9-10-2010

Code White: A Signed Code Protection Mechanism for Smartphones

Joseph M. Hinson IV

CODE WHITE: A SIGNED CODE PROTECTION MECHANISM FOR SMARTPHONES

THESIS

Joseph M. Hinson, IV, Captain, USAF

AFIT/GCO/ENG/10-10

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCO/ENG/10-10

CODE WHITE: A SIGNED CODE PROTECTION MECHANISM FOR SMARTPHONES

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Joseph M. Hinson, IV, B.S. Electrical Engineering

Captain, USAF

September 2010

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCO/ENG/10-10

CODE WHITE: A SIGNED CODE PROTECTION MECHANISM FOR SMARTPHONES

Joseph M. Hinson, IV, B.S. Electrical Engineering

Captain, USAF

Approved:

Rusty Bald
Dr. Rusty Baldwin (Chairman)

9/9/10
date

Barry Mullins
Dr. Barry Mullins (Member)

9 Sep 10
date

William Kimball
Mr. William Kimball (Member)

9/9/10
date

Abstract

This research develops Code White, a hardware-implemented trusted execution mechanism for the Symbian mobile operating system. Code White combines a signed whitelist approach with the execution prevention technology offered by the ARM architecture. Testing shows that it prevents all untrusted user applications from executing while allowing all trusted applications to load and run. Performance testing in contrast with an unmodified Symbian system shows that the difference in load time increases linearly as the application file size increases. The predicted load time for an application with a one megabyte code section remains well below one second, ensuring uninterrupted experience for the user.

Smartphones have proven to be invaluable to military, civic, and business users due in a large part to their ability to execute code just like any desktop computer can. While many useful applications have been developed for these users, numerous malicious programs have also surfaced. And while smartphones have desktop-like capabilities to execute software, they do not have the same resources to scan for malware. More efficient means, like Code White, which minimize resource usage are needed to protect the data and capabilities found in smartphones.

Acknowledgments

I would like to thank my committee for their invaluable contributions to this work. I could not have done it without them. My advisor, Dr. Rusty Baldwin, provided excellent guidance, expertise, and instruction throughout the past year. The high standards he set helped me stay on track, and his ever-present sense of humor was much appreciated. Dr. Barry Mullins has been an inspiring instructor and mentor in both my undergraduate and graduate studies. Mr. Bill Kimball provided countless hours of patient instruction and troubleshooting help, as well as many alternative viewpoints to help me think “outside the box”. I’m also deeply grateful to my family, especially my wonderful, patient wife for her support. She probably worked harder than I did every day just so I could focus on this research. Finally, to my beautiful twin daughters: your arrival in the middle of this research didn’t help me complete the work, but it did put it in perspective. Thank you.

Table of Contents

List of Figures

Figure Page

List of Tables

Table Page

CODE WHITE: A SIGNED CODE PROTECTION MECHANISM FOR SMARTPHONES

I. Introduction

1.1 Research Domain

Mobile phones are ubiquitous devices. Once only a means to extend the reach of the wired phone system, the cell phone has become much more. It carries many additional capabilities, like sending written messages, storing personal information, providing location and direction information, accessing web content, performing business transactions, unlocking car doors, and changing channels on the television. The list of features and capabilities continues to grow. While there are varying views of what constitutes a smartphone, for this research a smartphone is a mobile phone that can execute third party code.

1.2 Problem Statement

To the general public, mobile phones are much more than a convenient means to communicate; they have become almost another appendage. They often hold much of the critical information a person has. This encapsulation of information and identity is also an essential resource in business, civic, and military operations. Since the release of the iPhone in 2007, mobile users are now looking for one more capability in their phones: the ability to run software programs like they can on their personal computers. Given that, attackers have every incentive to develop malware to infect and exploit these small, ever-present, always-on, ready-to-execute-your-code smartphones to gain access to information and identities and thus to the capabilities of their respective users.

1.3 Research Goals

The goal of this research is to increase smartphone security by identifying and adapting a novel protection mechanism developed for the general purpose computing domain for use in a smartphone. The method is tested to verify it protects the cell phone from malware and determine whether the modified phone's performance is acceptable compared to the performance of an unmodified phone.

Chapter 2 discusses current literature on smartphones, their threats and vulnerabilities, and protection methods for them. Chapter 3 introduces the novel protection mechanism for smartphones called Code White. Chapter 4 evaluates Code White's performance. Finally, Chapter 5 discusses these results as well as future areas of research.

II. **Smartphones and Mobile Malware**

This chapter provides an overview of smartphone capabilities, vulnerabilities, threats, and protection mechanisms.

2.1 Incentives to Attack

The capabilities of modern cell phones come at the price of a host of security vulnerabilities. Despite that, a majority of survey respondents acknowledged they would prefer to lose their wallet than their cell phone [Glo09]. Mobile phones combine a number of features that make them invaluable as well as extremely vulnerable. They provide the capability to connect to the rest of the world, including friends, family, and emergency services. They hold valuable information: personal information like contact information, address book, current location as well as financial information like bank account and credit card numbers to name but two categories. They are small which makes them easy to steal or lose. They communicate via many different standards like cellular telephony, Bluetooth and WiFi. They can be updated or even modified by their users. Finally, so called smartphones can run third party code that extends their capabilities.

Cell phone attacks have been possible since the introduction of cell phones. They began with simple denial of service attacks, including RF jamming. With the advent of messaging capabilities, additional vulnerabilities arose. Each phone sent to market has its own code for handling incoming messages. Attackers use specially-crafted messages to exploit vulnerabilities in this code to disable the phone or accomplish some other goal.

But the greatest leap in malicious potential is the ability to run custom applications. Where previous attacks were constrained to conform to a particular vector (like SMS message size limitations [Rob03]), attacks now can be written into programs and downloaded for direct execution. A hacker must still defeat any code protections on the device, but the scope of possibilities available is much greater. Hackers need only access a mobile OS' API and the functionality it provides.

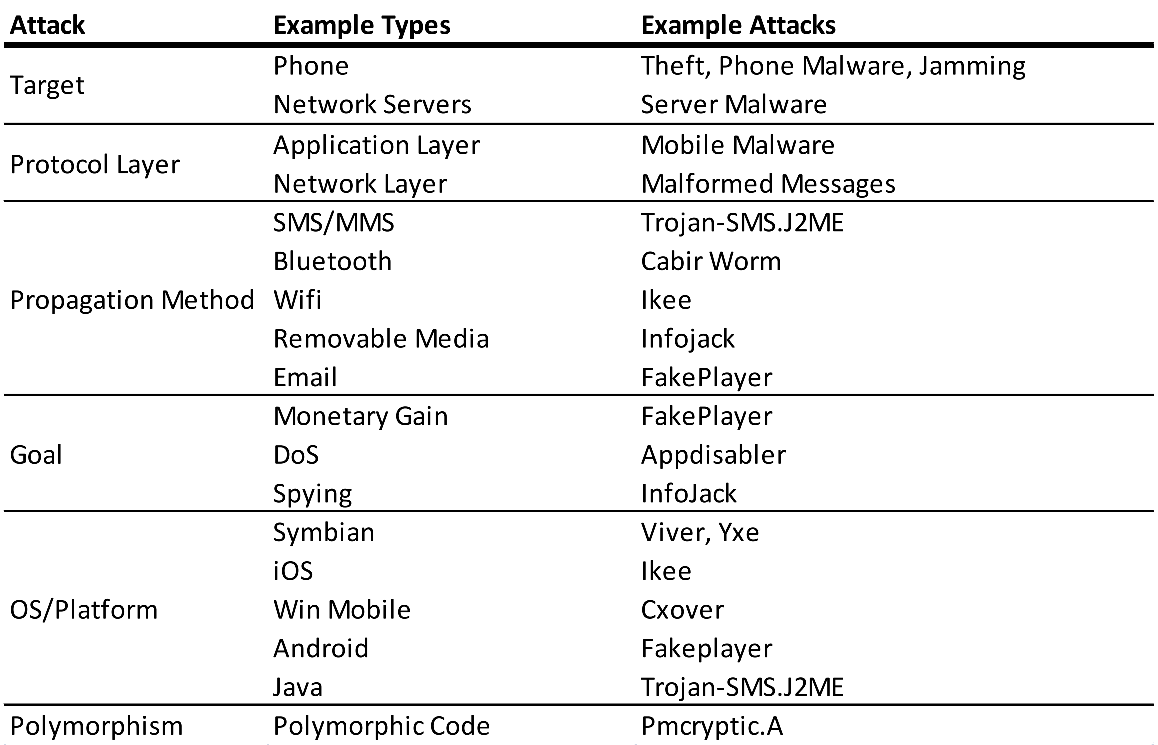
Another incentive for malware development on mobile devices is through premium services via the Short Messaging Service (SMS). By sending a message to a specific number (usually shorter than a standard number), the account of the sending phone is charged a fee for each message sent. A criminal can lease a short number and release malware that causes infected phones to send periodic messages to the number in question, unbeknownst to their owners. Since many premium fee services are international and the perpetrator can collect payment from the number provider before their victims receive billing statements, this crime is very difficult to prosecute. [Gos09] There are a number of ways to prevent messages from being sent; including having the user acknowledge each message, but such measures can be subverted. For example, a message promising a pornographic image if the user selects “yes” to show that they are 18 or older.

2.2 Threats

The unique nature and vulnerabilities of smartphones make them a prime target for malware. This section discusses the goals, methods, and features of smartphone-targeting malware.

The first known mobile malware was the Cabir worm, developed as a proof of concept in 2004 by the hacker group 29A [Gos09]. It contained no malicious payload, only the ability to replicate itself and propagate via Bluetooth on Symbian phones. Since 2004, mobile malware development has branched out to different platforms with different goals. It is beneficial to create a taxonomy of attacks and thereby classify the many variations of malicious code found in the wild. The following sections discuss various criteria for categorizing different types of malware. The resultant taxonomy is shown in Table 1.

Table 1 - A Taxonomy of Mobile Malware



Attack	Example Types	Example Attacks
Target	Phone	Theft, Phone Malware, Jamming
	Network Servers	Server Malware
Protocol Layer	Application Layer	Mobile Malware
	Network Layer	Malformed Messages
Propagation Method	SMS/MMS	Trojan-SMS.J2ME
	Bluetooth	Cabir Worm
	Wifi	Ikee
	Removable Media	Infojack
	Email	FakePlayer
Goal	Monetary Gain	FakePlayer
	DoS	Appdisabler
	Spying	InfoJack
OS/Platform	Symbian	Viver, Yxe
	iOS	Ikee
	Win Mobile	Cxover
	Android	Fakeplayer
	Java	Trojan-SMS.J2ME
Polymorphism	Polymorphic Code	Pmcryptic.A

2.2.1 Attacks Defined by Target or Protocol

The first category of malware is based on the attack target within the mobile network (i.e., call center, mobile phone, base station). Another distinguishes between attacks based on the protocol layer targeted (i.e., jamming on the physical layer, spoofing). This includes attacks against the mobile device itself as well as the applications running on it. [Guo07]

2.2.2 Attacks Defined by Propagation method

Another category focuses on propagation methods. Modern phones communicate via a variety of media and standards beginning with the mobile networks themselves. The category for these networks is further extensible to include communication type: Short Message Service (SMS), Multimedia Messaging Services (MMS), and data services (internet connectivity) to name a few. Many phones also have the capability to access WiFi or WiMax networks for additional data services. For shorter transmissions, (normally to peer phones or peripherals) many include native Bluetooth capabilities. Most also connect to and synchronize with personal computers, typically via USB or infrared interfaces. Finally, removable storage and peripherals provides an avenue through which malware may enter. From the messaging system to memory cards to Common Access Cards, there are many ways to reach victim phones. [Che07]

Bluetooth is a ready attack vector as many users leave it enabled on their phone in a discoverable mode which is precisely how the first virus written for smartphones was propagated. In 2005, the Cabir worm made news when an outbreak was discovered at the World Athletic Championships in Helsinki, Finland. No damage was done to the victim phones, but the dozens of infections at a single event opened the eyes of users and developers around the globe. [Ley05]

With an increase in text and multimedia messaging, SMS and MMS grow in popularity as methods for malware transmission. Figure 1 shows the growth of SMS messaging in the United States. In addition to Bluetooth, the Mabir variant of the Cabir worm uses MMS to propagate. Upon receipt of any SMS or MMS message, an infected phone replies via MMS with a message that contains only the installation file for the worm. If the user thinks it is a reply to their recent message and opens the file, the worm is installed in an autostart mode [Wor09].

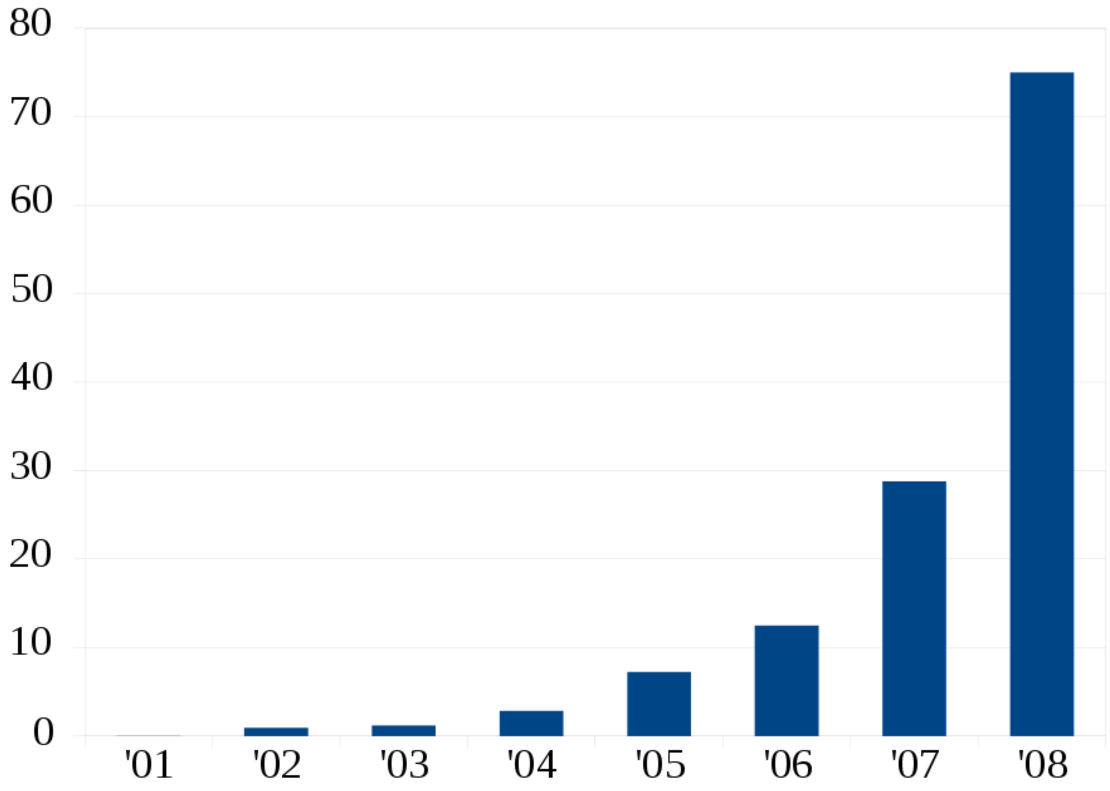


Figure 1 - SMS Text Messages Sent Monthly in USA in Billions (Wikipedia)

The InfoJack trojan is an example of malware that propagates via memory cards. Its creator packed it with legitimate executable files to encourage distribution. Once executed on a WinCE phone, the trojan installs itself and makes modifications to the operating system, including one that allows further installations without prompting the user. It also copies itself to any attached media cards and sets itself as an autorun file. If another user subsequently puts the media card into another WinCE phone, the process repeats. InfoJack attempts to contact its host website, upload information from the phone and download any updates. The website has since been taken down [Sha08][Gos09].

2.2.3 Attacks Defined by Goal

Another way to differentiate malware is by their attack goal. Malware has many objectives, but a leading one is theft. This can be direct monetary theft as discussed above for premium numbers or information theft which can indirectly be used for monetary gain. A virus can steal and destroy data from phones, run up bills by making calls to premium-rate numbers, record conversations where personal data and credit card numbers are exchanged, and even get a phone camera to spy on its owner and transmit photos [Bie05]. Other attack types include Denial of Service (direct or indirect via battery drain), hijacking, and others. These attacks can all be aligned with the three essential elements of computer security: confidentiality, integrity, and availability (CIA). Information theft attacks breach confidentiality, hijacking breaks integrity, and DoS attacks impede availability [Dag04].

In addition to analyzing the goal of the malware, the targeting methods can also be a basis for categorization. Some malware is packaged with a specific pre-determined target. Others choose targets from infected hosts by sending copies of themselves to contacts listed in a victim’s phone. Others choose targets randomly, propagating to devices at randomly generated numbers or addresses [Che07].

The Multidropper Trojan infects Symbian phones and installs a number of other programs on each device. One is the Kiazha Trojan which creates an account for the victim user on a remote server. It forwards the victim's messages and personal data to the server and deletes them from the phone. It then displays a message to the user demanding payment for the return of the stolen data [Tro08]. This is an example of malware created with the end goal of monetary gain.

Malware with a different goal is the Kblock Trojan. After being installed it locks the keypad. To clear the malware requires the phone to be reset to its factory default state which deletes all personal information [Sym09].

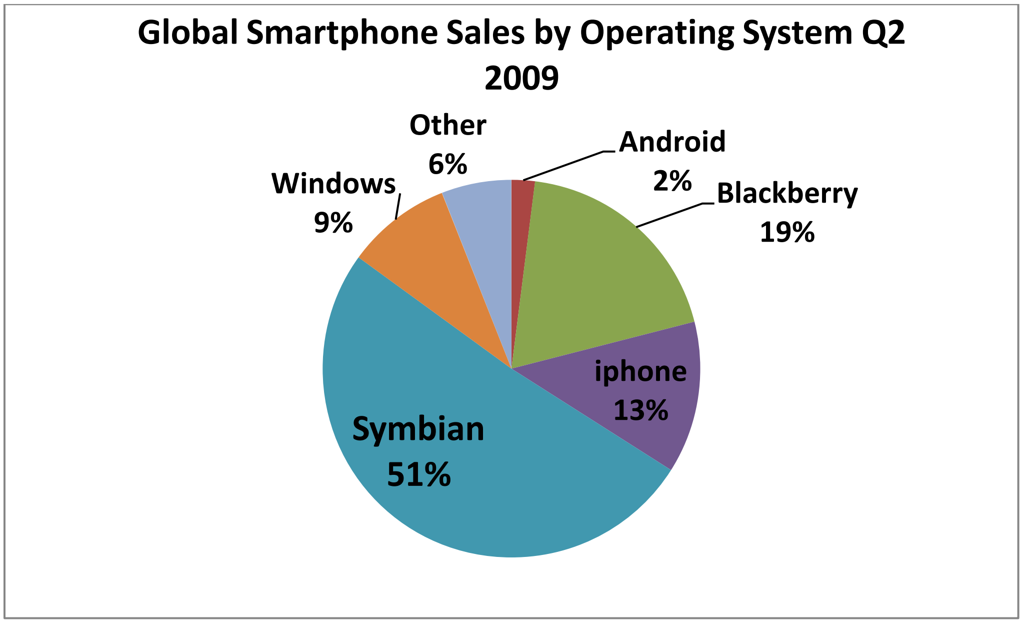


Figure 2 - Global Smartphone Sales by OS Q2 2009 (Source: Gartner)

2.2.4 Other Characteristics

Other useful distinctions between malware can be made as well. Examining the market share of the various mobile operating systems provides insight into the trends in Smartphone threats. Figure 2 shows Q2 2009 sales data for Smartphones around the world by operating system. Notably, the Symbian OS has sold more phones than all the others combined.

Figure 3 shows the distribution of mobile malware families by the operating systems they target. Symbian OS is the clear leader due to its large market presence which supports the correlation between malware targeting and targeted platform distribution. The Symbian OS has more threats than any other because of its wider distribution. Likewise, Java viruses are more prevalent since the Java Micro Edition (J2ME) provides a platform for a virus to run on many systems without being individually rewritten.

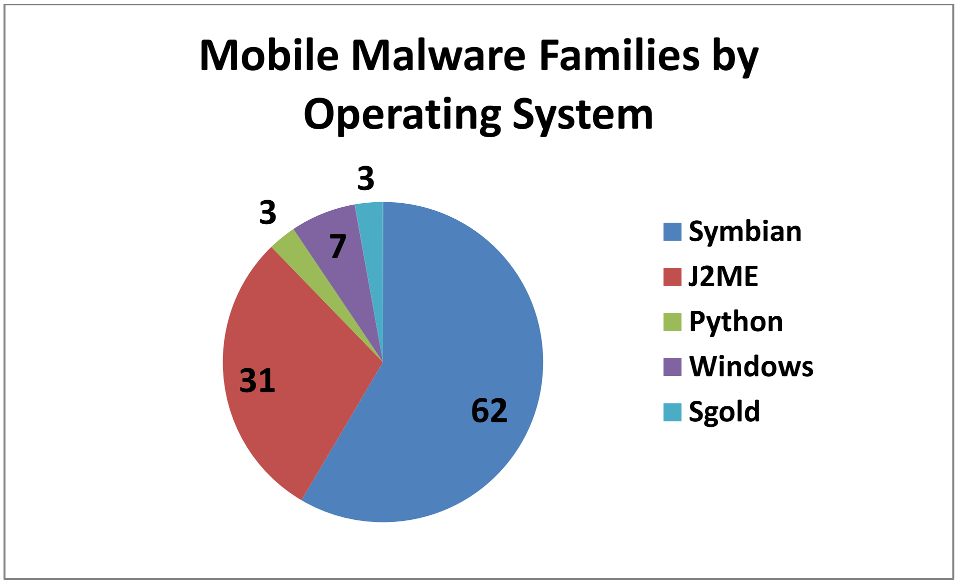


Figure 3 - Malware by Operating System [Gos09]

To date, relatively little malware has targeted iPhone OS, Google's Android, or Blackberry OS. Possible reasons include market share, length of time on the market (Android 1.0 was released in Oct 2008), capability of writing malware for platforms that run on top of those platforms, and the security features implemented to prevent malware.

The latter two are discussed in more detail later in this chapter. No doubt, more malware for these phones will emerge in time.

Another distinguishing feature of mobile malware is polymorphism (i.e., code that modifies itself each time it makes a copy to avoid detection). One polymorphic mobile worm is Pmcryptic.A which infects WinCE phones. It runs a variety of payloads from meaningless popups to dialing premium numbers to file deletion. In addition, it replicates itself multiple times on the device, including on each removable storage device it finds. Each replication is appended with up to 255 bytes of random data. This weak form of polymorphism makes it more difficult to detect by scanning [Fer08]. While only a few mobile viruses employ polymorphism, more are expected as mobile malware matures.

2.3 Current Research on Mobile Phone Protections

Given the potential damages discussed above and the ramifications to users, many options to protect Smartphones and the data they contain are being explored. This section discusses a number of protection strategies and detection mechanisms aimed at securing Smartphones from misuse.

2.3.1 Antivirus and Mobile Phones

Before discussing mobile security, it is helpful to review popular protection measures from the realm of general purpose computing and evaluate those measures from the standpoint of mobile computing. A protection many consider most analogous to computer security is the use of antivirus (AV) software.

AV software is a brute force approach to security. The software scans all files on an information system for the presence of malware, and repeats the process often. The frequency of scanning can be adjusted, but typically are daily or weekly.

Consider the differences between phones and desktop computers. Rather than a multi-gigahertz processor, a top of the line CPU in a phone runs a few hundred megahertz. In place of virtually limitless power from a wall socket, batteries supply a phone with the majority of its power. There are similar limitations in other resources which make AV unattractive for mobile use. Though phones generally have significantly fewer files than a typical PC and have the recently added capability to multitask, they cannot provide the expected performance in their primary services (calling, messaging, etc) and run an AV profile like that used on a desktop. Additionally, an AV program on a phone is not as effective as one running on a desktop since its effectiveness is based on the currency of its signature. Depending on factors such as range, reception, and roaming, the delivery of these files may not be timely for a given phone.

Though antivirus is not a good solution for mobile devices, there is still a need for host-based protection [Mie06]. Cellular networks provide some protections external to the mobile device, but they are no replacement for security measures on the device itself. The following protections are currently being used or are being investigated as a way to protect mobile phones and the data they process.

2.3.2 Application Protection

Third party applications pose one of the greatest risks to the smartphone. Some are poorly written and introduce security holes and instability to the system. Others are malicious. Many methods protect the smartphone from errant application activity. Table 2 shows two of the most popular application protection measures as implemented by five of the top mobile operating systems: sandboxing and signing.

Table 2 - Security Features of Mobile Operating Systems

OS	**Sandboxing**	**Signing**
Android	All applications run in JVM	All must be signed by author [Sig10]
Blackberry	All applications run in JVM except included core apps	All (signing authority based on capabilities) [Sch09]
iPhone OS	Native installation, sandboxed via file permissions, memory space, etc. [The10]	All
Symbian	Not required. Apps can be native or run on Java, etc	All (signing authority based on capabilities)
Windows	Not required. Apps can be native or run on Java, etc	Yes, but user can override

2.3.2.1 Sandboxing

Sandboxing, or virtualization, implements a computing environment within another computing environment. The virtual machine provides resources and acts as if it were running directly on hardware though it is fully contained by the host system. In the case of a guest system crash, the host maintains its stability, and merely terminates the guest environment process. Additionally, the host prevents the guest from accessing critical files and data on the host system. A phone may also have multiple virtualization layers implemented. This allows a user to use their phone for multiple roles (like personal and business use) and may allow some applications to only access data or resources for a single role [Kro09].

Five of the top mobile operating systems sandboxing implementations are shown in Table 2. Android and Blackberry require all third party applications to execute within a Java virtual machine. The iPhone OS installs all applications natively, but sandboxes them via virtual memory techniques and file protection measures. Finally, Symbian and Windows do not require sandboxing, though it is supported through Java and others.

2.3.2.2 Application Signing

Application signing affixes a unique signature to executable code for identification purposes. The signature can serve a number of purposes including validating the code integrity and providing an audit trail back to the original author. There are various signature types, but most current implementations use cryptographic public keys issued by a certification authority (CA). The CA issues the requesting party a unique public and private key pair. One key will identify the other, but cannot easily be deduced by having the other. The keys are used to sign the files in the following fashion: the signing authority computes a one way hash of the file. The signer's private key is used to encrypt the hash and any other desired information but generally not the code itself for efficiency reasons. The output is the actual signature. The signature is included in the executable file itself.

All major mobile OSes require some form of application signing. Usually they require that developers obtain and use a unique signature for identification purposes. However, some (Blackberry for instance) require that if an application makes use of certain critical APIs, the app must be vetted and signed by the OS' own authority. As for application installation, most OSes reject unsigned or malsigned applications. Windows however, allows installation if the user overrides the onscreen warning.

Application signing is not completely effective. Malware authors have successfully signed their wares before. The Yxe, Album, CommDN, and NMPlugin worms are all examples of malware that has been signed by Symbian. Malware authors take advantage of inexpensive signing fees, low audit rates, and an emphasis on stability over security in the application process to have their wares signed. [Jar07][Apv10].

Akin to application signing is application certification, which is is based on the capabilities of a prospective application as opposed to its signature. Many mobile OS' require applications to identify what system capabilities they require access to at signing time (like Symbian Signed). Kirin, a proof-of-concept certification method implements an application installer which checks all application capabilities at install time and looks for potentially dangerous capability combinations (for instance, an application that can start on boot, read geographic location and access the Internet could be a malicious tracker). Resource consumption is small, since this service is only invoked at install time [Enc09].

2.3.3 Anomaly Detection

One method used to decrease risk is to use observable behavior from the phone to determine if its activity is questionable. Within this category of protection, there are two variations: anomaly detection and misuse detection. Anomaly detection defines a "normal" profile for a system and looks for discrepancies; misuse detection uses rules to designate states as "good" or "bad".

A common consideration for anomaly and misuse detection is determining the best place to perform the detection. If detection and analysis are performed on the mobile device, the system has the same disadvantage as a standard antivirus program – excessive resource use. Much of the research in this area mitigates these limitations by using a hybrid host-based and network-based solution. This combines the more ample resources of the network with the granularity of host-based reporting. In general, each mobile device is programmed with an agent that sends small reports on activities and characteristics such as resource usage and communications sessions. Table 3 lists examples of potential indicators available in these reports which are periodically forwarded to a server on the network. The server analyzes not only the individual device reports it receives, but backbone-provided data and trends as well. In most implementations, the server contacts devices as necessary, passing instructions to protect them from attack (like blacklisting infected phones [Che07][Sch091].

Table 3 - Example Reportable Features of a Mobile Device

Data Category	Example Data
Computing Resources	CPU utilization, RAM utilization
Operating Entities	Process count, Thread count
Communication Channels	Bluetooth Connections, TX status
Messaging Statistics	SMS/MMS messages sent
User Information	User activity length, inactivity length
Other Hardware	Battery charge

The benefits of outsourcing the analysis from the devices themselves includes saving battery power and achieving a broader view than any one device itself is capable of via a proxy.

While useful in minimizing damage caused by mobile malware, these methods are not proactive. Detection can only occur after an infection or misuse has already occurred.

These methods try to limit continued misuse and the infection of other phones but cannot prevent the infection itself. One system requires ~20% of all phones to be infected before detection and mitigation [Che07].

2.3.4 Other Methods

Three other types of protection have been proposed [Guo07]. They suggest a reduction in the attackable “surface” of the phone, an example of which is turning off the non-signaling computing functions when not actively used. Second, they recommend hardening the OS itself which includes incorporating measures that ensure user notification when changes are made to the phone. Third, they advocate hardening the device hardware, specifically suggesting the use of Trusted Platform Modules to protect device configuration and data. Graphical Turing Tests have also been proposed which distinguish between human and computer response by the ability to solve a visual puzzle [Xie09]. These puzzles specifically protect the phone’s messaging capability and so detect and prevent the spread of malware.

2.3.5 SecureQEMU

The security features above add a measure of security to the systems they protect. However, they have a significant undesirable characteristic, they are reactive in the case of anomaly detection, or they require user interaction like the Graphical Turing Test. They mitigate malicious activity once it is observed and identified but the underlying problem is the identification of malware itself, which may require human interaction to make the determination. The anomaly detection method falls short because it is blacklist driven - it requires that an activity match undesired characteristics, and those characteristics must be defined a priori. The Turing Test requires no knowledge beforehand, but involves its user in all decisions.

SecureQEMU does not have these limitations. It is implemented in the emulation layer and requires that all code be signed at the page level and checked against a whitelist of known good code, resulting in a closed system by default. Pages that match known good signatures are allowed to execute while those that do not are ignored, with no user interaction required [Kim09].

SecureQEMU is implemented in an emulated Windows/Intel desktop system making use of the no-execute bit in the memory paging system. The no-execute bits are originally set to zero, and the exception handler is hooked so that a no-execute exception computes a hash of the page and compares it to the whitelist. If a match is found, the bit is set and the page is allowed to execute.

2.4 Summary

This chapter reviews the vulnerabilities of and threats to smartphones, as well as a number of protection mechanisms that are either already in use or are proposed for protecting them. It examines the various protection mechanisms' efficacy and the costs they incur. The following chapter provides a methodology for implementing a mechanism similar to SecureQEMU's signed code method in a mobile device.

III. Code White

3.1 Introduction

This chapter presents Code White (Code Whitelist and Hardware Implemented Trusted Execution,) a signed code adaptation of the Symbian OS running on an ARM processor. It enforces a code whitelist by means of the execution bit available in many newer mobile processors. It operates in kernel mode and effectively protects the system from malicious user-mode code.

Running signed code provides two benefits over unsigned code. First, it verifies the author or sender of the code, and it ensures that the code has not changed since it was installed on the device. These two benefits are known as non-reputability and integrity in the CIA security model.

This chapter first describes the salient features of the ARM architecture and Symbian Operating System that make them suitable for this research. A full discussion of the design and implementation of Code White follows. Much of Section 3.3 is adapted from [Sal06].

3.2 The ARM Architecture

The ARM Architecture (ACORN RISC Machine or Advanced RISC Machine) is a reduced instruction set computer (RISC) architecture that first saw production in 1983. Originally designed for desktop computing, ARM has evolved to be the embedded architecture of choice in many areas, especially in mobile phones. Nearly 98% of mobile phones worldwide contain at least one ARM processor [Kra06].

Being a RISC architecture, simplicity is ARMs hallmark. This promotes ease of implementation and lowers power consumption which is extremely desirable in the embedded world. Additional RISC features include: load/store processing, numerous uniform access registers, and a fixed-length instruction. ARM supports seven processor modes: one user mode and six privileged modes. The Supervisor privileged mode is reserved for protected operating system processing [Kna04].

The ARM architecture is actually a family of architectures that spans 26 versions in 14 families. Each version/family extends the capabilities of previous implementations. Version six (ARMv6) in the ARM11 family has notable changes over ARMv5, including the addition of the execute-never (XN) bit to the page table permissions. The OS can set this bit in the page table entries for pages that do not contain executable code. Any attempt to fetch an instruction from such a page results in a permission fault [ARM05]. Many smartphones today use Cortex-A8 based processors from the ARMv7 architecture which inherits the XN bit from ARMv6.

3.3 The Symbian Operating System

The Symbian Operating System was designed specifically for mobile computing. Its origin traces back to an 8-bit kernel developed in the 1980s for use in personal organizers and extends to the current 32-bit kernel, EKA2. Throughout that progression, Symbian remains an embedded OS, similar to other real time OSes, but also now includes functionality that was once only found in larger desktop systems. This makes it an ideal platform for smartphones since its real time capabilities support the signaling protocols for voice and data transmission while additional functionality marshals processes and applications in a secure manner.

Symbian's kernel, EKA2, is modular, a feature that permeates the rest of the OS as well. This modularity keeps Symbian's many functional areas streamlined and simple. Built for a single user (most mobile devices are not designed for multiple users), it has a preemptive multi-tasking kernel, allowing multiple applications to run, while ensuring that each application releases the CPU as required. It is also priority based, quickly allocating resources from lower priority threads for use by higher priority threads.

To keep the kernel as small as possible, a number of services often found within the kernel are implemented using user mode servers, including all file and windowing services. A partial overview of Symbian is shown in Figure 4 (adapted from [Sal06]).

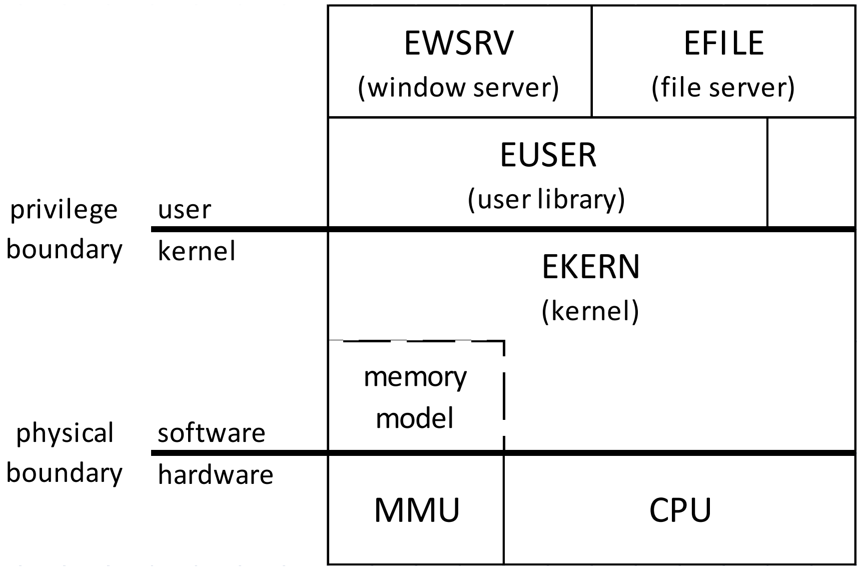


Figure 4 - An Overview of the Symbian Operating System

3.3.1 Memory Management

Symbian OS supports many kinds of devices and their associated hardware architectures. Its memory management component supports five different models. Understanding these models is important.

The Memory Management Unit (MMU) translates between physical and virtual addresses. The MMU (or lack thereof) is the leading characteristic that defines the memory model used. A list of memory models is shown in Table 4.

Before discussing the models themselves, it is important to define chunks and paging. A “chunk” is the primary unit of allocation, mapping physical RAM and other devices to contiguous virtual addresses. A chunk includes a reserved region (the set of virtual addresses available to the process) and a committed region (the physical mapped region of RAM). The kernel allocates chunks and may alter their size dynamically. Each process is allocated at least two chunks: one to hold the executable’s .data section (initialized gobal and writable static data), .bss section (zero filled data) and user-side stack space, the other to hold the main thread heap. If the executable is not loaded from ROM, the MMU allocates a third chunk for the code.

All memory models except the direct model use paged memory where processes are allocated linear, virtual memory addresses that translate to physical addresses on the RAM chip itself. Demand paging (i.e., swapping pages between memory and other storage for performance reasons) is not currently supported.

Table 4 - Symbian Memory Models and Their Characteristics

No MMU	Direct Memory Model
Virtually tagged cache	Moving Memory Model
Physically tagged cache	Multiple Memory Model
Emulator	Emulator Memory Model
Supports Symmetric Multi-Processor	Flexible Memory Model

3.3.1.1 The Moving Memory Model

The moving memory model was the most common model until the advent of the ARMv6 architecture. It is based on the use of a single page table directory for the entire OS and all processes. Processes share the virtual address space, and are accessed by moving memory chunks during context switches (i.e., changing their virtual addresses).

For security, the moving model makes use of page table permissions and domains. Page table permissions record the allowed access from user and supervisor modes (read/write/execute). Domains (up to 16 in ARMv5) provide a fast way to modify memory page rights. Each page is assigned to exactly one domain. The domain dictates whether that page is not accessible, accessible to all (ignoring table permissions), or accessible according to page permissions.

Context switches in the moving model are slow and complex. The MMU changes the page directory entry for the outgoing and incoming processes, along with the necessary domain entries and permission bits. The translation look-aside buffer (TLB) and cache are cleared to prevent false hits (as the old and new process were mapped to the same virtual addresses). The moving model uses an indexed and tagged virtual addresses cache.

3.3.1.2 The Multiple Memory Model

The multiple memory model makes improvements in performance and security. First, it uses two page directories as opposed to one in the moving model. One is global, and another is specific to the local process. For security it adds the no-execute bit to page table entries while maintaining the permissions and domains concepts from the moving model (though the use of domains has been deprecated). This ensures that data pages are never read during instruction fetching, no matter what the other permissions are. Processes receive an application space identifier (ASID) which is prepended to virtual addresses belonging to the process in structures like the TLB, thereby eliminating the need for flushing buffers during context switches. Finally, the cache is virtually indexed and physically tagged, which means that the cache does not require flushing between context switches either as memory references will always resolve to a single process.

The multiple model's name is due to keeping multiple processes mapped to memory simultaneously. Context switches are much quicker than the moving model since they involve a change to only two registers: the register that holds the page directory and the context id register.

3.3.1.3 The Direct Memory Model

This model disables the MMU so the OS is limited to direct mapping between virtual and physical addresses. Without an MMU, the memory must be divided among processes at build time since chunks cannot change at run time. Furthermore, there is no protection between kernel space and user space. For these reasons, Symbian does not support this mode for production devices. It is useful for porting software: a manufacturer may disable the MMU to simplify the debugging of other functions. Once they are stable, the MMU is re-enabled and a different memory model is used.

3.3.1.4 The Emulator Memory Model

The Emulator Model is based on a PC running Windows. This model does not interact directly with hardware to allocate memory, but interacts with the host OS via the host's APIs. It is only used to operate the WINS emulator for Symbian, not the QEMU emulator. The latter emulates actual ARM hardware on which any of the other compatible memory models may be used.

3.3.1.5 The Flexible Memory Model

The Flexible Memory Model is the newest model, supporting the ARMv7 and later families. It builds upon the Multiple Model, but adds support for the Symbian Symmetric Multiprocessor (SMP) kernel for multicore CPUs beginning with the ARM Cortex A9. At the time of this writing, documentation on this mode is limited, but departures from the Multiple Model include: 1) arrays of physical page addresses which represent storage for chunks, code, and thread stacks that replace chunks as the basic allocation entity called "memory objects". 2) Permissions and sharing properties apply to "memory mappings" rather than the chunk. One or more memory mappings may apply to one or more memory objects. 3) Processes sharing the same chunk may use different virtual addresses for access.

3.3.2 Security

Symbian security is built around the following three elements: the OS process is the unit of trust, capabilities control access to sensitive resources, and data caging protects files against unauthorized access.

3.3.2.1 Processes and Trust

There are three levels of trust within Symbian. The most trusted level is the Trusted Computing Base (TCB) which consists of the kernel, the file server, and the installer. This level is the source of all trust for the rest of the OS. It has unrestricted access to resources and has the most protection applied. Next is the Trusted Computing Environment (TCE). These modules need access to some but not all critical resources and are responsible for protecting those resources. Finally, the application layer is for processes that are not trusted to access resources directly, but which must request service via the other two levels.

Additional process protections extend beyond the basic levels of privilege above. First, privileged threads/processes like the kernel that need to access unprivileged user mode memory do so via special methods that ensure the stability of the process if invalid memory is accessed. Next, all thread memory spaces are considered private; they cannot be accessed by other processes. Additionally, new thread stacks and heaps are zeroed in memory to eliminate the chance of data leaking from an old process to a new one. The never-execute bit prevents execution of memory contents in stacks, heaps, and static data to prevent malicious code execution.

3.3.2.2 The Capability Model

Capabilities in Symbian are authorization tokens. They protect data resources by ensuring only those processes that have been approved to access them can do so. The first rule of this model is that a process' list of capabilities never changes during its lifetime. Capabilities are specified at compile time and cannot be modified thereafter. Second, no process can load a library with fewer capabilities than itself since a call to the loading library would then execute its code with the process' greater privilege.

3.3.2.3 Data Caging

Symbian's file access control system is based on directory paths. Under any drive, there are four directories under the root that denote differing access levels. Files in the \sys directory are only accessible by the TCB. These are files critical to the system: executables for example. Second, files in the \resource directory are readable by all processes, but only writeable by the TCB. These are fairly static files like images and help files. Next is the \private directory which has subdirectories for every process. Only that process and the TCB can access this directory. Any other directory under the root is considered public and has no restrictions.

3.3.3 Enhancing Symbian

Symbian OS is the platform used to implement Code White. It is open source, represents a large portion of the smartphone community, and supports the ARM processor and its associated execution bit.

Symbian already has a signing protection, so why enhance it? As discussed in Chapter 2, the signing process has been manipulated by malicious coders. Furthermore, Symbian Signed certificates only provide non-repudiation. Code White adds the XN bit to prevent untrusted code from executing and moves this from user to kernel mode enhancing its integrity.

3.4 Design

Chapter 2 explains the need and difficulty of achieving trusted execution in mobile devices. Mobile device manufacturers and network operators use a variety of methods to achieve such trust, even while running third party code. Even so, application signing has been subverted as malicious code writers also have certificates. Code White provides an alternative signed code execution method. It extends the Symbian OS loader to include an additional assurance that only trusted code may execute on the device by combining two protection mechanisms: a whitelist for executable code and execution permission bits in the page tables.

3.4.1 Signed Code Execution

The first Code White mechanism is a whitelist for executable code. Chapter 2 discusses many types of malware for mobile devices. Though diverse, they share one trait in common, they need to execute code on target devices. Using a signed code mechanism places code into two groups, trusted and untrusted. This is different than a blacklist. A blacklist explicitly identifies known malicious code and allows all of the rest to execute. This is a 'fail open' system as the default case allows the unknown code access to the system, and fails to adhere to the design principle of fail-safe defaults [Sal75]. A whitelist identifies trusted code disallowing all others, resulting in a 'fail closed' system. In this case, whether unknown code is malicious or not is unimportant, it cannot execute unless it is known and trustworthy.

3.4.1.1 Cryptographic Support

An important issue in a whitelist system is how to identify trusted code. The system could store copies of all allowable code and compare an arriving executable, but this is highly inefficient, even pointless since having copies of trusted code on the device would eliminate the need to bring them on the device in the first place. It is necessary to protect code with a smaller object that is unique, and cannot be easily counterfeited.

Code White uses a hash-based message authentication code (HMAC) to uniquely identify trusted code. Computing a cryptographic hash is a one-way function that takes data of arbitrary length and produces a 'digest' of that data. An HMAC is a cryptographic hash of both the message and a secret key. The reasons for using an HMAC are twofold. First, using a hash digest makes it infeasible for an attacker to pass malicious code into the system since a hash function is a one-way function. It is easy to compute a hash from a message, but impossible to compute a message from a hash. An attacker that knows the trusted digests is still unable to construct malicious code that results in the same digest as one on the list. Second, by using the HMAC, each hash list is unique to its user since the HMAC of each message/key pair is different than that of other users. Thus, even if an attacker were able to produce a malicious file that had a trusted HMAC, it would only be for one user. Trying to produce such a file only for one user is very inefficient.

3.4.1.2 Kernel Layer Implementation Protects User Layer

Code White protects a smartphone from untrusted user applications, including .exe files and statically linked libraries (.dll). Since it protects at the user level, it is necessary that the mechanism itself execute from the kernel level so that user code cannot affect or interfere with its performance.

Modifications to the Symbian OS that make up Code White are all within the kernel and the file server which are both part of the Trusted Computing Base. It is essential to have this mechanism which raises trust execute from the most trusted parts of the OS.

3.4.1.3 Trusting the List

The list of trusted HMACs is intended to be open. However, the HMAC of the list itself is computed at compile time for the kernel and included in the kernel binary file so the list can be validated before use. An attacker could modify the list, but then the list HMAC computed at run time would not match the original and the protection mechanism defaults to the closed state.

3.4.2 Hardware Assisted Execution Prevention

Computing a cryptographic digest and checking it against a whitelist provides the system with the necessary information to determine trustworthiness, but does nothing to enforce the decision made. Code White loads all code into memory but modifies the permissions of the code's page tables such that only *trusted* code has the bit set that allows execution. Untrusted code loaded into memory has the no execute bit set for its page tables. When the instructions are fetched from an untrusted page, a prefetch abort is raised and the appropriate actions can be taken to notify the user, isolate, monitor or report on the code in question.

3.5 Implementation

3.5.1 The Symbian Loading Process

In a standard Symbian loading process the loader operates like a server, and has portions that execute in user mode and others in kernel mode. The loading process begins when invoked by user mode functions to run executable files. An executable can be an executable application (.exe), library (.dll), driver (.ldd or .pdd), file system/system extension/system plugin or locale (all except .exe are types of dlls). The term 'executable' means any one of these file types and application denotes an .exe file only.

A TRomImage is an executable that resides in the device ROM. All symbols, relocation data and import data are discarded and the import stubs are replaced with the actual exported function addresses in ROM. They are not compressed. This enables the TRomImage to execute in-place (designated as XIP). When executing a TRomImage, the kernel allocates data and heap chunks to their processes, but the code itself executes directly from the ROM. E32Images cannot reside in ROM. If an E32Image is sent to the ROM building process, it is converted to a TRomImage. These two file types have different headers, the TRomImageHeader and E32ImageHeader.

The application loading process has six main phases, each stemming from a main function found in E32Image::LoadProcess. These are:

```
- RImageFinder::Search
- E32Image::Construct
- E32Loader::ProcessCreate
- E32Image::LoadToRam
- E32Image::ProcessImports
- E32Loader::ProcessLoaded
```

(E32Loader functions are kernel executive functions, while the others all execute in user mode.)

The following paragraphs describe the loading process for a non-execute-in-place application. RImageFinder::Search begins by finding the requested file for execution. If a complete path is not specified, it searches for any sys/bin (the required location for executable files per Symbian platform security) at each drive letter starting with Y and moving back to A, and lastly Z which is the drive letter for the device ROM image. Understanding this search is important since multiple executables may be on the device. The search function runs the first one that matches the requested filename. Thus an E32Image test.exe found on the S: drive would execute rather than one that was included in ROM unless the user requests the full path to it: Z:\sys\bin\test.exe.

The search function performs a number of activities upon finding the intended file. It collects basic information about the file and performs some sanity checking for platform security reasons. Included in the code is a sub-function to compute a hash of the file, but according to a comment in the source, the function is bypassed for performance reasons.

E32Image::Construct reads the entire E32ImageHeader and populates member variables with the header information. E32Image extends the TProcessCreateInfo which extends TCodeSegCreateInfo. The latter two are used extensively throughout the rest of the loading process.

E32Loader::ProcessCreate creates the new process object. It allocates a memory address, maps RAM for the code and then creates the data/bss/stack chunk. It adds the process to the kernel's process list and creates the main thread.

E32Image::LoadToRam then reads the code and data sections into their respective memory locations. The code is loaded into the loader's address space while the data is loaded into the new process' space. If the file is compressed by the deflate or byte pair algorithms, it is decompressed here. Once the code is loaded, LoadToRam calls a number of sub-functions to relocate the code.

E32Image::ProcessImports searches the file's import directory and loads each dll found there. RImageFinder::Search, E32Image::Construct, E32Image::LoadToRam, and E32Image::ProcessImports are invoked again for each dll, just as for an .exe. The difference is that the E32Loader functions ProcessCreate and ProcessLoaded are replaced by calls by subfunction calls CodeSegCreate and CodeSegLoaded. This occurs recursively for each .dll. Any imports that they include are loaded and checked for additional imports.

Finally, the E32Loader::ProcessLoaded function creates a new code chunk in the the new process' address space and remaps the code from the loader's address space. The loader sets the main thread state to 'ready' and returns.

3.5.2 Reading the Code Section

Figure 5 illustrates Code White's execution and shows where each part is located in the Symbian loading process. The three basic sections of Code White are reading the code, computing the HMAC, and setting the XN bit as necessary.

Access to the loading code section is the first objective of the signed code protection mechanism. Code White interacts with the Symbian loading process during the E32Image::LoadToRam stage within the E32Image::Read function as shown in Figure 5. The code and data sections are read into the memory allocated at their specified load addresses. The modified function allocates a buffer on the loader's user heap and reads a second copy of the code into the buffer.

The next step is reading the hmacs.txt file in s:\sys\bin. The full file path is specified to eliminate the need to search for it. The loader opens this file, checks its size, allocates a sufficient buffer on the user heap, and reads in the file contents.

A TCodeInfo object is created with a class unique to Code White. Four member variables are initialized, one for each buffer's size and one for each address. The function then calls E32Loader::CodeCheck, a kernel executive function created solely for Code White.

3.5.3 Computing the HMAC

The CodeCheck executive moves execution to kernel mode, in keeping with the desired kernel layer protection characteristic of Code White. As all other loader executive calls, CodeCheck performs a security check to validate it was called by the loader thread. Any other system entity fails this check, at which point the server is panicked and the kernel ends the loading process.

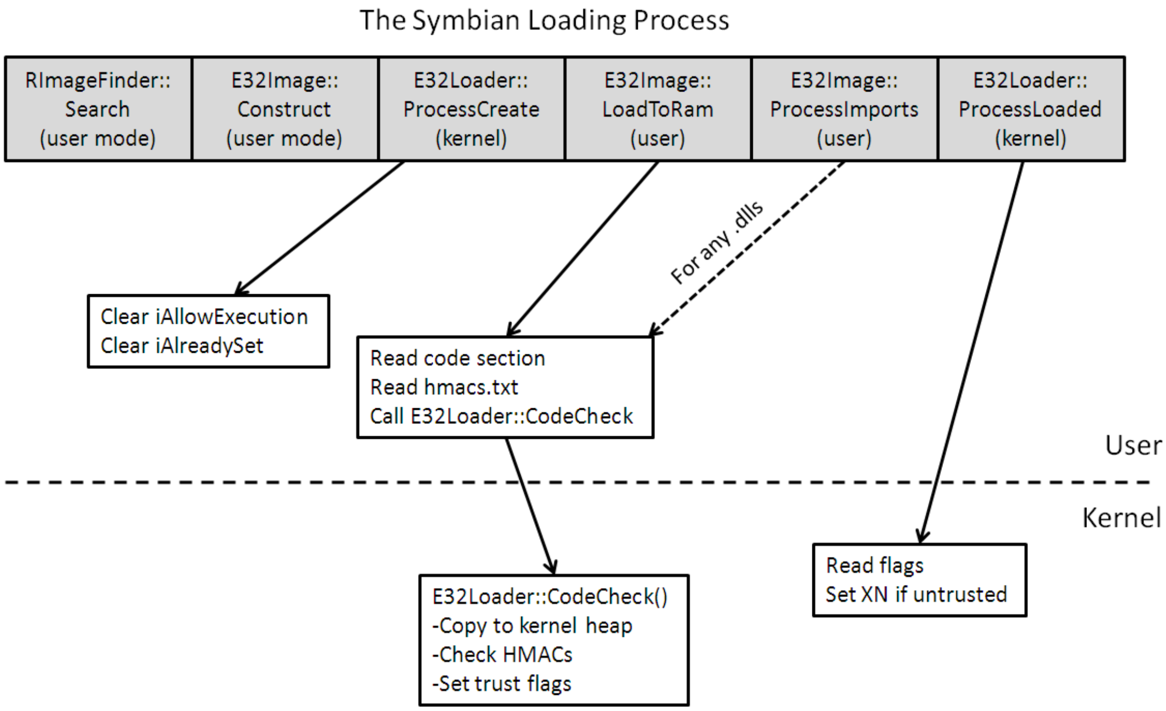

The Symbian Loading Process
RImageFinder:: Search (user mode)
E32Image:: Construct (user mode)
E32Loader:: ProcessCreate (kernel)
E32Image:: LoadToRam (user)
E32Image:: ProcessImports (user)
E32Loader:: ProcessLoaded (kernel)
For any .dlls
Clear iAllowExecution
Clear iAlreadySet
Read code section
Read hmacs.txt
Call E32Loader::CodeCheck
User
Kernel
Read flags
Set XN if untrusted
E32Loader::CodeCheck()
-Copy to kernel heap
-Check HMACs
-Set trust flags

Figure 5 - Code White Modifications to the Symbian Loading Process

After some argument marshalling, the loader creates two buffers on the kernel heap, one for the HMACs file and one for the loading code section. It copies information from the user heap to the kernel heap using the kumemget function which allows the kernel to access user memory.

The function computes the HMACs of each of the new buffers using the MD5 algorithm. The key used is a hard coded 32 byte string “12345678901234567890123456789 0AB” but in an actual system, the key is unique to the user, for instance a PKI private key. The hashing function returns a 20 byte HMAC for each buffer. It compares the HMACs file HMAC to determine if it is an exact match, and searches the HMACs file buffer to see if the code section HMAC exists within the buffer. The HMACs file is not sorted and there is no special algorithm for searching for a match, but this could optimize performance. There is no particular encoding for the HMACs file. It holds only the HMACs in their binary forms, separated by 16 bits of ‘0’s.

The results of the HMAC comparisons are recorded in a global variable: iAllowExecution. Another global variable iAlreadySet ensures that loading dependencies of varying trust levels in a particular order will not compromise protection effectiveness. All required dlls are loaded during the main executable’s loading process, which means both of these integer ‘flags’ are needed to correctly ensure that all code used by the new process is trusted. Both flags are cleared to 0 during the E32Loader::LoadProcess stage following the creation of the process object and declaration of a process ID. This initialization takes place before the main executable’s code is loaded into memory. All dll code is read into memory *after* the main executable before CodeCheck is ever called. Since the system allows at most one instance of the loader which can only load one process at a time, there are never conflicts when accessing these global flags.

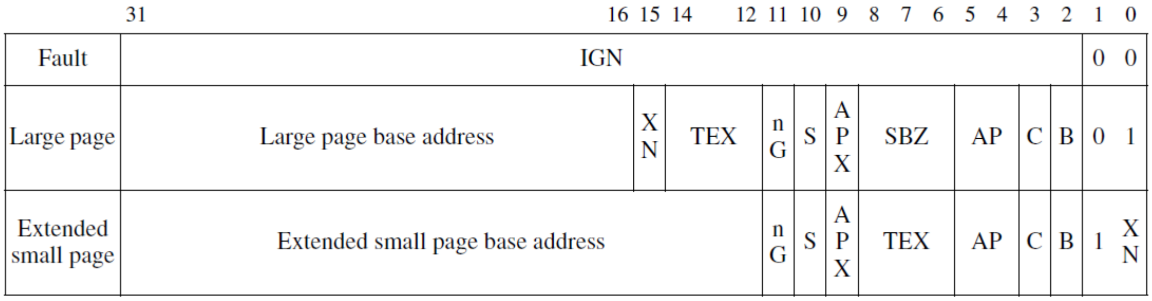
Table 5 shows the truth table for the decision process in the CodeCheck function. The AlreadySet flag is set during the code check for the new process’ main executable, and is never cleared. The AllowExecution flag can only transition from 1 to 0 if the AlreadySet flag is 0 (i.e., this is the first executable to load for this process) and both the HMACs file and the loading image code are trusted. If it is ever cleared in the CodeCheck function, subsequent calls to this function will not set it again. Once these variables are properly set, the CodeCheck function returns.

Table 5 - Truth Table for iAllowExecution and iAlreadySet

AllowExecution	AlreadySet	!IllegalHmacList & HmacIsAMatch	AllowExecution'	AlreadySet'
1	1	1	1	1
1	1	0	0	1
1	0	1	1	1
1	0	0	0	1
0	1	1	0	1
0	1	0	0	1
0	0	1	1	1
0	0	0	0	1

3.5.4 Setting the XN Bit

The final part of Code White comes in the ProcessLoaded() stage of the loading process. Here, as the loader creates the new process' code chunk, it sets the appropriate page table permissions before mapping the code. This is done in the DProcess::MapUserRamCode function. This function does a number of tasks to ensure it has the correct permissions for the chunk type. The modified function checks the global AllowExecution flag. If it is set to 1, it does nothing else. If the flag is 0, it replaces the chosen permissions with 0xA1F. Bit 0, the XN bit is shown in Figure 6. (Note the Multiple Memory model uses small pages by default.)



	31	16	15	14	12	11	10	9	8	7	6	5	4	3	2	1	0
Fault	IGN															0	0
Large page	Large page base address		XN	TEX		nG	S	APX	SBZ			AP		C	B	0	1
Extended small page	Extended small page base address					nG	S	APX	TEX			AP		C	B	1	XN

Figure 6 - Symbian Page Table Entry Format

The permissions 0xA1F indicate the chunk is neither shared nor global, and that the supervisor may read the chunk while user has no access. The untrusted code is secured, unavailable for instruction fetching, but is available for analysis if desired.

3.5.5 Limitations

The Code White implementation includes a number of limitations, either by design to manage the scope of the effort, or as a result of difficulties encountered in its creation. Code White protects a smartphone against untrusted user applications (.exe) and any associated, untrusted, statically linked libraries (.dll). It does not protect against the various other types of executable files that run on the Symbian/ARM system including drivers and locales. It does not protect against untrusted or malicious files that execute from the ROM. It is assumed these files are verified and trusted. Furthermore, if these files are compromised, then there is not a basis for trusting Code White itself. An attacker that can access the files before ROM creation is assumed to have a persistant presence on the device; no protection mechanism can successfully provide trust for such a device.

The next limitation is the requirement to create two extra copies of the code section. Ideally, the executive function E32Loader::CodeCheck should be able to read any user memory space, including the allocated space for the loading process' code. But in practice, this was not achieved, even using the Symbian built Kern::AccessCode() function which places a mutex on the code segment to allow access. Reading a copy to the user heap and later copying it to the kernel heap is a workaround, and affects Code White's performance. Even if this were not an issue, the call to check the code would remain in the E32Image::Read function since at that point the newly read code section has not been relocated. Performing the code check after relocation is not feasible since modified pointers throughout the code would change its HMAC resulting in a miss while searching the hmacs.txt file.

To simplify Code White's design, support was removed for compressed images. Standard Symbian supports three types of file compression: the Huffman Deflate algorithm, the bytepair compression algorithm and no compression. A small modification to the E32Image::Construct function checks the compression type and returns KErrNotSupported if the file is compressed. Adding support for compressed files would be simple if the ability to access user code from the CodeCheck executive function is added. The call to CodeCheck would only need to be invoked between the time the code section is uncompressed and the time that it is relocated.

Another improvement would be to load the HMACs file only once. Currently it is loaded each time a code section loads to RAM. Allocating a persistent area in the kernel's memory would provide a modest performance gain.

3.6 Conclusion

With a brief introduction to both the Symbian Operating System and the ARM Architecture, this chapter presents Code White, a signed code mechanism to promote trusted execution of third party applications. It should have minimal impact on device performance. Since reading extra copies of the code and computing their respective HMACs while loading requires additional time, it will take longer to load than a standard build of the same OS, most likely adding an overhead of 50%-200%. However any decrease in performance is expected to be suffered only at load time, and will not affect the steady state operation of the trusted applications.

IV. Code White Performance

4.1 Approach

To measurably achieve the goal of improving smartphone security, it is necessary to execute Code White and evaluate its ability to prevent the execution of untrusted code while minimizing processing overhead. The methodology to determine this consists of two rounds of testing: one to evaluate the protection mechanism, the other to detect any change in performance compared to an unmodified system. Each system attempts to execute approved and unapproved code to verify expected operation. A small group of applications serve as the trusted set for Code White's HMAC table while several others are the untrusted input.

To illustrate successful protection it is expected that Code White will allow all trusted applications (.exe) and their statically loaded libraries (.dll) to load and execute, as long as both the HMACs of the application and all libraries are in the Code White hmacs.txt file. Further, Code White will not execute any combination of .exe/.dll files that includes an untrusted file.

It is expected that Code White will add significant overhead to the loading process compared to an unmodified system and that this overhead will grow linearly as the number of libraries and the code size increases.

4.2 System Under Test

The system under test (SUT) is the smartphone application protection system, shown in Figure 7. Though the system interacts with other entities, (user, network, etc.) the system is limited to the handheld device itself.

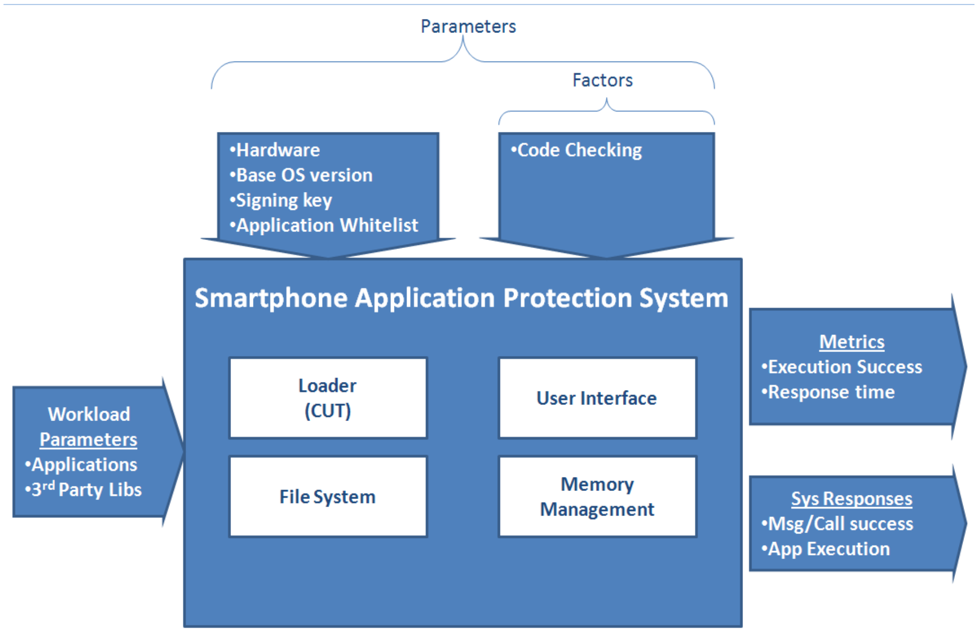


Figure 7 - Smartphone Application Protection System

The system includes four components. The loader loads executable files and performs other functions to ensure system security during and after the loading process. Code White's implementation within the loader makes it the component under test. The user interface provides some security by validating user input. The memory management system applies security policies to protect code and data in main memory. Finally, the file system includes many protections for offline files in the device. These components provide two services, system and data security. Table 6 shows various outcomes that result from these services.

Table 6 - Protection System Services and Outcomes

Service	Outcome
System Security	Untrusted code: Executed, Blocked System settings: Correct, Incorrect Malformed input: Allowed, Blocked
Data Security	Data protected, Data compromised (unauthorized access)

To measure Code White's successful operation and performance requires two metrics. First, the handling of trusted/untrusted files yields a Boolean result - either the subject application executes or it does not. Second, the response time for each loading application serves as the performance indicator. Both the standard Symbian platform and the modified version contain a timer to provide this metric.

4.3 System Parameters, Factors & Workload

The SUT includes four fixed parameters for the testing procedure: hardware, the OS base version, the security key, and the application whitelist. The hardware is virtual, emulated by the Symbian-modified Quick Emulator (QEMU) known as the 'Syborg Virtual Platform' (hereafter 'syborg'). Syborg emulates an ARM Cortex-A8 processor and 128 MB of RAM. The Cortex-A8 supports the multiple memory model and thus the XN bit required for Code White. Though Code White's design is centered around the hardware XN bit, it is a software modification. The hardware configuration is the same for all tests.

The base OS version in both the standard and modified images is Symbian 3.0.h. Keeping the same base ensures that only the Code White modification affects the performance metrics.

The secure key used to calculate the HMACs in Code White could affect performance if its length is varied between tests. The key for all tests is 32 bytes: ‘1234567890123456789012345678 90AB’.

The length of the HMAC whitelist file is fixed since a longer file requires longer searches to find a valid HMAC or determine that the application is untrusted. The whitelist for the tests contains only the HMACs of the fifteen trusted files (identified below).

Code checking is a system testing factor. Either Code White is present or not. Meanwhile, the workload consists of the applications presented to the system along with their associated libraries. The size of the application code, number of dlls, and trust level of the files are workload factors.

The .exe file sizes range across four values: 1, 10, 50 and 100kB, represented by s, m, l and x respectively in the first character of the file name. Each has a number of dependencies (dlls) from none to two as shown in Table 7, columns labeled 0, 1 and 2.

Table 7 - Workload Executables

Size/Dlls	0	1	2
1kB	s0.exe	s1.exe	s11.exe
10kB	m0.exe	m1.exe	m11.exe
50kB	l0.exe	l1.exe	l11.exe
100kB	x0.exe	x1.exe	x11.exe

The workload also includes three libraries: d1.dll, d2.dll and d3.dll. All three are ~1kB in size. The three applications named _11.exe explicitly include d1.dll and implicitly include d2.dll. In other words, d1 has a dependency on d2. The three _1.exe files include only d3.

The untrusted executable files are duplicates of the trusted versions that have a few bytes changed in their text sections. All of the executables print a few messages to the console, so the simplest change to make is to change the message. For instance, “press any key” is changed to “break any key”. This keeps the file and section sizes the same, and does not require any change to the file headers. Both .exe and .dll files have trusted and untrusted versions. Since untrusted files never execute, they are not used for performance testing, but only to validate the protection mechanism.

Table 8 shows the four experimental factors and their levels used in both test groups. Code checking is fixed ‘on’ for testing protection since all files are expected to execute on standard Symbian, which was verified before running the protection tests. File trust level is ‘trusted’ for all files used in the performance test as the loading performance of an untrusted application that will fail to execute is not important to the user experience. In all cases of failed execution caused by Code White blocking an untrusted file, an error message was returned to the user who could subsequently type another command into the console. This is acceptable for the case of a blocked file.

Table 8 - Experimental Factor Levels

Factor	**Levels**	**Protection Testing**	**Performance Testing**
Code Checking	On/Off	Fixed (On)	Varied
File Size (kB)	1, 10, 50, 100	Varied	Varied
Number of dlls	0, 1, 2	Varied	Varied
File Trust Level	Trusted, Untrusted	Varied	Fixed (Trusted)

With these levels, protection testing includes 56 experiments, the result of multiplying 4 file sizes by the sum of 2 raised to the total number of files required for each application, since all combinations of trusted/untrusted files are tested for execution success. Thus 8 tests for all combinations with no dlls, 16 for all combinations with 1 dll and 32 for all combinations with 2 dlls yields the 56 tests.

Performance testing requires 72 tests, due to varying the 2 code checking levels, 4 file sizes and 3 dll levels. All tests are repeated 3 times.

4.4 Evaluation Technique and Experimental Design

To measure the response time, both the modified and standard systems include two TTime objects in the RProcess::Create function. RProcess::Create invokes the loader itself which contains Code White. When it returns, the newly loaded application executes. Thus, this function captures the start and stop objects for calculating the application response time. The addition of the TTime object with its start and stop instructions is the only modification to the standard Symbian platform.

To calculate the response time, a start object and stop object each set themselves to the system time. The objects themselves support a resolution of microseconds. However, the Syborg platform only offers a usable precision of 1 millisecond. Because of such a low resolution, and since the emulator does not emulate any of a smartphone's communications functions (network overhead, voice calls, SMS, etc), the expected variance is extremely small.

The host machine running Syborg is an Intel Xeon 5160 quad core desktop (3.0 GHz) with 3.25GB RAM. The host OS is Windows XP SP3.

4.5 Protection Testing

Table 9 - Protection Experimental Results

Test	.exe	.dll	.dll	Loads?	Test	.exe	.dll	.dll	Loads?
1	s0			yes	29	s11	d1	d2	no
2	s0			no	30	s11	d1	d2	no
3	m0			yes	31	s11	d1	d2	no
4	m0			no	32	s11	d1	d2	no
5	l0			yes	33	m11	d1	d2	yes
6	l0			no	34	m11	d1	d2	no
7	x0			yes	35	m11	d1	d2	no
8	x0			no	36	m11	d1	d2	no
9	s1	d3		yes	37	m11	d1	d2	no
10	s1	d3		no	38	m11	d1	d2	no
11	s1	d3		no	39	m11	d1	d2	no
12	s1	d3		no	40	m11	d1	d2	no
13	m1	d3		yes	41	l11	d1	d2	yes
14	m1	d3		no	42	l11	d1	d2	no
15	m1	d3		no	43	l11	d1	d2	no
16	m1	d3		no	44	l11	d1	d2	no
17	l1	d3		yes	45	l11	d1	d2	no
18	l1	d3		no	46	l11	d1	d2	no
19	l1	d3		no	47	l11	d1	d2	no
20	l1	d3		no	48	l11	d1	d2	no
21	x1	d3		yes	49	x11	d1	d2	yes
22	x1	d3		no	50	x11	d1	d2	no
23	x1	d3		no	51	x11	d1	d2	no
24	x1	d3		no	52	x11	d1	d2	no
25	s11	d1	d2	yes	53	x11	d1	d2	no
26	s11	d1	d2	no	54	x11	d1	d2	no
27	s11	d1	d2	no	55	x11	d1	d2	no
28	s11	d1	d2	no	56	x11	d1	d2	no

*Grey files are untrusted

Table 9 shows the results of the 56 tests to validate the efficacy of the protection mechanism. White cells represent trusted files, while untrusted files are shown in grey. For each run, the S:\sys\bin directory only contains the files for that test to ensure the correct combinations executed. Only tests that include all white cells loaded and executed which is the desired response. Thus, Code White successfully allows trusted code to execute and prevents untrusted code.

4.6 Performance Testing

The results of the 72 performance tests are shown in Appendix A. Figure 8 is a difference plot of the mean load times for an application with no .dlls loading on both Symbian and Code White.

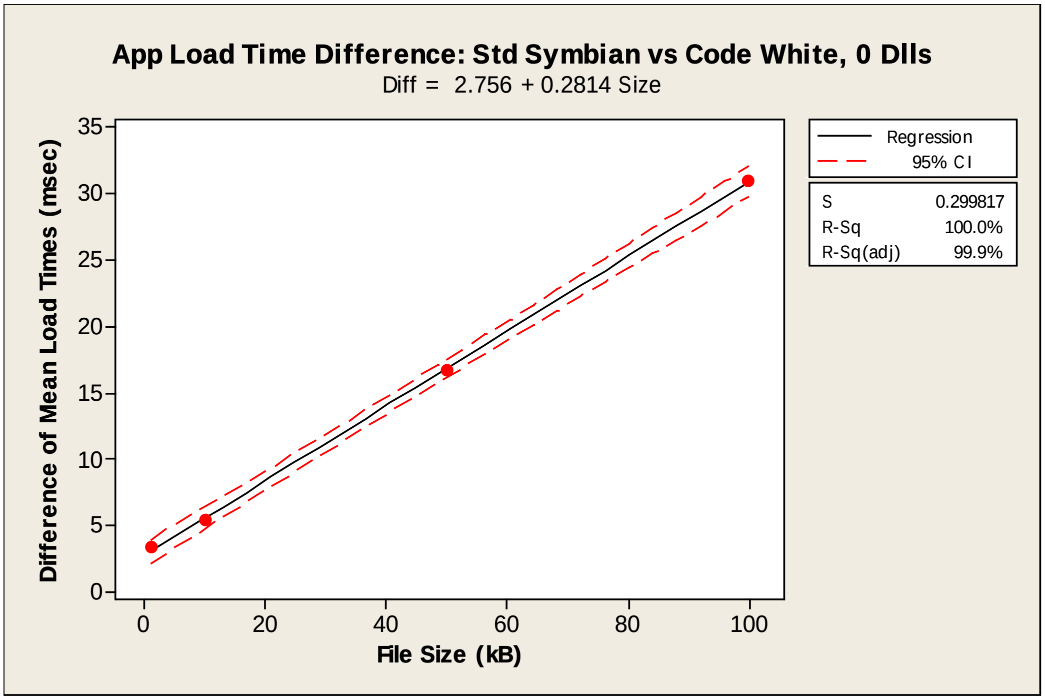


Figure 8 - Difference In Application Load Times

As expected, the difference in mean load times is linear as both systems show a linear increase in load time in response to increasing file size. Likewise, increasing the number of .dlls produces a similar response, linear according to the size of the .dlls being loaded as shown in Figure 9.

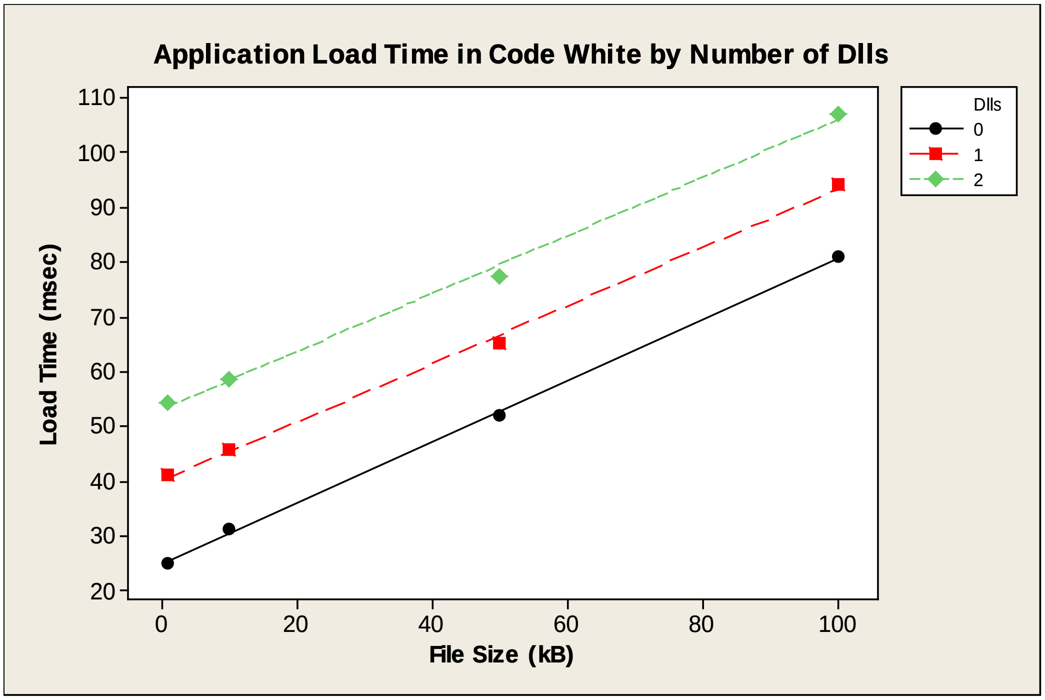


Figure 9 - Application Load Time in Code White by Number of Dlls

The regression model for Code White's load time performance is

```
Load Time = 25.8 + 0.540 * Size + 13.5 * Dlls        (1)
```

where 'Size' is the sum of the .exe and all associated .dll code section sizes and 'Dlls' is the number of .dll files included. The sum of squares given by an analysis of variance test yields 20,443 as the regression sum of squares and 52 for that of the residual error. The P value for the regression is less than .001 which gives convincing evidence that the the regression model fits the data well.

Based on (1) an application with a 1MB code section and no dlls would load in 566 milliseconds using Code White (95% prediction interval 555 to 577 msec) while one with two dlls would take 593 milliseconds (95% interval 582 to 604 msec). Both times are less than the one second limit proposed as the most a user will maintain uninterrupted thought [Nie94]. While it is not fast enough for the user to feel the machine is instantaneously responsive, it is practical enough for general use. Knowing the load time of a one megabyte code section is important as many applications currently remain below this limit. Meanwhile the 95% confidence interval for an unmodified Symbian image to load the same applications puts the 1MB load time between 243 and 354 milliseconds. Code White's performance is therefore noticeably slower than the alternative (~133% slower for 1MB with no dlls,) but it is still acceptable for the user.

4.7 Summary

Compared to a standard Symbian image running on QEMU, Code White incurs a linearly increasing overhead based on the size of the executable code that is loaded. For applications up to 1MB, the total load time is still within usable limits for the user. Chapter 5 proposes several ways to improve Code White's performance as well as related work.

V. Conclusion

5.1 Research Accomplishments

This research explores the emerging area of smartphones, their uses and vulnerabilities, and the current and potential threats to them, mostly brought on by their capability to execute arbitrary third party code. A signed code mechanism called Code White was developed and implemented in a Symbian OS kernel running on QEMU. (A diff output in Appendix B identifies the modifications made to the original Symbian source code.) Code White's performance while loading executable files was compared to that of a standard Symbian system.

Smartphones can communicate over many different protocols and mediums, thus offering many attack vectors for malicious actors. However, rather than seeking to protect against these many diverse methods, Code White addresses and protects the common capability malicious code cannot succeed without: the ability to execute.

Code White successfully prevents the execution of untrusted code while imposing an acceptable increase in load time of 43 milliseconds for a 100kB file with 2 dlls, and a predicted increase of 300 milliseconds for a 1MB file with 2 dlls (95% prediction interval). A user may use a phone running Code White without experiencing any disruptions to their normal activities. Furthermore, resource usage is kept to a minimum. A few extra milliseconds of loading are required of the CPU and batteries, which is much less than that required by using antivirus. The ability to stop untrusted code meets the research goal of improving security for smartphones.

5.2 Research Significance

Realizing that there is a balance between security and usability, Code White may not be a popular protection mechanism for the mass market of consumer smartphones. Many choose to buy these phones based upon the capability to crawl through any number of applications, trying and buying many of them. Such a use case is the antithesis of Code White. Certainly developers and vendors could provide HMACs of these applications, but two limitations stand out: providing large HMAC lists will affect performance, and worse, malicious coders have been able to receive valid signatures in the application signing process. Creating a more robust signed code mechanism does nothing to fill holes in the trust granting procedure.

However, there are many corporate and government areas that could benefit from Code White. Many such organizations have policies that explicitly list what applications may run on company or government equipment. These lists are generally short and relatively static. Thus the HMAC lists would be short and infrequently used. Corporate and government trust granting mechanisms are more capable and more highly motivated to use closer scrutiny, and the distribution processes can be more carefully guarded than that of the general consumer public.

5.3 Future Research

The Symbian Operating System was released as open source in early 2010. Learning the inner workings of an unknown OS - even one for mobile devices - takes time. Perfecting any modifications to that OS take even longer. There are a number of improvements that can be made to Code White and there is additional research that can be done.

Code White makes three copies of the loading application's code to get it to the code checking executive function. Fixing this need would greatly improve its performance, possibly putting its performance very close to that of the unmodified OS. This should be done with care to ensure the protection of the kernel while accessing user code directly.

The hmacs.txt file used in testing only contained the fifteen HMACs that represented the fifteen trusted files tested. The performance change resulting from the HMAC list size could be determined by testing with a larger list. It is expected that any increase in load time due to a larger list (hundreds of HMACs) would be small compared to increasing the file size. If that is not the case, and a larger list adds significant overhead to the load time, the list could be sorted, and Code White could be modified with a more efficient search algorithm to lower the search time.

This research implemented Code White in an emulated environment (QEMU). One limitation of this was the timers that measured performance were also emulated and had a very low precision. Implementation on a hardware device will result in more precise results. Testing on an actual smartphone while it executes its many other capabilities including network interaction with voice and data connections running would be very valuable.

There are other types of executable files that are not currently checked by Code White, including drivers and locales. Some of them are loaded directly into the kernel. While checking their HMAC is trivial, additional study is required to determine how best to isolate them if they are found to be untrusted.

Code White could be used to detect and capture malicious code for mobile devices. A number of smartphones or emulated phones could be scripted to browse mobile web sites, and devices could be placed in high traffic areas, with the purpose of being a ‘victim’ of malicious code. For high traffic areas, devices could make themselves discoverable in as many modes as possible, to see if connection attempts are made and any files are pushed to them. Using the phones could gather files with untrusted/unrecognized code sections as a type of early warning system for mobile malware.

The Code White mechanism should also be tested on devices that are not typically thought of as smartphones. For instance, devices like the General Dynamics GD300X wearable computer is essentially a smartphone developed for ground troops to wear [Cra10]. Such devices will certainly be targeted by adversaries.

Mobile phones are already an enabler in our military. Once frowned upon, they are now even issued to commanders - not just phones that make and take calls, but smartphones that can connect to other devices and run programs. The same is true of business and civic leaders. Current trends show more business at these levels being done using the computing power of these devices as opposed to the desktop computers in their offices. Protecting these critical devices is as important as protecting other types of computing systems. While antivirus and similar protection methods are not up to the task, Code White has shown a secure system can be provided with acceptable limits of usability and resource usage.

Appendix A. Performance Testing Data

	Code White			
	Run1	**Run2**	**Run3**	**Mean**
s0	25	25	25	25
m0	31	32	31	31.33333
l0	52	52	52	52
x0	81	81	81	81
s1	41	41	41	41
m1	46	45	46	45.66667
l1	65	65	65	65
x1	94	94	94	94
s11	54	55	54	54.33333
m11	58	58	60	58.66667
l11	78	77	77	77.33333
x11	107	107	107	107

	Standard			
	Run1	**Run2**	**Run3**	**Mean**
s0	22	21	22	21.66667
m0	27	25	26	26
l0	36	35	35	35.33333
x0	52	49	49	50
s1	33	32	32	32.33333
m1	34	34	34	34
l1	43	44	43	43.33333
x1	60	58	57	58.33333
s11	38	39	39	38.66667
m11	43	41	40	41.33333
l11	50	51	50	50.33333
x11	66	64	64	64.66667

Appendix B. Code White and Symbian PDK 3.0.h Diff

```
diff -r 2ee5df201f60 kernel/eka/euser/us_ksvr.cpp
--- a/kernel/eka/euser/us_ksvr.cpp  Mon Mar 08 11:58:34 2010 +0000
+++ b/kernel/eka/euser/us_ksvr.cpp  Thu Aug 26 11:46:14 2010 -0400
@@ -4892,7 +4892,20 @@
      RLoader loader;
      TInt r=loader.Connect();
      if (r==KErrNone)
+           {
+           TTime OrigTime;
+           OrigTime.HomeTime();//Start time for loading process
+           TTime timetwo;
+           TInt64 us;
+           TTimeIntervalMicroSeconds ttims;
+

      r=loader.LoadProcess(iHandle,aFileName,aCommand,aUidType,aType);
+
+           timetwo.HomeTime();//End time for loading process
+           ttims=timetwo.MicroSecondsFrom(OrigTime);
+           us=ttims.Int64();
+           RDebug::Printf("%S loaded in %d
microseconds",&aFileName,us);
+           }
      loader.Close();
      return r;
      }
diff -r 2ee5df201f60 kernel/eka/include/e32ldr_private.h
--- a/kernel/eka/include/e32ldr_private.h Mon Mar 08 11:58:34 2010
+0000
+++ b/kernel/eka/include/e32ldr_private.h Thu Aug 26 11:46:14 2010 -
0400
@@ -308,6 +308,15 @@
      TInt64 iStartAddress;
      TInt iDriveNumber;
      };
+
+class TCodeInfo  //Class exclusive to Code White
+     {
+public:
+     TInt iSize; //size of code section
+     TUint8* iCLA; //pointer to user heap buffer holding code section
+     TInt iHLSize; //size of hmac whitelist
+     TUint8*     iHmacList; //pointer to user heap buffer holding hmac
whitelist
+     };

 //
 // Loader magic executive functions
@@ -333,7 +342,7 @@
      IMPORT_C static TAny* ThreadProcessCodeSeg(TInt aHandle);
      IMPORT_C static void ReadExportDir(TAny* aHandle, TUint32*
aDest);
      IMPORT_C static TInt LocaleExports(TAny* aHandle,
TLibraryFunction* aExportsList);
-
+     IMPORT_C static TInt CodeCheck(TProcessCreateInfo& aInfo,
TCodeInfo* aCodeInfo);
 #ifdef __MARM__
      IMPORT_C static void GetV7StubAddresses(TLinAddr& aExe, TLinAddr&
aDll);
      static TInt V7ExeEntryStub();
diff -r 2ee5df201f60 kernel/eka/kernel/execs.txt
--- a/kernel/eka/kernel/execs.txt   Mon Mar 08 11:58:34 2010 +0000
+++ b/kernel/eka/kernel/execs.txt   Thu Aug 26 11:46:14 2010 -0400
@@ -2437,6 +2437,16 @@
      arg1 = TDes8&
 }

+slow {
+     name = CodeCheck
+     user = E32Loader
+     export
+     return = TInt
+     arg1 = TProcessCreateInfo&
+     arg2 = TCodeInfo*
+}
+
+


/***********************************************************************
********
  * End of normal executive functions
diff -r 2ee5df201f60 kernel/eka/kernel/scodeseg.cpp
--- a/kernel/eka/kernel/scodeseg.cpp      Mon Mar 08 11:58:34 2010
+0000
+++ b/kernel/eka/kernel/scodeseg.cpp      Thu Aug 26 11:46:14 2010 -
0400
@@ -19,6 +19,370 @@
 #include <e32uid.h>
 #include "execs.h"

+/* GLOBAL.H - RSAREF types and constants (added for Code White)
+ */
+
+/* PROTOTYPES should be set to one if and only if the compiler
supports
+  function argument prototyping.
+The following makes PROTOTYPES default to 0 if it has not already
+  been defined with C compiler flags.
+ */
+#ifndef PROTOTYPES
+#define PROTOTYPES 0
+#endif
+
+/* POINTER defines a generic pointer type */
+typedef unsigned char *POINTER;
+
+//#if 1
+/* UINT2 defines a two byte word */
+typedef unsigned short int UINT2;
+
+/* UINT4 defines a four byte word */
+typedef unsigned long int UINT4;
+
+typedef struct {
+  UINT4 state[4];                                   /* state (ABCD) */
+  UINT4 count[2];        /* number of bits, modulo 2^64 (lsb first) */
+  unsigned char buffer[64];                         /* input buffer */
+} MD5_CTX;
+
+void MD5Transform (UINT4 state[], unsigned char block[]);
+void Encode(unsigned char *output, UINT4 *input, unsigned int len);
+void MD5Init (MD5_CTX *context);
+void MD5Final (unsigned char digest[], MD5_CTX *context);
+void MD5Update (MD5_CTX *context, unsigned char *input, unsigned int
inputLen);
+void Decode (UINT4 *output,unsigned char *input,unsigned int len);
+
+/* PROTO_LIST is defined depending on how PROTOTYPES is defined above.
+If using PROTOTYPES, then PROTO_LIST returns the list, otherwise it
+  returns an empty list.
+ */
+#if PROTOTYPES
+#define PROTO_LIST(list) list
+#else
+#define PROTO_LIST(list) ()
+#endif
+
+/* MD5.H - header file for MD5C.C
+ */
+
+/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
+rights reserved.
+
+License to copy and use this software is granted provided that it
+is identified as the "RSA Data Security, Inc. MD5 Message-Digest
+Algorithm" in all material mentioning or referencing this software
+or this function.
+
+License is also granted to make and use derivative works provided
+that such works are identified as "derived from the RSA Data
+Security, Inc. MD5 Message-Digest Algorithm" in all material
+mentioning or referencing the derived work.
+
+RSA Data Security, Inc. makes no representations concerning either
+the merchantability of this software or the suitability of this
+software for any particular purpose. It is provided "as is"
+without express or implied warranty of any kind.
+These notices must be retained in any copies of any part of this
```

```
+documentation and/or software.
+ */
+
+//static char rcsid[] = "$Id: md5c.c,v 1.2 1999/08/25 21:45:14 lennox
Exp $";
+
+/* Constants for MD5Transform routine.
+ */
+
+#define S11 7
+#define S12 12
+#define S13 17
+#define S14 22
+#define S21 5
+#define S22 9
+#define S23 14
+#define S24 20
+#define S31 4
+#define S32 11
+#define S33 16
+#define S34 23
+#define S41 6
+#define S42 10
+#define S43 15
+#define S44 21
+
+static unsigned char PADDING[64] = {
+  0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
+  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
+  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
+};
+
+/* F, G, H and I are basic MD5 functions.
+ */
+#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
+#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
+#define H(x, y, z) ((x) ^ (y) ^ (z))
+#define I(x, y, z) ((y) ^ ((x) | (~z)))
+
+/* ROTATE_LEFT rotates x left n bits.
+ */
+#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))
+
+/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4.
+Rotation is separate from addition to prevent recomputation.
+ */
+#define FF(a, b, c, d, x, s, ac) { \
+ (a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
+ (a) = ROTATE_LEFT ((a), (s)); \
+ (a) += (b); \
+  }
+#define GG(a, b, c, d, x, s, ac) { \
+ (a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
+ (a) = ROTATE_LEFT ((a), (s)); \
+ (a) += (b); \
+  }
+#define HH(a, b, c, d, x, s, ac) { \
+ (a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
+ (a) = ROTATE_LEFT ((a), (s)); \
+ (a) += (b); \
+  }
+#define II(a, b, c, d, x, s, ac) { \
+ (a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
+ (a) = ROTATE_LEFT ((a), (s)); \
+ (a) += (b); \
+  }
+
+/* MD5 initialization. Begins an MD5 operation, writing a new context.
+ */
+void MD5Init (MD5_CTX *context)
+                                       /* context */
+{
+  context->count[0] = context->count[1] = 0;
+  /* Load magic initialization constants.
+*/
+  context->state[0] = 0x67452301;
+  context->state[1] = 0xefcdab89;
+  context->state[2] = 0x98badcfe;
+  context->state[3] = 0x10325476;
+}
+
+/* MD5 block update operation. Continues an MD5 message-digest
+  operation, processing another message block, and updating the
+  context.
+ */
+void MD5Update (MD5_CTX *context, unsigned char *input, unsigned int
inputLen)
+{
+  unsigned int i, index, partLen;
+
+  /* Compute number of bytes mod 64 */
+  index = (unsigned int)((context->count[0] >> 3) & 0x3F);
+
+  /* Update number of bits */
+  if ((context->count[0] += ((UINT4)inputLen << 3)) < ((UINT4)inputLen
<< 3))
+    context->count[1]++;
+  context->count[1] += ((UINT4)inputLen >> 29);
+
+  partLen = 64 - index;
+
+  /* Transform as many times as possible.  */
+  if (inputLen >= partLen) {
+    memcpy((TAny*)&context->buffer[index], (const TAny*)input,
(TInt)partLen);
+    MD5Transform (context->state, context->buffer);
+
+    for (i = partLen; i + 63 < inputLen; i += 64)
+      MD5Transform (context->state, &input[i]);
+
+    index = 0;
+  }
+  else
+    i = 0;
+
+  /* Buffer remaining input */
+  /* fixed by Akira Tsukamoto 04/04/2002 */
+  if (i >= inputLen)
+    return;
+  /* end fix */
+  memcpy
+    ((POINTER)&context->buffer[index], (POINTER)&input[i],
+    inputLen-i);
+}
+
+/* MD5 finalization. Ends an MD5 message-digest operation, writing the
+  the message digest and zeroizing the context.
+ */
+void MD5Final (unsigned char digest[], MD5_CTX *context)/* context */
+{
+  unsigned char bits[8];
+  unsigned int index, padLen;
+
+  /* Save number of bits */
+  Encode (bits, context->count, 8);
+
+  /* Pad out to 56 mod 64. */
+  index = (unsigned int)((context->count[0] >> 3) & 0x3f);
+  padLen = (index < 56) ? (56 - index) : (120 - index);
+  MD5Update (context, PADDING, padLen);
+
+  /* Append length (before padding) */
+  MD5Update (context, bits, 8);
+
+
+  /* Store state in digest */
+  Encode (digest, context->state, 16);
+
+  /* Zeroize sensitive information. */
+  memset ((POINTER)context, 0, sizeof (*context));
+} /* MD5final */
+
+
+/* MD5 basic transformation. Transforms state based on block.
+ */
+void MD5Transform (UINT4 state[], unsigned char block[])
+{
+  UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];
+
+  Decode (x, block, 64);
+
+  /* Round 1 */
+  FF (a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
+  FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
+  FF (c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
+  FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
+  FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
+  FF (d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
+  FF (c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */
+  FF (b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */
+  FF (a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */
+  FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */
+  FF (c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
+  FF (b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
+  FF (a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
+  FF (d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
+  FF (c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
+  FF (b, c, d, a, x[15], S14, 0x49b40821); /* 16 */
+
+  /* Round 2 */
+  GG (a, b, c, d, x[ 1], S21, 0xf61e2562); /* 17 */
+  GG (d, a, b, c, x[ 6], S22, 0xc040b340); /* 18 */
+  GG (c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
+  GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */
+  GG (a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */
+  GG (d, a, b, c, x[10], S22,  0x2441453); /* 22 */
+  GG (c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
+  GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */
+  GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */
+  GG (d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
+  GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */
+  GG (b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */
+  GG (a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
+  GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */
+  GG (c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */
+  GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */
+
+  /* Round 3 */
+  HH (a, b, c, d, x[ 5], S31, 0xfffa3942); /* 33 */
+  HH (d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */
+  HH (c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
+  HH (b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
+  HH (a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */
+  HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */
+  HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */
+  HH (b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
+  HH (a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
+  HH (d, a, b, c, x[ 0], S32, 0xeaa127fa); /* 42 */
+  HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */
+  HH (b, c, d, a, x[ 6], S34,  0x4881d05); /* 44 */
+  HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */
+  HH (d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
+  HH (c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
+  HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */
+
+  /* Round 4 */
+  II (a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */
+  II (d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */
+  II (c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
+  II (b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */
+  II (a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
+  II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */
+  II (c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
+  II (b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */
+  II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */
+  II (d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
+  II (c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */
+  II (b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
+  II (a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */
+  II (d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
+  II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb); /* 63 */
+  II (b, c, d, a, x[ 9], S44, 0xeb86d391); /* 64 */
+
+  state[0] += a;
+  state[1] += b;
+  state[2] += c;
+  state[3] += d;
+
+  /* Zeroize sensitive information. */
+  memset ((POINTER)x, 0, sizeof (x));
+}
+
+/*
+ * Encodes input (UINT4) into output (unsigned char). Assumes len is
+ * a multiple of 4.
+ */
+void Encode(unsigned char *output, UINT4 *input, unsigned int len)
+{
+  unsigned int i, j;
+
+  for (i = 0, j = 0; j < len; i++, j += 4) {
+ output[j] = (unsigned char)(input[i] & 0xff);
+ output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
+ output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
+ output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
+  }
+}
+
+/*
+ * Decodes input (unsigned char) into output (UINT4). Assumes len is
+ * a multiple of 4.
+ */
+void Decode (
+UINT4 *output,
+unsigned char *input,
+unsigned int len)
+{
+  unsigned int i, j;
+
+  for (i = 0, j = 0; j < len; i++, j += 4)
+ output[i] = ((UINT4)input[j]) | (((UINT4)input[j+1]) << 8) |
+   (((UINT4)input[j+2]) << 16) | (((UINT4)input[j+3]) << 24);
+}
+
+/* RFC 2202 HMAC-MD5 */
```

```
+
+#define MD5_DIGESTSIZE  16
+#define MD5_BLOCKSIZE   64
+
+#ifdef __cplusplus
+extern "C" {
+#endif
+
+#ifdef __cplusplus
+}
+#endif
+
+
+/* RFC 2202 HMAC-MD5 */
+
+void truncate
+(
+unsigned char*   d1,   /* data to be truncated */
+unsigned char*   d2,   /* truncated data */
+int     len   /* length in bytes to keep */
+)
+{
+       int     i ;
+       for (i = 0 ; i < len ; i++) d2[i] = d1[i];
+}
+
+
 extern void InvalidExecHandler();

 // Compare code segments by name
@@ -1790,6 +2154,183 @@
      return r;
      }

+
+/** Code White - Checks to see if loading code is trusted.
+
+     Capt Joe Hinson, Air Force Institute of Technology, July 2010
+
+     Calculates HMAC of loading code section and searches for the HMAC
in a file
+     of trusted HMACs (hmacs.txt).  The HMAC of the hmacs.txt file is
also
+     calculated and checked against its known HMAC.  Based on the
outcome of the
+     two comparisons, the global flag 'AllowExecution' is set to
signal whether
+     the process will be allowed to execute (done during
E32Loader::ProcessLoaded).
+
+     @param      aInfo       Information about the loading process.
+     @param      aCodeInfo   Information about the buffers containing
the code section
+                                   and HMAC list that are to be
checked.
```

```
+                                   aCodeInfo.iCLA - address of user
heap buffer containing code
+                                   aCodeInfo.iSize - size of loading
code section
+                                   aCodeInfo.iHmacList - addr of user
heap buffer containing list
+                                   aCodeInfo.iHLSize - size of hmac
list
+
+ */
+TInt ExecHandler::CodeCheck(TProcessCreateInfo& aInfo, TCodeInfo*
aCodeInfo)
+     {
+     K::CheckFileServerAccess();               // only F32 can use
this exec function
+     TInt IllegalHmacList=0;
+     TInt HmacIsAMatch=0;
+     TProcessCreateInfo info;
+     kumemget32(&info, &aInfo, sizeof(info));
+     TCodeInfo CInfo;
+     kumemget32(&CInfo, aCodeInfo, sizeof(CInfo));
+
+     Kern::Printf("%S", &info.iFileName);
+
+     //Create Heap buffer for code section
+     NKern::ThreadEnterCS();
+     TUint8* HashBlock=new TUint8[CInfo.iSize];
+     NKern::ThreadLeaveCS();
+     if (!HashBlock)
+           return KErrNoMemory;
+     //Use kumemget to move code from user heap to kernel heap
+     kumemget(HashBlock, CInfo.iCLA, CInfo.iSize);
+
+
+
+     //Create Heap buffer for hmacs list
+     NKern::ThreadEnterCS();
+     TUint8* HmacsList=new TUint8[CInfo.iHLSize];
+     NKern::ThreadLeaveCS();
+     if (!HashBlock)
+           return KErrNoMemory;
+     //Use kumemget to move code from user heap to kernel heap
+     kumemget(HmacsList, CInfo.iHmacList, CInfo.iHLSize);
+
+     //Print proof of successful copy if needed
+     //TPtr8 HLptr(HashBlock, 20, 20);
+     //TBuf8<20> printbuf;
+     //printbuf.Copy(HLptr);
+     //Kern::Printf("Buf is %S",&printbuf);
+     _LIT8(KHmacListHmac,
"\x07\x44\xD8\xC6\x32\x1B\x44\x34\x74\x65\x89\x66\x34\x51\xB7\x4E");//h
mac of hmacs.txt
+
+     TPtrC8 HLH(KHmacListHmac);
+
+     unsigned char g_key[] = "12345678901234567890123456789 0AB";
+     unsigned char codehmac[MD5_DIGESTSIZE];
+     unsigned char listhmac[MD5_DIGESTSIZE];
+     char* k = (char*)g_key;
+     int lk = 32;
+     int t = MD5_DIGESTSIZE;
+     char* d = (char*)HmacsList;  //pointer to list of HMACs
+     int ld = CInfo.iHLSize;       //size of the list
+     char* out = (char*)listhmac; //will hold the list's computed HMAC
+     TInt n;
+
+     //compute hmacs
+     for (n=0; n<2; n++) //compute hmac of hash list when n=0
+           {
+
+           if (n) //compute hmac of code section when n=1
+                 {
+                 d = (char*)HashBlock; //location of the code to be
hashed
+                 ld = CInfo.iSize; //size of the code
+                 out = (char*)codehmac; //will hold the code's
computed hmac
+                 }
+
+           MD5_CTX ictx, octx ;
+           unsigned char    imd5[MD5_DIGESTSIZE], omd5[MD5_DIGESTSIZE]
;
+           unsigned char    key[MD5_DIGESTSIZE] ;
+           unsigned char    buf[MD5_BLOCKSIZE] ;
+           int     i ;
+
+           if (lk > MD5_BLOCKSIZE) {
+
+                 MD5_CTX         tctx ;
+
+                 MD5Init(&tctx) ;
+                 MD5Update(&tctx, (unsigned char*)k, lk) ;
+                 MD5Final(key, &tctx) ;
+
+                 k = (char*)key ;
+                 lk = MD5_DIGESTSIZE ;
+                 }
+
+           /**** Inner Digest ****/
+
+           MD5Init(&ictx);
+
+           /* Pad the key for inner digest */
+           for (i = 0 ; i < lk ; ++i) buf[i] = k[i] ^ 0x36 ;
+           for (i = lk ; i < MD5_BLOCKSIZE ; ++i) buf[i] = 0x36 ;
+
+           MD5Update(&ictx, buf, MD5_BLOCKSIZE) ;
+           MD5Update(&ictx, (unsigned char*)d, ld) ;
+
+           MD5Final(imd5, &ictx) ;
+
+           /**** Outter Digest ****/
+
+           MD5Init(&octx) ;
+
+           /* Pad the key for outter digest */
+
+           for (i = 0 ; i < lk ; ++i) buf[i] = k[i] ^ 0x5C ;
+           for (i = lk ; i < MD5_BLOCKSIZE ; ++i) buf[i] = 0x5C ;
+
+           MD5Update(&octx, buf, MD5_BLOCKSIZE) ;
+           MD5Update(&octx, imd5, MD5_DIGESTSIZE) ;
+
+           MD5Final(omd5, &octx) ;
+
+           /* truncate and print the results */
+           t = t > MD5_DIGESTSIZE ? MD5_DIGESTSIZE : t ;
+           truncate(omd5, (unsigned char*)out, t) ;
+
+           if (!n)
+                 {
+                 TPtr8 hlhptr(listhmac,MD5_DIGESTSIZE,
MD5_DIGESTSIZE);
+                 IllegalHmacList=hlhptr.Compare(HLH);
+                 //Kern::Printf("IllegalHmacList=%d",
IllegalHmacList);
+                 }
+
+           }
+     TPtrC8 codehmacptr(codehmac, MD5_DIGESTSIZE); //pointer to hmac
of the code
+     TPtrC8 listptr(HmacsList, CInfo.iHLSize); //pointer to the list
+     TPtrC8 listhmacptr(listhmac, MD5_DIGESTSIZE);
+     TInt r=listptr.Find(codehmacptr);
+     if (r!=KErrNotFound)
+           HmacIsAMatch=1;
+
+     //print code hmac if needed
+     //TBuf8<MD5_DIGESTSIZE> hmacbuf;
+     //hmacbuf.Copy(codehmacptr);
+     //Kern::Printf("code hmac = %S",&hmacbuf);
+     //TBuf8<MD5_DIGESTSIZE> listbuf;
+     //listbuf.Copy(listhmacptr);
+     //Kern::Printf("list hmac = %S",&listbuf);
+
+     //clean up the heap
+     NKern::ThreadEnterCS();
+     delete[] HashBlock;
+     NKern::ThreadLeaveCS();
+
+     NKern::ThreadEnterCS();
+     delete[] HmacsList;
+     NKern::ThreadLeaveCS();
+
+     extern TUint AllowExecution;
+     extern TUint HasBeenSet;
+     Kern::Printf("IllegalHmacList=%d HmacIsAMatch=%d
AllowExecution=%d HasBeenSet=%d", IllegalHmacList, HmacIsAMatch,
AllowExecution, HasBeenSet);
+     if (IllegalHmacList!=0 || HmacIsAMatch!=1)
+           AllowExecution=0;
+     if (IllegalHmacList==0 && HmacIsAMatch==1 && HasBeenSet==0)
+           AllowExecution=1;
+     Kern::Printf("IllegalHmacList=%d HmacIsAMatch=%d
AllowExecution=%d HasBeenSet=%d", IllegalHmacList, HmacIsAMatch,
AllowExecution, HasBeenSet);
+     HasBeenSet=1;
+
+     return KErrNone;
+     }
+
 TInt ExecHandler::ProcessLoaded(TProcessCreateInfo& aInfo)
      {
      TProcessCreateInfo info;
diff -r 2ee5df201f60 kernel/eka/kernel/sprocess.cpp
--- a/kernel/eka/kernel/sprocess.cpp      Mon Mar 08 11:58:34 2010
+0000
+++ b/kernel/eka/kernel/sprocess.cpp      Thu Aug 26 11:46:14 2010 -
0400
@@ -159,6 +159,12 @@

      BTrace8(BTrace::EThreadPriority,BTrace::EProcessPriority,this,iPr
iority);
 #endif
      iId = K::NewId();
+
+     extern TUint AllowExecution; //Code White - Initialize
AllowExecution & HasBeenSet
+     extern TUint HasBeenSet;
+     AllowExecution=0;
+     HasBeenSet=0;
+
      iCreatorId = iId;  // Initialise as self for safety because
creator has special capabilities
      if(TheSuperPage().KernelConfigFlags() &
EKernelConfigPlatSecProcessIsolation)
            {
diff -r 2ee5df201f60 kernel/eka/memmodel/epoc/multiple/mprocess.cpp
--- a/kernel/eka/memmodel/epoc/multiple/mprocess.cpp  Mon Mar 08
11:58:34 2010 +0000
+++ b/kernel/eka/memmodel/epoc/multiple/mprocess.cpp  Thu Aug 26
11:46:14 2010 -0400
@@ -20,6 +20,9 @@
 #include "cache_maintenance.h"
 #include <demand_paging.h>

+TUint AllowExecution; //Code White global flags
+TUint HasBeenSet;
+
 #define iMState        iWaitLink.iSpare1
 // just for convenience...
@@ -533,7 +536,17 @@
                        LOCK_USER_MEMORY();
                        }
                  }
-           if(r!=KErrNone)
+           if(!aLoading)  //Code White - If in ProcessLoaded and not
trusted, set permissions to same as user data (includes XN bit)
+                 {
+                 if (!AllowExecution)
+                       {
+                       TPte perm=0x83f;
+                       iCodeChunk->ApplyPermissions(offset, codeSize,
perm);
+                       Kern::Printf("Execution disallowed for iId
%08x!", iId);
+                       }
+                 }
+
+                 if(r!=KErrNone)
                  {
                  // error, so decommit up code pages we had already
committed...
                  DChunk::TDecommitType decommitType = paged ?
DChunk::EDecommitVirtual : DChunk::EDecommitNormal;
diff -r 2ee5df201f60 userlibandfileserver/fileserver/sfile/sf_lepoc.cpp
--- a/userlibandfileserver/fileserver/sfile/sf_lepoc.cpp    Mon Mar 08
11:58:34 2010 +0000
+++ b/userlibandfileserver/fileserver/sfile/sf_lepoc.cpp    Thu Aug 26
11:46:14 2010 -0400
@@ -33,6 +33,7 @@
 #include <e32uid.h>
 #include <e32rom.h>
 #include "sf_cache.h"
+#include <hal.h>

 #include "sf_pgcompr.h"

@@ -1540,10 +1541,10 @@
      iS = aFinder.iNew.iS;

      // check if executable has already been loaded...
-     r = CheckAlreadyLoaded();
-     if(r!=KErrNone)
-           return r;
-
+     //r = CheckAlreadyLoaded();
+     //if(r!=KErrNone)
+     //    return r;
+     iAlreadyLoaded=0; //Code White - force all applications to load
from file - even if loading multiple instances
      // if we are going to need to load it...
      if(!iAlreadyLoaded || !iIsDll)
            {
```

```
@@ -1632,9 +1633,11 @@
            __IF_DEBUG(Printf("%S is not marked SMP safe",
&iFileName));
            iAttr &= ~ECodeSegAttSMPSafe;
            }
-
+     if (iHeader->iCompressionType!=KFormatNotCompressed) //Code White
- only support uncompressed files
+           return KErrNotSupported;
      // check if executable is to be demand paged...
-     r = ShouldBeCodePaged(iUseCodePaging);
+     //r = ShouldBeCodePaged(iUseCodePaging);
+     iUseCodePaging=EFalse;
      __IF_DEBUG(Printf("ShouldBeCodePaged r=%d,iUseCodePaging=%d", r,
iUseCodePaging));
      if(iUseCodePaging==EFalse || r!=KErrNone)
            return r;
@@ -2076,6 +2079,59 @@
            TInt r = iFile.Read(aPos,p,aSize);
            if(r!=KErrNone)
                  return r;
+
+           //Code White - Read code section into heap buffer
+           //Create buffer and descripter to point to it
+           if (aPos==0x9C)
+                 {
+                 TUint8* codebuf=new TUint8[aSize];
+                 TPtr8 codebufptr((TUint8*)codebuf,aSize,aSize);
+
+                 //Read code section
+                 iFile.Read(aPos, codebufptr, aSize);
+
+                 //Print proof if needed
+                 //TPtr8 bufprnt(codebuf,20, 20);
+                 //RDebug::Printf("buf:%S",&bufprnt);
+
+
+                 //Read Hmacs file into heap buffer pointed to by
codeInfo.iHmacList
+                 RFs hfs;
+                 hfs.Connect();
+                 _LIT(KHmacsFile,"s:\\sys\\bin\\hmacs.txt");
+                 RFile HmacsFile;
+                 //open the hmac file
+                 TInt h =
HmacsFile.Open(hfs,KHmacsFile,EFileShareExclusive|EFileRead);
+                 if (h!=KErrNone)
+                       {
+                       RDebug::Printf("Couldn't open hmacs.txt - %d",
h);
+                       return h;
+                       }
+                 TInt fsize;
+                 HmacsFile.Size(fsize);
+                 TUint8* hmacslist=new TUint8[fsize];
+                 TPtr8 hmacsptr((TUint8*)hmacslist,fsize,fsize);
+                 HmacsFile.Read(0, hmacsptr, fsize);
+                 HmacsFile.Close();
+                 hfs.Close();
+
+                 //Prep codeInfo to pass to CodeCheck
+                 TCodeInfo codeInfo;
+                 codeInfo.iSize=iCodeSize;
+                 codeInfo.iCLA=codebuf;//CodeForHmac;
+                 codeInfo.iHmacList=hmacslist;
+                 codeInfo.iHLSize=fsize;
+                 TCodeInfo* CIPtr = &codeInfo;
+
+                 //Call CodeCheck and pass the buck to kernel mode
+                 TInt t=E32Loader::CodeCheck(*this, CIPtr);
+                 if (t!=KErrNone)
+                       return r;
+
+                 delete[] codebuf;
+                 delete[] hmacslist;
+                 }
+                 //end
            }

      // check we got the amount of data requested...
```

Bibliography

[Glo09] (2009, Sep.) Synovate. [Online]. http://www.synovate.com/news/article/2009/09/global-mobile-phone-survey-shows-the-mobile-is-a-remote-control-for-life.html

[Rob03] P. F. Roberts. (2003, Feb.) Infoworld. [Online]. http://www.infoworld.com/d/networking/nokia-phones-vulnerable-dos-attack-068

[Gos09] A. Gostev. (2009, Sep.) Securelist. [Online]. http://www.securelist.com/en/analysis/204792080/Mobile_Malware_Evolution_An_Overview_Part_3

[Guo07] C. Guo, H. J. Wang, and W. Zhu, "Smart-Phone Attacks and Defenses," 2007.

[Che07] J. Cheng, S. H. Wong, H. Yang, and S. Lu, "SmartSiren: virus detection and alert for smartphones," *Proceedings of the 5th International Conference On Mobile Systems, Applications and Services*, pp. 258-271, 2007.

[Ley05] J. Leyden. (2005, Aug.) SecurityFocus. [Online]. http://www.securityfocus.com/news/11279

[Wor09] F-Secure. [Online]. http://www.f-secure.com/v-descs/mabir.shtml

[Sha08] J. Shah. (2008, Feb.) McAfee Labs Blog. [Online]. http://www.avertlabs.com/research/blog/index.php/2008/02/26/windows-mobile-trojan-sends-unauthorized-information-and-leaves-device-vulnerable/

[Bie05] C. Biever. (2005, Mar.) NewScientist. [Online]. http://www.newscientist.com/article/dn7080-phone-viruses-how-bad-is-it.html

[Dag04] D. Dagon, T. Martin, and T. Starner, "Mobile Phones as Computing Devices: The Viruses are Coming!," *IEEE Pervasive Computing*, pp. 11-15, 2004.

[Tro08] (2008, Mar.) F-Secure. [Online]. http://www.f-secure.com/v-descs/trojan_symbos_multidropper.shtml

[Sym09] (2009, Aug.) UMU Mobile Security. [Online].

http://www.umuglobal.com/encyclopaedia/113

[Fer08] P. Ferrie. (2008, Dec.) Microsoft Malware Protection Center. [Online]. http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Worm%3aWinCE%2fPmcryptic.A

[Mie06] M. Miettinen and P. Halonen, "Host-Based Intrusion Detection for Advanced Mobile Devices," *Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 02*, pp. 72-76, 2006.

[Kro09] K. L. Kroeker, "The Evolution of Virtualization," *Communications of the ACM*, pp. 18-20, 2009.

[Jar07] Jarno. (2007, May) F-Secure. [Online]. http://www.f-secure.com/weblog/archives/00001190.html

[Apv10] A. Apvrille. (2010, Jul.) Fortinet. [Online]. http://blog.fortinet.com/symbian-signed-mobile-malware-one-gang/

[Enc09] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security* , pp. 235-245, 2009.

[Sig10] Android Developers. [Online]. http://developer.android.com/guide/publishing/app-signing.html

[Sch09] J. Schiffman. (2009) Pennsylvania State University Department of Computer Science and Engineering. [Online]. http://www.cse.psu.edu/~enck/cse597a-s09/slides/security_blackberry.pdf

[The10] iOS Reference Library. [Online]. http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/RuntimeEnvironment/RuntimeEnvironment.html#//apple_ref/doc/uid/TP40007072-CH2-SW3

[Sch091] A.-D. Schmidt, F. Peters, and e. al., "Monitoring Smartphones for Anomaly Detection," *Mobile network Applications*, pp. 92-106, 2009.

[Xie09] L. Xie, X. Zhang, and e. al., "Designing System-Level Defenses Against Cellphone Malware," *Proceedings of 28th IEEE International Symposium on Reliable Distributed Systems* , 2009.

[Kim09] W. B. Kimball, *SecureQEMU: Emulation-Based Software Protection Providing Encrypted Code Execution and Page Granularity Code Signing*. MS thesis, AFIT/GCO/ENG/09-03. School of Engineering and Management, Air Force Institute of Technology (AU), Wright Patterson AFB OH, March 2009., 2009.

[Kra06] T. Krazit. (2006, Apr.) CNet News. [Online]. http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html

[Kna04] P. Knaggs and S. Welsh, *ARM: Assembly Language Programming*. Bournemouth: Bournemouth University Press, 2004.

[ARM05] *ARM Architecture Reference Manual*. Cambridge, England: ARM Limited, 2005.

[Sal06] J. Sales, *Symbian OS Internals: Real-Time Kernel Programming*. John Wiley & Sons, Ltd, 2006.

[Sal75] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE, Vol. 63, No. 9*, 1975.

[Cra10] D. Crane. (2010, Aug.) Defense Review. [Online]. http://www.defensereview.com/general-dynamics-itronixs-new-android-based-gd300-rugged-wearable-computersecure-radio-enables-unprecedented-gps-and-situational-awareness-in-the-battlespace-for-infantry-warfighters/

[Nie94] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1994.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)* 02-09-2010	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From – To)* Aug 2009 - Sept 2010

4. TITLE AND SUBTITLE

Code White: A Signed Code Protection Mechanism for Smartphones

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Joseph M. Hinson, IV

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management
2950 Hobson Way
WPAFB, OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCO/ENG/10-10

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
INTENTIONALLY LEFT BLANK

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research develops Code White, a hardware-implemented trusted execution mechanism for the Symbian mobile operating system. Code White combines a signed whitelist approach with the "execute never" bit in the ARM architecture. It prevents all untrusted code from executing while minimally impacting the user experience, as shown via validation and performance testing. Smartphones have proven to be invaluable to military, civic, and business users due in a large part to their ability to execute code just like any desktop computer can. While many useful applications have been developed for these users, numerous malicious programs have also surfaced. And while smartphones have desktop-like capabilities to execute software, they do not have the same resources to scan for malware. More efficient means which minimize resource usage are needed.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Rusty Baldwin Rusty.Baldwin@afit.edu
REPORT U	**ABSTRACT** U	**c. THIS PAGE** U	UU	84	**19b. TELEPHONE NUMBER** *(Include area code)* 937-785-3636 ext 4445