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data Attacks Are Realistic Threats. In *Proc. 14th Conference on USENIX Security Symposium*, pages 12–12, August 2005.

[24] M. Christodorescu. *Behavior-based Malware Detection*. PhD thesis, University of Wisconsin-Madison, August 2007.

[25] E. Cuervo and A. Balasubramanian and D. Cho and A. Wolman and S. Saroiu and R. Chandra and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. 8th Conference on Mobile Systems, Applications and Service*, pages 49–62, June 2010.

[26] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proc. 16th Conference on Computer and Communications Security*, pages 235–245, November 2009.

[27] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. 69(1-3):35–45, December 2006.

[28] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Manuscript, Univ. of Maryland, `http://www.cs.umd.edu/~avik/projects/scandroidascaa`.

[29] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. 10th Network and Distributed Systems Security Symposium*, pages 191–206, February 2003.

[30] M. Grace, Z. Wang, D. Srinivasan, J. Li, X. Jiang, Z. Liang, and S. Liakh. Transparent Protection of Commodity OS Kernels Using Hardware Virtualization. In *Proc. 6th Conference on Security and Privacy in Communication Networks*, September 2010.

[31] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. In *Proc. 16th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, March 2011.

[32] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *Proc. 5th Consumer Communications and Networking Conference*, pages 257–261, January 2008.

[33] M. Hypponen. The state of cell phone malware in 2007. `www.usenix.org/events/sec07/tech/hypponen.pdf`.

[34] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proc. 13th Conference on USENIX Security Symposium*, pages 13–13, August 2004.

[35] H. Kim, J. Smith, and K. G. Shin. Detecting Energy-greedy Anomalies and Mobile Malware Variants. In *Proc. 6th Conference on Mobile Systems, Applications, and Services*, pages 239–252, June 2008.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. 22nd Symposium on Operating Systems Principles*, pages 207–220, October 2009.

[37] McAfee Labs. Mcafee threats report: Third quarter 2011, November 2011.

[38] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proc. 17th Conference on USENIX Security Symposium*, pages 243–258, August 2008.

[39] LWN.net. A New Adore Root Kit. `lwn.net/Articles/75990/`.

[40] Denis Maslennikov. Mobile malware and the internet: A myth or a reality?, June 2010.

[41] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. USENIX Annual Technical Conference*, pages 23–23, 1996.

[42] E. Monti. iPhone Rootkit? There's an App for That. `http://sandiego.toorcon.org/index.php?option=com_content&task=view&id=48&Itemid=9`, October 2010.

[43] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. 10(3), August 2006.

[44] J. Oberheide and F. Jahanian. When Mobile is Harder Than Fixed (and Vice Versa): Demystifying Security Challenges in Mobile Environments. In *Proc. 11th Workshop on Mobile Computing Systems and Applications*, pages 43–48, February 2010.

[45] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized In-Cloud Security Services for Mobile Devices. In *Proc. 1st Workshop on Virtualization in Mobile Computing*, pages 31–35, June 2008.

[46] N. Percoco and C. Papathanasiou. This Is Not the Droid You're Looking For... http://www.defcon.org/images/defcon-18/dc-18-presentations/Trustwave-Spiderlabs/DEFCON-18-Trustwave-Spiderlabs-Android-Rootkit-WP.pdf, July 2010.

[47] N. Petroni and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proc. 14th Conference on Computer and Communications Security*, pages 103–115, October 2007.

[48] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An Architecture for Specification-based Detection of Semantic Integrity Violations of Kernel Dynamic Data. In *Proc. 15th Conference on USENIX Security Symposium*, August 2006.

[49] G. Portokalidisi, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile Protection For Smartphones. In *Proc. 26th Annual Computer Security Applications Conference*, pages 347–356, December 2010.

[50] R. Racic and D. Ma and H. Chen. Exploiting MMS Vulnerabilities to Stealthily Exhaust Mobile PhoneâĂŹs Battery. In *Proc. 2nd Conference on Security and Privacy in Communication Network*, pages 1–10, August 2006.

[51] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proc. 11th Symposium on Recent Advances in Intrusion Detection*, pages 1–20, September 2008.

[52] Lookout Mobile Security. Update: Security Alert: DroidDream Malware Found in Official Android Market. http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/, March 2011.

[53] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proc. 21st Symposium on Operating Systems Principles*, pages 335–350, November 2007.

[54] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. 14th Conference on Computer and Communications Security*, pages 552–561, October 2007.

[55] P.C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computting*, 2(2):82–92, April 2005.

[56] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proc. 16th Conference on Computer and Communications Security*, pages 545–554, November 2009.

[57] J. Xuxian. GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3 (Gingerbread). http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/, August 2011.

[58] J. Xuxian. Security Alert: DroidCoupon Masquerades as Coupon App. http://research.netqin.com/?p=112/, September 2011.

[59] J. Xuxian. Security Alert: New Root-Capable DroidDeluxe Malware Found in Alternative AndroidMarkets. http://www.csc.ncsu.edu/faculty/jiang/DroidDeluxe/, September 2011.

[60] J. Xuxian. Security Alert: New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html, May 2011.

[61] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. In *Proc. 10th Workshop on ACM SIGOPS European Workshop: Beyond the PC*, pages 239–242, July 2002.