RUHR UNIVERSITÄT BOCHUM RUB

hgi Lehrstuhl für Systemsicherheit

Analysis and Retrofitting of Security Properties for Proprietary Software Systems

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

vorgelegt von

Ralf Hund

aus Heidelberg

10. Juni 2013

Erstgutachter:	Prof. Dr. Thorsten Holz (Ruhr-Universität Bochum)
Zweitgutachter:	Prof. Dr. Felix C. Freiling (Friedrich-Alexander-Universität Erlangen-Nürnberg)

Abstract

In this thesis, we present new methods to analyze and extend the security properties of software systems. We concentrate on *proprietary* software environments in which no source code of the relevant components is available. Analyzing and retrofitting security aspects of proprietary software is an especially intricate task and requires the development and use of customized techniques and tools.

The contributions of the thesis are thus twofold. First, we develop tools and revise existing techniques to analyze proprietary systems. To that end, static reverse engineering is used to extract algorithms from proprietary software. Second, we also provide solutions to overcome security-related shortcomings in proprietary systems by extending them, i.e., we *retrofit* proprietary systems. Overall, we present an analysis and/or retrofitting of four different software systems in this thesis.

At first, we describe the reverse engineering of the proprietary GMR-1 satellite phone communication encryption algorithm. We present a generic approach to identify and extract unknown cryptographic algorithms in mobile firmware images and provide an in-depth description of such an analysis using the example of one concrete GMR-1 firmware. In the end, we were able to successfully extract the unknown GMR-1 encryption algorithm which later proved to be vulnerable to cryptographic attacks.

Another contribution is the development of new side channel timing attacks against kernelspace ASLR implementations. We therefore reverse engineered the proprietary ASLR implementation of Windows operating systems and present three different side channel attacks which allow a local attacker to reconstruct large parts of the privileged kernel space by abusing side channels in the processor's memory management facilities. This allows us to effectively bypass kernelspace ASLR protections. We also present a mitigation solution with negligible overhead that renders the attack infeasible.

The third topic of this thesis is the design and implementation of the dynamic runtime components of the control flow integrity (CFI) framework MoCFI. MoCFI protects binary iOS applications against attackers that exploit software vulnerabilities to execute arbitrary code. We therefore developed techniques to allow for protecting arbitrary binary program images on iOS. Our evaluation shows that MoCFI is capable of protecting various popular applications with reasonable overhead.

Finally, we present a new approach to detect malicious command and control (C&C) bot connections. By enriching network-level information with host-level data, we create so-called behavior graphs that connect system activity with network packet data. This is achieved by monitoring the proprietary native API of Windows. Our evaluation shows that behavior graphs can be used to accurately tell apart C&C connections from legitimate benign traffic.

Zusammenfassung

Die vorliegende Dissertation befasst sich mit der Entwicklung neuer Methoden zur Analyse und nachträglichen Erweiterung von Sicherheitseigenschaften von Softwaresystemen. Die Arbeit konzentriert sich dabei auf *proprietäre* Softwareumgebungen in denen kein Quellcode für die relevanten Softwarekomponenten verfügbar ist. Sowohl die Analyse als auch Erweiterung proprietärer Software erfordert die Entwicklung und den Einsatz speziell angepasster und neuartiger Techniken und Werkzeuge. Die Dissertation unterteilt sich in vier Themenbereiche, in denen jeweils unabhängig voneinander die Analyse und/oder Erweiterung bestimmter Softwaresysteme präsentiert wird.

Der erste Themenkomplex umfasst das Reverse Engineering der unbekannten Verschlüsselung des proprietären Satellitentelephonie-Standards GMR-1. Es wird ein generischer Ansatz zur Identifizierung und Extraktion unbekannter kryptographischer Algorithmen in mobilen Firmware-Images vorgestellt. Anhand der umfassenden Analyse einer konkreten Firmware wird der GMR-A5-1 Verschlüsselungsalgorithmus rekonstruiert. Dieser weist große Ähnlichkeiten zur bekanntermaßen angreifbaren GSM-A5/2 Verschlüsselung auf.

Ein weiterer Beitrag ist die Entwicklung neuartiger timingbasierter Seitenkanalangriffe auf Kernelspace ASLR Implementierungen. Hierfür wurde die proprietäre Implementierung von Windows Betriebssystemen rekonstruiert und es wurden darauf aufbauend drei verschiedene timingbasierte Angriffe entwickelt. Diese erlauben es einem lokalen Angreifer, große Teile des privilegierten Kernelspace-Adressraums zu rekonstruieren. Dadurch können ASLR Sicherheitsmechanismen vollständig umgangen werden. Es wird außerdem eine Betriebssystemerweiterung präsentiert, welche die diskutierten Angriffe effektiv verhindert.

Als drittes Thema befasst sich die Arbeit mit dem Entwurf und der Implementierung der Laufzeitkomponenten des Kontrollflussintegrität-Frameworks MoCFI. MoCFI schützt beliebige binäre iOS Applikationen gegen Angreifer indem die Ausnutzung von Softwareschwachstellen verhindert wird. Zu diesem Zweck wurden neue Techniken entwickelt um binäre Anwendungen zur Laufzeit effizient und fehlerfrei zu überwachen. Die Evaluation des Frameworks zeigt, dass MoCFI in der Lage ist diverse weitverbreitete iOS Applikationen mit akzeptablen Geschwindigkeitseinbußen zu schützen.

Im letzten Teil der Arbeit wird ein neuer Ansatz zur Identifizierung von bösartigen Command and Control (C&C) Bot-Netzwerkverbindungen vorgestellt. Durch die Kombination von Netzwerk- mit Host-basierten Informationen werden Verhaltensgraphen erzeugt. Diese verbinden die Systemaktivität eines Bots mit generierten Netzwerkpaketen indem die proprietäre, native Windows-API überwacht wird. In der Evaluation kann gezeigt werden, dass dieser neuartige Ansatz eine effektive Unterscheidung zwischen bösartigen C&C und gutartigen Verbindungen ermöglicht.

Contents

1

Introduction

1.1 Motivation

The rapid development of the computer industry has fundamentally changed the way we communicate with each other. Groundbreaking advancements in the field of hardware manufacturing lead to lowered costs of computer devices which in turn opened the digital world to large parts of the global population. The concurrent breakthrough of the Internet was the second pivotal element that contributed to the global interconnection of almost all electronic devices nowadays. As a natural consequence, this digital revolution has also led to a shift of traditional fields into the digital world, in both the private and business domain. For example, our social life increasingly takes place on social media sites such as Facebook, shopping needs are satisfied by a plethora of web shops and financial transactions from private individuals up to multi-billion companies are processed digitally.

While this technologic progress clearly facilitates our everyday lives, it also comes with certain downsides. Namely, it also raises new kinds of security and privacy issues that have to be addressed accordingly. There exists a growing incentive for attackers and criminals to move their malicious and criminal activities towards digital systems. This is mainly caused by the large amount of sensitive and possibly valuable data that are being stored and processed digitally. Sensitive data can range from private authentication data (e.g., for online banking sites or web shops) up to corporate secrets that can cause immense damage if they land in the wrong hands. Recent incidents such as the infamous Stuxnet malware proved that digital warfare has even become standard practice for many secret agencies operating worldwide [143].

Apart from such large scale operations that are backed by immense financial budgets, we can also observe that digital attacks in general are getting increasingly sophisticated on the technical side, which emphasizes the importance of new comprehensive and innovative security measures. Not only have attacks become more

advanced, recent trends such as the emergence of mobile devices have opened new attack vectors. By definition, a mobile device mainly communicates with remote services over wireless transmissions. Due to the broadcast nature of wireless connections, attackers might easily eavesdrop on potentially sensitive conversations or data transmissions — even in case they are far away from their victim. Mobile communication systems thus must provide tap-proof wireless connections. Furthermore, devices typically connect to a variety of different local networks depending on their current location. For example, users might use their mobiles to log into several wireless hot spots at airport or train stations, connect to the home or corporate network, and log into several different operator GSM networks without even noticing. Each of these networks may have different security requirements and measures in place, which opens many new attack surfaces for malicious adversaries.

Another important area is the development and implementation of *generic* and efficient measures to mitigate the exploitation of user application vulnerabilities. As operating system vendors implemented such generic anti-exploitation techniques, attackers concurrently refined their methods to bypass them. This has lead to the consecutive development of techniques such as stack cookies [68], W $\oplus$ X [116] and address space layout randomization [30]. Modern operating systems thus provide multiple generic defense techniques that complement each other and significantly raise the hurdle for an attacker to mount successful attacks against software vulnerabilities. However, the past has shown that inadequate assumptions and shortcomings in the concrete implementations may quickly break the entire approach. It is thus necessary to evaluate these implementations against new innovative kind of attacks.

Moreover, the rapid success of Internet software sales portals such as the Apple App Store or the Google Android Market has opened the software market to a wider variety of developers. While such portals mostly exist for mobile platforms at the time of writing, both Microsoft and Apple announced that they plan on porting this concept to their desktop operating systems in the near future. Even non-commercial developers can easily create new applications and quickly make them available to a large user group. This also raises security concerns as it is unclear as to which extent developers follow secure coding practices.

Eventually, the analysis and evaluation of software security schemes is thus an important topic in an effort to assure the resilience of computer systems against malicious attackers. However, it is non-trivial to achieve this goal since the majority of computer platforms rely on *proprietary* software. When we speak of proprietary software in this thesis, we mean software whose source code is not being published and is thus only available as compiled binary packages. Proprietary software is prevalent especially among digital end user systems. Consequently, the two biggest end user operating system vendors Microsoft and Apple almost exclusively offer binary software and third party vendors that implement programs for these platforms

typically follow the same strategy.

The presence of proprietary software raises a diversity of security concerns. Firstly, it complicates the assessment and evaluation of security properties since oftentimes, it is unclear which security critical algorithms (e.g., encryption) are employed and even if they are known it is still cumbersome to ensure the absence of undermining implementation flaws. Many past incidents have proven that proprietary software vendors often rely on the *security by obscurity* principle which is known to cause many security issues. However, it is non-trivial to *analyze* proprietary software since many high-level abstractions are lost during the compilation process. Special approaches and techniques are therefore required.

A second problem that arises is how to *retrofit* proprietary software systems with improved security measures. End users cannot rely on vendors to distribute new or revised security techniques that overcome existing weaknesses all the time. Therefore, it is necessary to extend existing binary software packages. This however poses many significant obstacles that have to be overcome since it is impossible to simply modify existing high-level source code. New approaches and techniques are required to retrofit proprietary software.

1.2 Topic and Contributions

In this thesis, we analyze the security aspects of different software systems and also extend them to improve the overall security properties of the respective systems. Analyzing and retrofitting proprietary software is an especially intricate task since the source code of the analyzed programs is not available. This requires the development and use of customized techniques and tools. In a typical scenario, it is necessary to deal with binary code, which inherently operates on a low-level hardware assembler layer that impedes the abstract reconstruction of program semantics.

The contributions of the thesis are thus twofold. First, we develop tools and revise existing techniques to analyze proprietary systems. To that end, static reverse engineering is used to extract algorithms from proprietary software. Second, we also provide solutions to overcome security-related shortcomings in proprietary systems by extending them, i.e., we *retrofit* security to proprietary systems. Overall, we present an analysis and/or retrofitting of four different software systems in this thesis.

Security Analysis of a Proprietary Satphone System. We show how to extract the secret encryption algorithm employed in one of the two most widely used satellite telephony systems, GMR-1. Although the GMR-1 standard is publicly available from the ETSI standardization institute, the encryptions algorithms that are employed for wireless satphone to satellite communications are only presented as

a black box. There exist no concrete information on the structure and inner workings of the algorithm. However, the strength of the encryption plays an essential role to ensure privacy of users since data transmissions (especially the ones directed from the satellite to the satphone) are broadcasted to a large region surrounding the targeted receiver. Furthermore, satellite phones are often used in life-critical situations, e.g., for military purposes or by war correspondents. Our analysis targets the publicly available firmware image of a satellite phone that implements the GMR-1 standard. Due to the special hardware layout of the device and the fact that it incorporates two different processor architectures, we developed methods to identify relevant code regions inside the firmware. We present methods to reconstruct the virtual memory layout of the firmware, identifying the digital signal processor (DSP) image within the firmware, and show how the completely unknown encryption algorithms can be spotted within a large chunk of code instructions. To that end, we applied and advanced a variety of existing techniques for the purpose of identifying unknown encryption algorithms in binary code. We then successfully extracted the encryption/decryption algorithms of the GMR-1 standard. The results of this work were later used to mount practical and efficient crypto attacks against this algorithm, showing that the encryption of the GMR-1 system is insecure by today's standards.

Practical Timing Side Channel Attacks Against Kernel Space ASLR. The second contribution of this thesis is an analysis on the feasibility of timing side channel attacks against kernelspace Address Space Layout Randomization (ASLR) implementations. ASLR[30] is a modern security approach that aims at hindering the exploitation of software vulnerabilities. Therefore, the base loading addresses of program and library images in the address space of a running system process or the kernel are randomized. As a consequence, an attacker (who does not have access to the address space information previous to successful exploitation) cannot know the concrete memory addresses that she must access or execute in the last stage of the exploit. ASLR can be applied to both unprivileged (i.e. usermode) and privileged (i.e. kernelmode) domains. In this thesis, we analyze the feasibility of timing side channel attacks against privileged kernelmode ASLR protections. We therefore focus on the proprietary implementations of Microsoft Windows operating systems starting from Windows Vista. At first, we reverse engineered the ASLR algorithm in the operating system to figure out which components are randomized and as to which extend this happens, i.e. how high the randomization entropy is. We then developed three different attacks that allow a local attacker on the system to reconstruct at least parts of the concealed kernelspace. In any case, the sum of code and data with known base addresses is big enough to allow for mounting arbitrary, turing-complete return-oriented programming (ROP) attacks [133]. We thus suc-

cessfully break the kernelspace ASLR implementations of Windows. Furthermore, we also show how to reverse engineer the proprietary cache hash function of Intel Sandy Bridge processors, the latest hardware architecture of Intel CPUs at the time of writing. As explained later on, this was a necessary requirement to mount one of the three side channel attacks. In the final step, we discuss mitigations to the attack and provide a concrete software-based solution in the operating system that renders our timing attacks infeasible.

Providing Control Flow Integrity for Proprietary Mobile Devices. Another contribution of this thesis is the dynamic component of the CFI [2] framework Mobile CFI (MoCFI). MoCFI is a security retrofitting tool for the Apple iOS operating system. iOS is employed on all mobile Apple devices such as iPhones, iPads, and iPods. MoCFI introduces CFI to iOS applications and works solely on the binary level, no source code access to the protected applications is needed. Providing CFI is, similarly to ASLR, a preventative approach to hindering the exploitation of software vulnerabilities within application code. To that end, the code (in our case in binary form) of the application is statically analyzed and the valid control flow graph (CFG) is extracted. The CFG specifies allowed transitions between code blocks in the application code. Whenever a software exploit against a vulnerability is executed, this constitutes a violation of the allowed CFG execution transitions because the program will eventually execute shellcode. This allows to even detect sophisticated ROP attacks. In order to enforce the CFI rules at runtime, a dynamic rewriting library is needed that redirects all control flow transition of the application code and verifies that each respective control flow transition is valid. In this thesis, we provide the design and implementation of a dynamic CFI enforcement framework for the proprietary and restrictive Apple iOS operating system. We describe the obstacles that one faces in doing so and explain how these issues are addressed by MoCFI. Furthermore, we also extended the framework to not only provide protection against software exploits, but also introduce the possibility to apply fine-grained policies for individual iOS applications. By default, iOS ships with an application sandbox, but only assigns a *generic* sandboxing profile to all third-party applications. This profile grants every application access to multiple privacy-sensitive data sources (e.g., the address book, GPS coordinates, etc.). By extending MoCFI in a new tool called PSiOS, we give the end user the opportunity to assign individual profiles to each installed third-party application.

Behavior-Graph-Based C&C Detection for Proprietary Operating Systems. Lastly, we introduce a new approach to detect malicious bot network connections. Bots are a special type of malware and constitute a significant threat. Their distinctive feature is the establishment of a command and control (C&C)

server connection that allows them to be controlled remotely. Bots can then be used, e.g., for denial of service (DOS) attacks or to spy on the infected computer system. Researchers have proposed various either network- or host-based approaches to mitigate bots by detecting C&C network connections. However, these approaches oftentimes suffer from high false positive rates since they cannot tell apart C&C from benign connections, which are also frequently established by bots. We present a new model to solve this problem in the form of behavior graphs. Behavior graphs combine network and host information to allow for an improved modeling of C&C connections. In this thesis, we explain the behavior graph generation in the bot detection framework Jackstraws. The framework eventually produces behavior-graph-based C&C templates that allow matching malicious connections effectively with only few false positives. Behavior graph generation is provided for the proprietary Windows platform since this is the prevalent targeted platform for malware.

1.3 Outline

The thesis is structured in four different chapters. Each chapter covers one main topic of the thesis and is a self-contained unit that can be read independently from the other chapters.

Chapter 2 describes the reconstruction of the GMR-1 encryption algorithms from a proprietary satphone firmware image. The chapter first gives an introduction into satellite telephony and reverse engineering in general. We then present existing related work on identifying cryptograph primitives in binary code. The chapter also explains a general approach in reverse engineering mobile device firmware. To be precise, we present techniques to reconstruct the virtual memory mapping of the firmware image, identify and understand the DSP initialization, extract the DSP firmware and finally finding the searched-for (and unknown) encryption/decryption algorithms.

Chapter 3 presents three different timing side channel attacks against kernelspace ASLR facilities, with a special focus on the proprietary Windows operating system implementation. The chapter first explains the necessary technical background and related work in the area of hardware caches and side channel timing attacks. We then proceed to present our proposed attacks that circumvent the ASLR protection scheme of the kernel. We also provide detailed insights into implementation aspects, conduct an evaluation of the attacks on different hardware configurations, and present a mitigation solution that nullifies the proposed attack.

Chapter 4 introduces the dynamic components of the MoCFI control flow integrity framework for proprietary Apple iOS based devices. The chapter explains various implementation details and shows an evaluation using multiple popular iOS applications. We also present an extension of the framework called PSiOS that

allows for individual fine-grained application policies.

Chapter 5 presents our behavior graph generation that is embedded into the JACKSTRAWS system. We discuss related work on network- and host-based bot detection, explain the system from an abstract view, and then give many details on the behavior graph generation. We also provide an evaluation of the system to demonstrate its effectiveness.

1.4 Publications

The work presented in this thesis has been published at several academic conferences. This section gives an overview on the publications that are related to the contents of this thesis, as well as any additional academic publications that emerged in the course of the PhD studies.

Chapter 2 was published together with Driessen, Willems, Paar, and Holz [52] in our paper about the analysis of two satphone standards. The publication received the best paper award at the 33th IEEE Symposium on Security & Privacy.

The results of Chapter 3 are based on joint work with Willems and Holz [82] that presents our attacks against kernelspace ASLR. The paper is was published at the 34th IEEE Symposium on Security & Privacy.

The findings of Chapter 4 were published together with Davi, Dmitrienko, Egele, Fischer, Holz, Nürnberger, and Sadeghi [50] in our publication about the MoCFI framework at the NDSS Symposium 2012. Furthermore, the PSiOS extension was developed in the course of the master thesis of Tim Werthmann and has been published as joint work with Werthmann, Davi, Sadeghi, and Holz [154] at ASIACCS 2013 and received the distinguished paper award.

Chapter 5 was published as part of a paper together with Jacob, Kruegel, and Holz [87] about the C&C connection detection system JACKSTRAWS at the USENIX Security Symposium 2011.

Several other publications emerged in the course of my studies, which are not part of this thesis though. Namely, the findings of my diploma thesis about a new form of software attacks in the form of return-oriented rootkits were published together with Holz and Freiling [81]. The results of the InMAS project, a long-term project in an effort to create an effective automated malware analysis system, were published together with Engelberth, Freiling, Gorecki, Göbel, Holz, Trinius, and Willems [58]. Therefore, I contributed in the field of automated unpacking of packed binaries. New hardware-based approaches to provide improved automated analysis of malicious programs by using processor features were published together with Willems, Fobian, Felsch, Holz, and Vasudevan [157]. Similarly, together with my colleague Carsten Willems and Thorsten Holz we developed a novel malware analysis system that leverages hypervisor-based CPU features [158].

2

Security Analysis of a Proprietary Satphone System

2.1 Introduction

The rapid technological progress of the semiconductor industry during the last decades has lead to groundbreaking advancements in the development of mobile electronic devices. As several complex and power-efficient computer chips can now be incorporated even on small hardware boards, this evolution opened the creation of new mobile communication systems that allow for communicating from even remote places of the world. The first widespread communication system that heavily benefited from this revolution were mobile phone systems, which are available worldwide nowadays. More recently, several providers emerged that offer satellite phones (abbr. *satphones*) in order to fill the gaps left behind by radio-based telephony.

In a satellite communication system, the satphone directly communicates with one or more satellites in the earth orbit. This allows to almost provide a gapless service coverage worldwide. Satphones have thus become an essential building block for reachability in remote places. This includes, e.g., oil rigs, ships on the high sea, expedition teams, planes, etc., and generally situations in which a reliable emergency system is required to communicate with the outer world in case of need. Satellite systems were originally developed and employed for military purposes. Since the 90s, several private companies emerged that opened the service to businesses and private persons.

Backbone providers and device manufacturer quickly realized that common standards are required to push the success of mobile communication systems so that providers do not go their own ways. This offers various advantages to the user: she can choose between several mobile device models and the competition between multiple vendors cuts the prices. Therefore, official standards were introduced by standardization institutes. The Global System for Mobile Communications (GSM)

standard has emerged as the de-facto standard of cellular mobile phone communication with more than four billion subscribers in 2011. It is thus the most widely deployed standard for cellular networks and almost every mobile phone adheres to the corresponding specifications. Although large parts of the GSM standard are published freely, several security-critical aspects are kept secret. Unfortunately, the European Telecommunications Standards Institute (ETSI), which is the creator of the GSM, chose to rely on security by obscurity in that regard.

One important security aspect of every mobile communication system is the use of encryption algorithms to encrypt voice and data traffic between the device and its remote receiver/sender station. This is especially important due to the broadcast nature of every mobile device. Transmission are not only sent directly to the receiver, but are distributed to a large area surrounding the user. This makes it very easy to spy on mobile communication even if the eavesdropper is not close to the victim. Moreover, mobile devices are oftentimes used in sensitive situations, such as for military use or by war reporters. Since the encryption algorithms are kept secret and devices almost always exclusively consist of proprietary software and hardware components, the question arises how the end user can trust the confidentiality of her conversations and data transmissions.

Briceno et al. [36] reverse engineered the GSM algorithms in 1999. Since then, several researchers proved that the employed algorithms are insufficient and can be easily attacked in order to achieve real-time decryption of transmitted data without any foreknowledge about the key material [31, 26, 33]. However, no similar studies have been conducted about satellite systems. Briceno also does not provide any insights into the internals of the reverse engineering process that was used to extract the algorithms. It thus remains unclear which concrete approaches and methods were employed and how they can be adopted for other systems.

Two satphone standards were developed analogously to the GSM cellular standards in the past:

- *Geostationary Earth Orbit (GEO) Mobile Radio Interface* (better known as GMR-1) is a family of ETSI standards that were derived from the terrestrial cellular standard GSM. In fact, the specifications adopt large parts of the existing GSM standard and only explicitly specify those parts that differ. This protocol family is can be considered as the de-facto standard and has undergone several revisions to support a broader range of services. GMR-1 also supports the transmission of packet-oriented data streams, similar to GPRS. The newest GMR-1 3G specifications support IP-based voice, data, and video transmissions.

- The *GMR-2* family is a concurrent ETSI standard developed in collaboration with Inmarsat. It deviates from the GMR-1 specifications in numerous ways;

most notably the network architecture is different. It is only used by Inmarsat at the time of writing.

Similarly to GSM, the specification documents of both GMR-1 and GMR-2 are publicly available. The employed encryption is however treated as a black box and no further information about the encryption algorithms is given. This information is only made available to licensees of the standard. Since both GMR-1 and GMR-2 borrow large parts of the specification from GSM — in fact the GMR standards in places only specify differences to the GSM and adopt the rest — and GSM has been proven to rely on weak encryption algorithms, the question arises whether GMR uses the same or similar encryption schemes that are also vulnerable.

We are convinced that encryption algorithms of popular communication systems should be publicly available so that researchers can evaluate their effectiveness against eavesdropping attacks. This significantly helps in finding weak spots within these algorithms and increases the trust end users can have in the privacy of their voice and data transmissions. We think that it is virtually impossible to hide these algorithms from potential adversaries such as criminals or secret agencies anyway since they have the means to obtain this information.

In order to analyze and evaluate the proprietary encryption algorithms used by the GMR standards, it is at first necessary to retrieve these encryption algorithms. Since they are only made available to GMR licensees, the only solution that remains is to extract the cipher code from firmwares of satphones. In this thesis, we analyzed the Thuraya SO-2510 phone that implements the GMR-1 standard. It was released in November 2006 and is one of the most popular handsets sold by Thuraya.

2.1.1 Contributions

We are the first to perform an empirical security analysis of the satphone standard GMR-1, focusing on the encryption algorithms implemented in the handsets. This includes reverse-engineering of firmware images to understand the inner working of the phones, developing tools to analyze the code, and incorporating prior work on binary analysis to efficiently identify cryptographic code.

We provide several solutions to overcome various challenges that emerge throughout the analysis and are oftentimes resulting from the special structure of satphone devices. At first, we present means to reconstruct the virtual memory mapping of the firmware image. One of the core differences of mobile communication devices in comparison to desktop systems is the use of different and multiple CPU architectures. Satphones typically contain an additional DSP co-processor that adopts specific computation tasks. We tackle the architectural differences and show how to tell apart DSP code and data from the regular ARM architecture. Finally, we present our approach to identifying unknown cryptographic entities in code.

The solutions we provide are partially specific to the concrete Thuraya firmware that we analyzed. Nevertheless, we think that the general approach can be ported to other satphone or mobile devices in general. In fact, we later successfully analyzed the firmware from another phone that implements GMR-2[1].

The firmware image was the only starting point for our analysis and we stress that we conducted our analysis only using static tools since we did not have a satphone at our disposal. We also did not have any foreknowledge about the searched encryption algorithms which makes it especially difficult to find the searched-for needle in the haystack. In the end, were able to successfully reconstruct the proprietary encryption algorithm of GMR-1. The algorithm showed striking resemblance to one of the weak algorithms employed in GSM and later proved to be vulnerable to similar kinds of attacks.

This chapter is based on a previous publication together with Driessen, Willems, Paar, and Holz [52].

2.1.2 Outline

The chapter is structured in the following way: in Section 2.2 we introduce the technical background relevant to our analysis. Section 2.3 discusses related work in the field of identification of cryptographic entities. The general approach that we employed to extract the encryption algorithm is presented in Section 2.4. Section 2.5 describes the individual steps of our analysis in-depth. In the end, Section 2.6 summarizes our analysis and discusses possible topics for future work.

2.2 Technical Background

In this section, we introduce the relevant technical background for our analysis. We present basic concepts of satellite telephone systems, differences in the hardware structure of satphones in comparison to traditional devices, present the two CPU architectures ARM and TMSC55X that are used in the analyzed firmware, and give a brief overview over the GSM-A5/2 encryption algorithm.

2.2.1 Satellite Telecommunication Systems

Satellite telecommunication systems significantly differ from traditional landline or cellular-based infrastructures from the end-user device perspective. Instead of communicating with an operator or nearby base station, satphones send and receive messages directly from an orbital satellite. The satellite serves as an arbiter and forwards incoming satphone transmissions to a fixed gateway station that is operated

[1] This analysis is not part of this thesis.

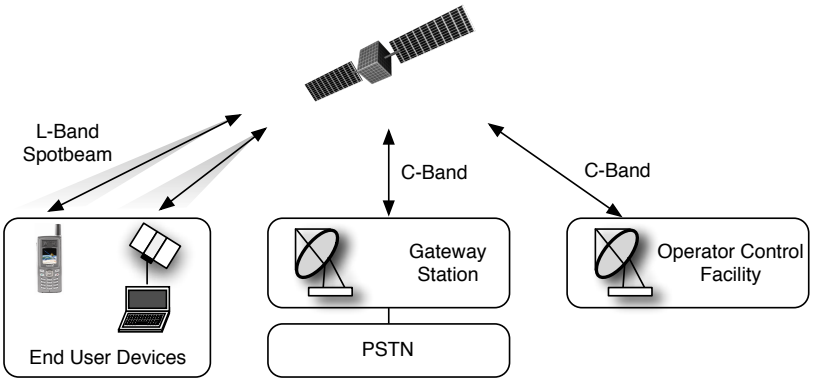


Figure 2.1: Layout of a geostationary orbit telephone network [88].

by the service provider. From there on, communication data takes its normal way into the public switched telephone network (PSTN). Multiple satellites may be in place, whereas each one serves a dedicated region. Figure 2.1 shows a schematic overview over a satellite telecommunication system. The satellite and end user devices communicate over a low frequency channel (the L-Band). The service area of each satellite is separated into several so-called *spotbeams*, which are circular regions that use same frequencies. This is mainly done to allow the reuse of frequency ranges for areas that are far afield from each other, since the service area covered by a single satellite can be huge.

Figure 2.2 shows an example of the spotbeam coverage of the Inmarsat system. The three big rectangles (light blue, green, and dark blue) constitute the service areas of three different satellites; the small circular areas are the spotbeams. Please note that any transmission from the satellite to the satphone is broadcasted to an area *at least* as large as the spotbeam they occur in. It is therefore easily possible to eavesdrop on transmitted data from a victim being located more than 1000 km away.

Apart from L-Band spotbeam connections, the satellite acts as a forwarder between the device and a ground-based gateway station on a C-Band connection. Please note that no further information is given on C-Band connections in the GMR standards. Thus it is unknown which protocols and algorithms are employed there and we did not focus on these connections in this thesis. Apart from the gateway station, the satellite operator also runs control facilities that allow maintaining and controlling the satellites.

As mentioned in the introduction, the GMR standards cover topics such as signaling, encodings, and protocols but only treat encryption as a black box. Figure 2.3

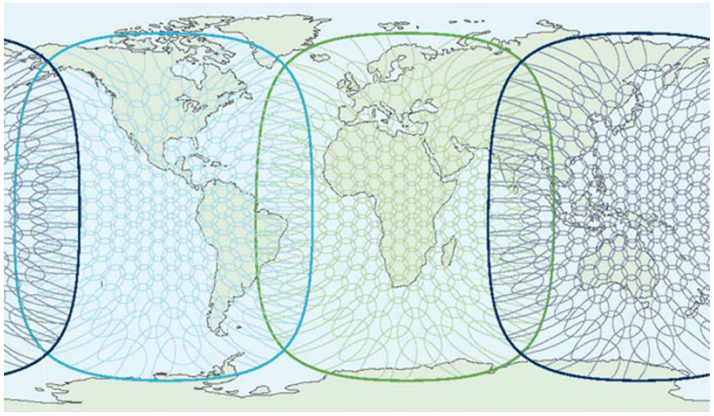


Figure 2.2: Inmarsat spotbeam coverage map [73].

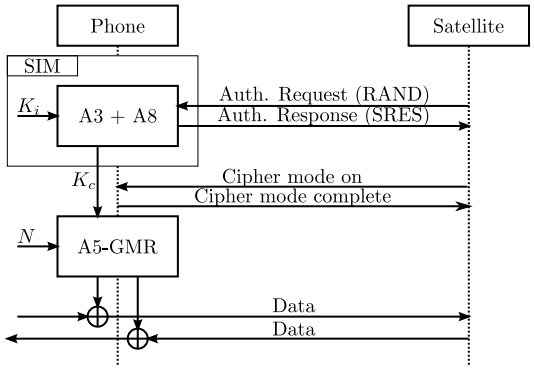


Figure 2.3: Schematic overview on the authentication and encryption facilities of GMR satellite-to-device (and vice versa) connections.

shows a schematic overview on the sparse information that is given in the documents. After a wireless physical connection between both devices has been established, the satellite proceeds to send a random number to the satphone. The phone's SIM card then calculates a session key and an authentication response using the A3 and A8 algorithms. From then on, each packet is encrypted using the A5-GMR algorithm. No information on the inner workings on any of the employed algorithms is provided. More information about GMR can be found in the literature [161, 105, 60, 104, 88].

2.2.2 Satellite Telephone Architecture

We now briefly elaborate on the general architectural structure of satphones and the hardware behind such devices. In Section 2.5, we provide more details on the specific phone we studied during our analysis.

In general, the architecture of satphones is similar to the architecture of cellular phones [153]. Both types of phones have to perform a lot of signal processing due to speech processing and wireless communication, thus they typically ship with a dedicated digital signal processor (DSP) for such purposes. Consequently, complex mathematical operations (like for example data compression or speech encoding) are outsourced to the DSP where the actual computations are performed. More relevant for our purpose is the fact that DSPs are also suitable for executing cryptographic algorithms efficiently, which makes DSP code a prime candidate for holding GMR cipher code.

The core of the phone is a standard microprocessor (usually an ARM-based CPU) that serves as the central control unit within the system. This CPU initializes the DSP during the boot process. Furthermore, both processors share at least parts of the main memory or other peripheral devices to implement inter-processor communication. To understand the flow of code and data on a phone, we thus also need to analyze the communication between the two processors.

The operating system running on a phone is typically a specialized embedded operating system that is designed with respects to the special requirements of a phone system (e.g., limited resources, reliability, real-time constraints, etc.). All of the software is deployed as one large, statically linked firmware binary. For our analysis, we were especially interested in the inter-processor communication functionality provided by the operating system as well as the DSP initialization routine. This is due to the fact that cipher code will likely be implemented in the DSP for performance reasons. Our interest for the DSP initialization routine arises from the fact that it typically reveals where DSP code is located in the firmware and how it is mapped to memory.

2.2.3 ARM Architecture

Mobile devices typically employ different processors than desktop or server devices. This mainly stems from differing requirements since power is a limited resource. Therefore, vendors must employ energy-saving components in their product. The prevalent desktop architecture x86 from Intel is a complex instruction set computer (CISC) architecture. During execution, every instruction first has to be translated into several micro instructions that are closer to the low level structure of the chip. This translation step requires a significant amount of additional transistors and in the end increases the power consumption. On the other hand, reduced instruction set computer (RISC) architectures follow a simple structure and do not require this additional translation step. This saves space on the silicon board and lowers the power consumption. Thus, most mobile devices (including satphones) employ RISC CPUs. ARM is the most prevalent CPU architecture for mobile devices and is also used by the GMR-1 device we analyzed. We present a brief summary of the

technical details of the architecture in the following.

ARM is a 32bit RISC architecture that provides the typical instruction set to allow for arithmetic and logic instructions, control flow instructions (branches and subroutine calls), and memory access instructions. It uses a fixed instruction length, which means that every instruction is 32bit long and instructions generally have to be aligned in memory, i.e., an instruction address must be divisible by four without any remainder. The CPU provides 16 general purpose register that are labeled R0-R15. However, the last three registers are typically allocated as the link register (LR), stack pointer (SP), and instruction pointer (IP). All the remaining registers can be used without any further imposed restrictions.

ARM is a load store architecture which means that memory accesses can only be implemented using dedicated load (`LDR`) and store (`STR`) instructions. All other instructions only use registers as operands. Common instructions may use up to three operand, one destination register and two source registers. The destination register is always the left-most register in the textual encoding. Thus, the instruction layout typically looks as follows:

```
INSTR rDest, rSource1, rSource2
```

For example, an addition that computes $r0 = r1 + r2$ is implemented as be `ADD r0, r1, r2`.

The official ARM specifications also impose the calling convention for subroutine calls. Therefore, the first four parameters of a subroutine are provided in the registers R0-R3, all remaining parameters are passed on the stack. The called subroutine can use R0-3 as scrap registers, the values of all other registers must be restored upon return and the return address of the called function is not stored on the stack but is instead provided in the LR register.

ARM also specifies the use of an memory management unit (MMU) for providing virtual memory. The MMU supports multi-level page table translations, a fine-grained permission system, and quick translation caching using translation lookaside buffers (TLBs). It is accessed using the co-processor instructions `MCR` and `MRC`.

2.2.4 TMS320C55X DSP Architecture

The TMS320C55X DSP architecture is part of the Texas Instruments TMS320 series. As with most other DSPs, the CPU is designed as a complementary chip for special purposes and typically does not act as the main CPU of the system. The chip is often used in devices such as mobile phones, audio players, and so on.

C55X is a 16bit architecture that is specialized on performing digital signal processing computations with a special focus on power efficiency. The DSP provides four general purpose registers, eight auxiliary registers, four accumulator registers, and various specials purpose registers (e.g., program status word, stack pointer,

etc.). The instruction set is large and flexible and provides various instructions for special arithmetic purposes. Instructions are encoded in variable length ranging from one up to six bytes. C55X is not a load store architecture which means that most instructions may directly access the memory.

2.2.5 GSM-A5/2

Similar to GMR-1, the GSM-A5/2 cipher was also not part of the official GSM specification. The algorithm was reverse engineered and analyzed in 1999 by Briceno et al. [36]. Figure 2.4 shows a structure of the algorithm. It consists of four linear feedback shift registers (LFSR) R1-R4 of a length between 19 and 23 bits. All four registers are clocked using a special clocking mechanism. The result of the clocking operation is a single output bit. According to Barkan et al. [26], the key stream is generated in the following four steps:

1. Initialization of the internal LFSR states using the key and frame number.
2. Force bits R1(15), R2(16), R3(18), and R4(10) to be one.
3. Execute 99 clocks and discard output
4. Execute 228 clocks to produce they key stream

The first three register are clocked under the control of R4. Clocking of R1-R3 happens depending on whether certain bits in R4 agree with a majority function. Three specific bits of R1-R3 (of which one is inverted) serve as the input to a majority function whose output is xored to form the final output bit. From the 228 key stream bits, the first half is used to encrypt base station to mobile device communication, while the latter half encrypts the communication from the mobile device to the base station. Since GSM-A5/2 is a stream cipher, the key stream bits are simply xored against the plain text to produce the cipher text.

An obvious conclusion that can be drawn from the inner workings of GSM-A5/2 is that the corresponding cipher code is bound to contain plenty of shift and xor operations. LFSR clocking constitutes the majority of the cipher computation and the registers have to be shifted and their output has to be xored.

2.3 Related Work

In this section, we discuss related work in the field of detection of cryptographic algorithms in binary code. The automatic identification of cryptographic algorithms without access to high-language source code has been documented by numerous researchers recently either as a whole or as a required stepping stone to enable further analysis of a binary program. Existing approaches stem from the fields of bot

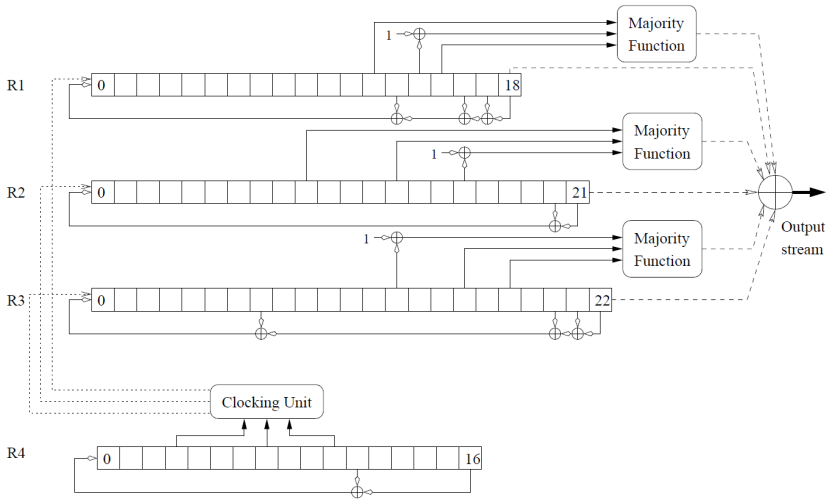


Figure 2.4: The GSM-A5/2 cipher [25].

analysis, e.g., to facilitate analysis of malware that employs unknown encryption algorithms for communication with a remote command and control (C&C) server, or automatic protocol reverse engineering which is bound to filter out possible encryption or decryption phases before the actual analysis can take place.

Lutz [113] automatically identifies decryption code to methodically obtain the plain text of encrypted malware network traffic. He therefore observed three specific attributes that speak for the presence of a decryption algorithm. Using a dynamic analysis on top of the Valgrind framework, the system is able to identify where and when plain text is present in memory by leveraging memory tainting and feature extraction. The author shows that his system is capable of analyzing an instance of the Kraken bot.

ReFormat is a tool developed by Wang et al. [151] that aims at analyzing encrypted messages for the purpose of automatic protocol reverse engineering. It is driven by the observation that there exist two different phases: message decryption and normal protocol processing. Transitions in execution traces from the former phase to the latter are detected by heuristics that leverage the fact that respective code from both phases is likely to execute a significant amount of different CPU instructions.

Dispatcher is a framework written by Caballero et al. [38] and also aims at enabling automatic protocol reverse engineering using automatic approaches to detect and circumvent encryptions of any type. The heuristics to detect cryptographic code are driven by the observation that encryption code differentiates in the type of used instructions. As a consequence, dispatcher is able to analyze the MegaD C&C

protocol and several open protocols.

Finally, Gröbert et al. [75] presented a set of different methods to identify cryptographic primitives in binary code using dynamic analysis. Their approaches not only have the ability to spot corresponding algorithms, but allow detecting and extracting crypto keys. The authors also improve the detection rates of existing system by refining and advancing existing methods and compare their approaches to previous work and show that they can achieve a higher detection rate and more accurate results.

The following paragraphs give an overview on the central approaches employed in the referenced work, along with the underlying observations and assumptions. Note that some approaches can be classified as static analysis, while others can only be performed in the context of a dynamic analysis. Since it is oftentimes possible to apply the same approach both statically and dynamically (e.g. by working on execution traces rather than disassemblies), we — in contrast to previous work — refrain from any classification attempts in these categories.

Signatures

Signatures take advantage of the fact that known cryptographic algorithms often contain a set of concrete and mostly unique instruction sequences that can be spotted using instruction encoding signatures. Furthermore, many algorithms contain unique constants or data structures that suggest the presence of a specific encryption or decryption. For example, AES and DES are bound to use static S-boxes that can be found in the data sections of a program. Gröbert used a more advanced approach by combining mnemonics and constants to tuples. These tuples are then pre-computed using open implementations of common crypto algorithms and matched against the sample program. As soon as the percentage of matched tuples for a given code piece exceeds a pre-defined threshold, this is considered as a match.

Density of Specific Instruction Types

Many cryptographic algorithms inherently require the computation of a large number of bitwise and arithmetic operations. This leads to an abnormal high density of such specific instructions inside encryption or decryption code. One can leverage this fact by rating all basic blocks or functions according to their amount of bitwise and arithmetic instructions.

It is also possible to loosen this premise by only assuming that crypto code generally contains very different instruction types in comparison to other algorithms. For example, ReFormat tries to detect the turning point from message decryption to normal protocol processing by observing an apparent change of executed instruction

types.

Variation of Information Entropy

One of the main consequences of a well-designed encryption algorithm is that produced cipher text has high information entropy. On the contrary, the plain text that is used as the input of the encryption function typically has considerably lower entropy since the data usually follows a pre-defined structure, such as text encodings or binary formats. If we, for the sake of convenience, assume that the input buffer of an encryption is equal to the output buffer, than the actual encryption leads to a considerable increase of the buffer's entropy. Likewise, decryption leads to a considerable decrease of information entropy.

It is possible to leverage this fact by identifying the input and output buffers of functions and monitoring changes in the information entropy in both respective data areas after a function was executed. For example, Lutz uses memory tainting and feature extraction for this purpose. Note that dynamic analysis is required for this approach in order to measure the entropy of concrete values.

Loop Detection

Implementations of cryptographic computations typically use several loop constructs in their code to compute recurring operations. The amount and size of loops depend on the processed data structure types (e.g. arrays, matrices, etc.) and the structure of the underlying algorithm. Occurrences of loops with distinct attributes in certain code pieces can thus be indicative of cryptographic computations. Since loops are a common construct and are used in the majority of not crypto-related code, this approach is usually rather used as a supplement in conjunction with the other described methods. Loop detection can also be hampered by the presence of loop unrolling compiler optimizations.

The approach can be performed both statically and dynamically, whereas the latter approach has the advantage that concrete numbers of loop iterations are known.

Verifiable Input and Output Relation

Encryptions and decryptions are deterministic in that they always generate the same output for identical input. If candidate functions are identified using approaches described above, then the dynamically observed input and output can be reproduced using a set of reference implementations of known cryptographic algorithms. In case the reproduced output matches the observed output, then the corresponding encryption or decryption algorithm was found.

The advantage of this approach is that it allows identifying concrete algorithms. On the downside, it cannot be used to identify previously unknown algorithms.

2.4 General Approach

In this section, we outline the general methodology we used for identifying and extracting encryption algorithms from satphones. Furthermore, we also discuss the assumptions that helped us during the analysis phase and provide an overview of our target satphone.

We analyzed a recent firmware of the Thuraya SO-2510 satphone that implements the GMR-1 specification. The starting point of our analysis was the publicly available firmware upgrade. The entire analysis was performed purely statically since we did not have a real satellite phone at our disposal that we could use to perform a dynamic analysis. Furthermore, we did not have access to a whole device simulator that enables debugging of the firmware image, thus we had to develop our own set of analysis tools. However, the ARM code for the main microprocessor can be partially executed and debugged in a CPU emulator such as QEMU.

The process of reverse engineering a concrete satphone is to some extent specific to that particular device as different phones implement different hard- and software components. Nevertheless, we try to abstract from these peculiarities as much as possible in this section.

2.4.1 Assumptions

The approach that we employed follows three underlying assumptions:

1. The satphone uses two different CPU architecture: one main CPU (typically ARM) and one DSP.

2. The searched-for encryption algorithms are implemented in the DSP code for efficiency reasons.

3. There is no obfuscation that requires the use of dynamic analysis techniques.

Please note that the first assumption does not restrict our approach to any significant degree since almost any mobile device uses the two-processor concept. Therefore, it is also reasonable to assume that the encryption is implemented in the DSP code since these are tailored to executing math-heavy operations. Our last assumption mainly stems from the absence of any concrete device during our analysis. Static analysis can be impeded by the presence of code obfuscation. However, we did not experience any such measures in the firmware and we expect that no other

vendor has taken such measures. If this were the case, the firmware image would have to be unpacked first.

Furthermore, we also presume three assumptions that facilitate the finding of the relevant pieces of cryptographic code:

1. The key length of the encryption algorithm is known.
2. The frame length is equal to the key length.
3. Since the GMR standards are derived from GSM, the ciphers bear at least some resemblance to the well-known, LFSR-based GSM-A5 algorithms.

The first two assumptions can be derived from the publicly available parts of the GMR specification [61]. The third assumption was conjectured by us. Note that the standard only specifies the general parameters of the crypto algorithms, but no details about the actual algorithm are publicly available. Nevertheless, these assumptions enabled us to decrease the search space of potential code. The last assumption is rather speculative, but also helped us in finding the algorithm.

2.4.2 Approach

The approach we followed to analyze the satphone can be separated into the following four steps:

1. *Firmware extraction*: extract the firmware image from the firmware installer.
2. *Virtual memory reconstruction*: reconstruct the correct memory mappings of the code and data sections in the firmware image.
3. *DSP initialization*: identify the DSP initialization procedure in order to extract the DSP code/mapping.
4. *Crypto code identification*: search for the encryption algorithms in the DSP code using specific heuristics.

The first step can vary depending on the type of installer that is employed. In our experience, the firmware image is oftentimes provided as a separate file in the installer package file. The second step is necessary to reconstruct the execution environment in the static analysis. Therefore, the correct mappings of the firmware in the device memory have to be known because otherwise the code cannot be disassembled correctly. The next step is to identify the DSP code and data within the firmware. A reasonable starting point is to search for the DSP initialization routine in the ARM code since the DSP is typically initialized at some point in the bootstrap process of the ARM. The DSP initialization also discloses the DSP code and data addresses. In the last step, one has to search for the unknown encryption algorithm with the help of certain heuristics and assumptions.

2.5 Reverse Engineering the Encryption Algorithms

We used the Thuraya SO-2510 phone in our analysis which follows the GMR-1 standard. This decision was solely driven by the fact that the firmware of this satphone is publicly available from the vendor's website. Thuraya is a satellite communication provider based in Abu Dhabi that operates three satellites spanning most of the European, African, and Asian continents. The company also offers custom made satphones that integrate into their system easily, such as the analyzed Thuraya SO-2510.

We also later analyzed a second satphone (the GMR-2 Inmarsat IsatPhone Pro), which is not part of this thesis. However, the analysis showed that albeit the differences in employed hardware chips and software components, the overall procedure of reverse engineering as depicted in Section 2.4 largely applies to both phones. We thus think that the approach can be applied to other satphones in similar fashion.

In the following, we at first present the technical hardware details of the analyzed phone. The rest of the analysis is structured analogous to the four different steps presented in Section 2.4.

2.5.1 Hardware Architecture

The Thuraya SO-2510 runs on a Texas Instruments OMAP5910 hardware platform. The board is structurally identical to the OMAP1510. OMAP is a series of proprietary system on a chip (SoC) boards from Texas Instruments that are mainly used for image and video processing. The core of the platform is an ARM CPU along with a Texas Instruments C55x DSP processor. This information can be deduced from corresponding strings in the binary and from pictures of the actual components soldered on the circuit board [114]. Figure 2.5 provides a detailed functional overview of the architecture as a whole.

Both processors (i.e., the TI925T ARM and TMS320C55X DSP cores) can communicate with each other using the memory interface traffic controller (TC), which serves as the central hub that manages accesses between the different devices and buses in the system. There also exists two dedicated shared peripheral bus (one for the ARM and one for the DSP core) for connecting the processors to attached components such as USB devices and DSP peripherals.

Memory modules and their mapping to the processors' address space are of special interest for identifying the DSP initialization routines, as explained in Section 2.4. As depicted in the figure, both processors share the same SDRAM and can access additional memory (e.g., SRAM or Flash) on equal terms through the TC. The system is initialized by the ARM processor, i.e., DSP code or data has to be loaded by the ARM CPU into the specific memory regions of the DSP during the bootstrapping process. The DSP code can be located in the on-chip SARAM (which

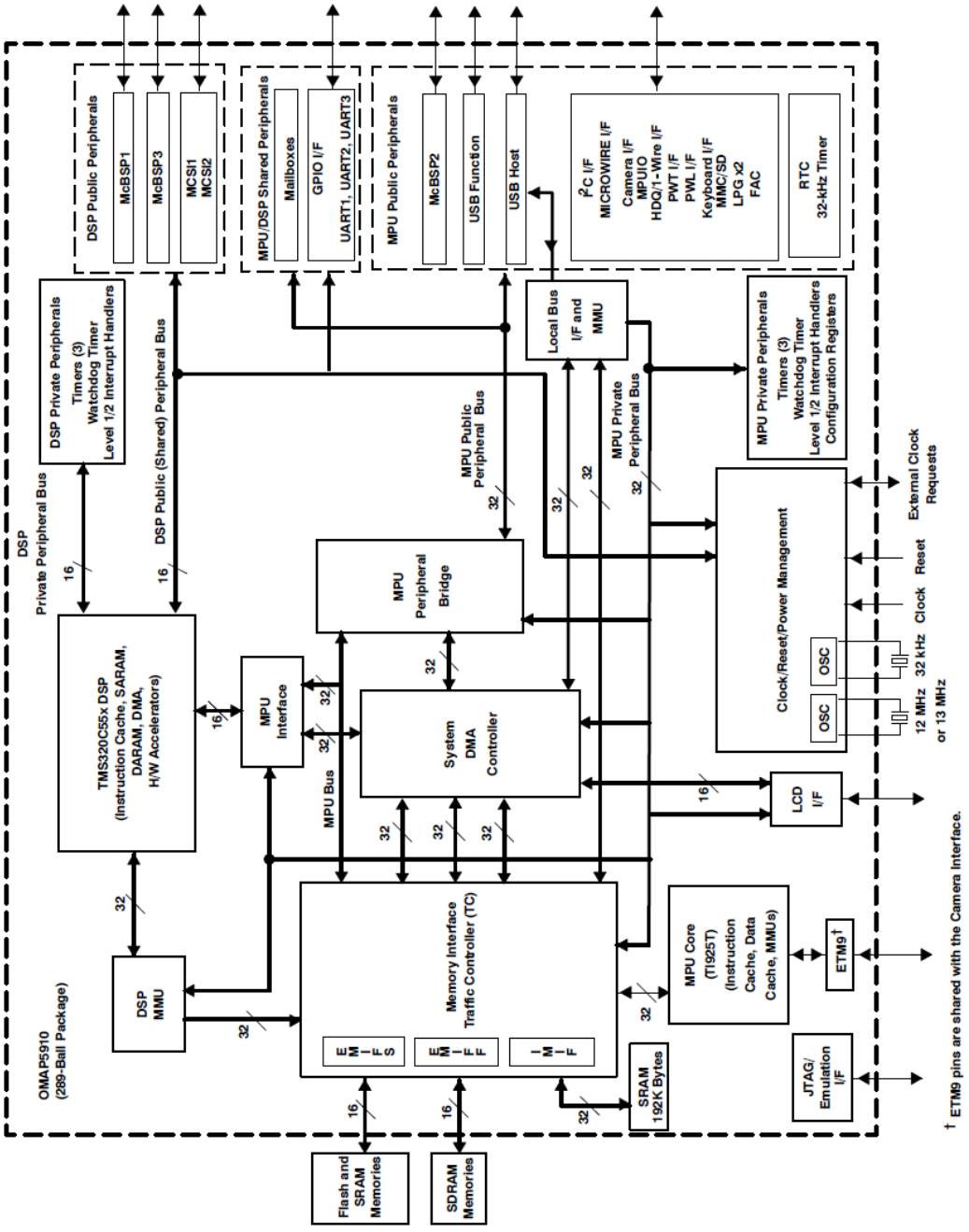


Figure 2.5: Functional overview of the OMAP5910 Platform [145].

Byte Address Range	On-Chip	External Interface
0x0000 0000 - 0x01FF FFFF		EMIFS (Flash CS0) 32M bytes
0x0200 0000 - 0x03FF FFFF	Reserved	
0x0400 0000 - 0x05FF FFFF		EMIFS (Flash CS1) 32M bytes
0x0600 0000 - 0x07FF FFFF	Reserved	
0x0800 0000 - 0x09FF FFFF		EMIFS (Flash CS2) 32M bytes
0x0A00 0000 - 0x0BFF FFFF	Reserved	
0x0C00 0000 - 0x0DFF FFFF		EMIFS (Flash CS3) 32M bytes
0x0E00 0000 - 0x0FFF FFFF	Reserved	
0x1000 0000 - 0x13FF FFFF		EMIFF (SDRAM) 64M bytes
0x1400 0000 - 0x1FFF FFFF	Reserved	
0x2000 0000 - 0x2002 FFFF	IMIF Internal SRAM 192K bytes	
0x2003 0000 - 0x2FFF FFFF	Reserved	
0x3000 0000 - 0x5FFF FFFF	Local Bus space for USB Host	
0x6000 0000 - 0xDFFF FFFF	Reserved	
0xE000 0000 - 0xE0FF FFFF	DSP public memory space (accessible via MPUI) 16M bytes	
0xE100 0000 - 0xEFFF FFFF	DSP public peripherals (accessible via MPUI)	
0xF000 0000 - 0xFFFA FFFF	Reserved	
0xFFFB 0000 - 0xFFFB FFFF	MPU public peripherals	
0xFFFC 0000 - 0xFFFC FFFF	MPU/DSP shared peripherals	
0xFFFD 0000 - 0xFFFE FFFF	MPU private peripherals	
0xFFFF 0000 - 0xFFFF FFFF	Reserved	

Figure 2.6: Global memory map of the ARM core of an OMAP5910 board [145].

holds 96 KB of memory) or in externally connected memory storages such as 192 KB sized SRAM. In both cases the memory holds unified code and data regions. Writes to the SARAM region of the DSP are therefore especially interesting for extracting the corresponding DSP code. The official OMAP5910 documents suggest that pre-defined memory regions are to be used by the two MMUs for mapping the various different memory types [145], as shown in Figure 2.6. The DSP public memory space is thus mapped to the address range `0xE0000000 - 0xE0FFFFFF`. This also includes the DSP SARAM memory. Furthermore, many peripheral registers are mapped into predefined address regions of the ARM address space from `0xFFFB0000` to `0xFFFEFFFF`. Among other things, these registers can be used to reset the DSP processor.

Since the DSP address space is constructed by a dedicated MMU, the specifications also suggest an own memory map for the DSP. This is depicted by Figure 2.7. The SARAM regions (and thus the DSP code) are therefore contained in the address range `0x010000 - 0x027FFF`.

Byte Address Range	Word Address Range	Internal Memory	External Memory
0x00 0000 - 0x00 FFFF	0x00 0000 - 0x00 7FFF	DARAM 64K bytes	
0x01 0000 - 0x02 7FFF	0x00 8000 - 0x01 3FFF	SARAM 96K bytes	
0x02 8000 - 0x04 FFFF	0x01 4000 - 0x02 7FFF	Reserved	
0x05 0000 - 0xFF 7FFF	0x02 8000 - 0x7F BFFF		Managed by DSP MMU
0xFF 8000 - 0xFF FFFF	0x7F C000 - 0x7F FFFF	PDROM (MPNMC = 0)	Managed by DSP MMU

Figure 2.7: Global memory map of the DSP core of an OMAP5910 board [145].

2.5.2 Firmware Extraction

The firmware of the Thuraya SO-2510 is publicly available as a 16 MB sized binary file from the vendor's website. The firmware file is neither (partially) packed nor encrypted and thus the ARM code can be analyzed directly without having to take any additional steps. The firmware version used in this thesis is `6.8_ML_20090421_143005`.

2.5.3 Virtual Memory Reconstruction

In the very beginning of the boot process of the satphone, the firmware image is loaded into the physical SDRAM address space and the ARM execution is started from the memory address `0x00000000`. The bootstrapping code then proceeds to initialize several memory mapped registers that steer various aspects of the memory mapping and attached peripherals. Subsequently, the code initializes the ARM virtual address space using the special coprocessor registers.

ARM employs a *Virtual Memory System Architecture* (VMSA) that is controlled by accessing dedicated MMU coprocessor registers. It enables the underlying operating system to construct a fine-grained and dynamical address space to, e.g., facilitate the implementation of multitasking and sophisticated access control systems. The virtual memory mapping is determined by a set of common two-level translations pages that separate the virtual address space into 4 KB (small) or 64 KB (large) pages. Please note that this thesis only covers small pages for the sake of brevity. All related figures on this section only apply to small page translation. The translation of large pages is very similar and is described in the official reference manual [23].

Figure 2.8 shows the translation of an ARM virtual address to the final physical address using the two-level small page translation process. The translation of an address is determined by the first-level and second-level descriptors which reside in the first and second page table respectively. The first-level descriptor is determined by the initial pointer to the first page table (the *translation base*) that is stored in a dedicated MMU register, along with *first-level index* (i.e. bits 20-31) of the virtual address. Consequently, the second-level descriptor is determined using the page

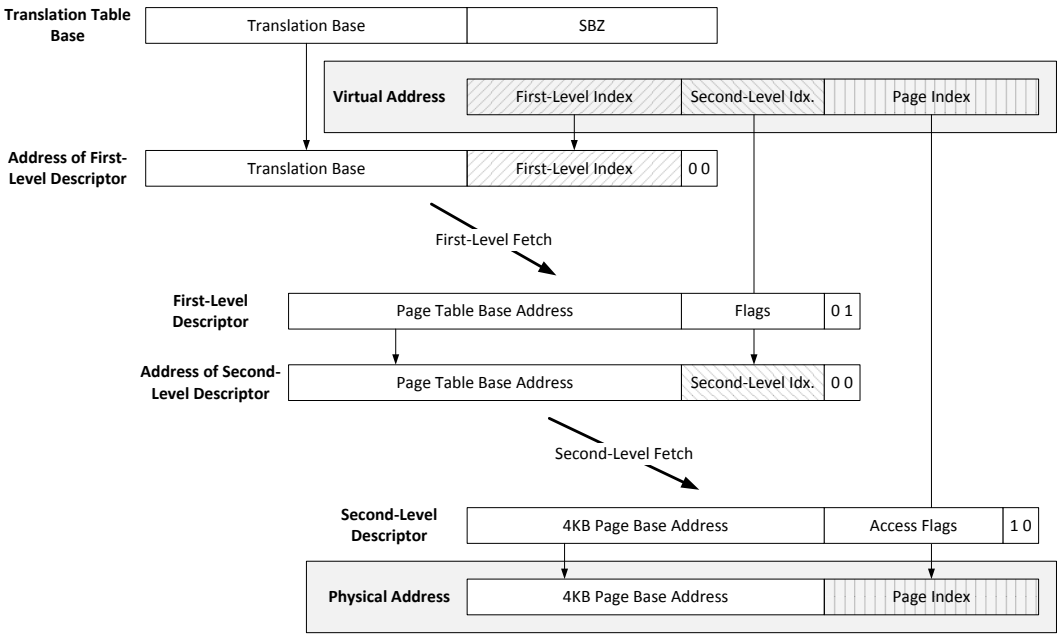


Figure 2.8: Illustration of the two-level ARM MMU virtual address translation process [23].

table base address of the first-level descriptor, along with *second-level index* (i.e. bits 12-19) of the virtual address. This yields the page base address in the second-level descriptor which (together with the *page index*) compose the final physical address. The translation process can be further customized by adjusting the special access flags in the lower bits of the page table descriptors. For example, this allows marking memory regions to be only accessible in privileged CPU modes. The virtual address mapping of the entire address space can be reconstructed if the translation base and thus all other page tables are known.

MMU registers can be read and modified using the coprocessor register `p15` and the special ARM coprocessor access instruction `MCR`. Since this register is only used for MMU control accesses, which appear rarely in comparison to other instructions, the corresponding initialization can be found by searching for `MCR` instructions that access the coprocessor register `p15`. We therefore implemented an IDA extension script that automates this task. Since the searched for code piece is located within the first hundred instructions of the bootstrapping code (which starts at offset `0`), it can also be spotted by manual human analysis in our case.

In the analyzed Thuraya SO-2510 firmware, there exist static page table descriptors that are already included in the firmware image. Thus, all first-level and second-level descriptors are already contained in the file. Consequently, the MMU

```
ROM:00000104 loc_104:
ROM:00000104                    LDR     R0, =0xFFFECC1C
ROM:00000108                    LDR     R1, =0x45A9
ROM:0000010C                    STR     R1, [R0]
ROM:00000110                    LDR     R1, =0x804000
ROM:00000114                    SUB     R1, R1, #0x800000
ROM:00000118                    MOV     R2, #1
ROM:0000011C                    MCR     p15, 0, R2,c3,c0, 0
ROM:00000120                    MCR     p15, 0, R1,c2,c0, 0
ROM:00000124                    MCR     p15, 0, R0,c8,c7, 0
```

List of Listings 2.1: An excerpt of the ARM MMU initialization code in the Thuraya SO-2510 firmware.

setup code confines itself to initializing MMU flags and setting the translation base register.

Listing 2.1 shows an excerpt of the found MMU initialization code. According to the official ARM reference manual, the translation base register is set by moving the appropriate pointer into the second coprocessor subregister `c2`. The instruction at offset `ROM:00000120` sets this subregister to `R1`, which in turn holds the static offset `0x804000 - 0x800000 = 0x4000`. Consequently, the first-level page table starts at the physical offset `0x4000` in the firmware image.

In the next step, we created an IDA extension script that automatically parses the entire static firmware page table and remaps the physical pages to their virtual addresses within IDA. After rerunning the auto-analysis of IDA, this yields a correct disassembly of the entire ARM code of the firmware, resulting in 12.451 identified subroutines and a total of 1.122.097 ARM instructions. All data and code addresses in the disassembly are valid and the ARM code is thus amenable to further analysis.

2.5.4 DSP Initialization

As explained previously, code that performs cryptographic operations can be implemented in a DSP more efficiently than in a common integer CPU such as ARM. It is therefore a reasonable assumption that the encryption algorithms are contained in the DSP assembler code.

The DSP has to be initialized by the ARM CPU during the bootstrap process by loading the DSP code and data into DSP accessible memory regions first (e.g. the DSP internal SARAM). We call this code and data the *DSP firmware* in the following. The DSP firmware is thus contained within the ARM firmware image. In the simplest case, it is stored as a single contiguous data region within the ARM firmware. In order to spot this region, one can make use of the fact that the TMSC55X DSP code follows a distinct format and structure, due to the structured

binary representation of instructions in form of opcodes and operand encodings. Unfortunately, this approach is not applicable to the Thuraya SO-2510 firmware as the DSP code is not present in plain contiguous form.

In order to retrieve the DSP firmware, a promising approach is to search for the DSP initialization routine in the ARM code. To that end, we propose two different ways to find the corresponding piece of code.

Static Strings Firstly, the firmware image contains plenty of referenced static strings. IDA uses a built-in heuristic to identify possible string messages in the auto-analysis phase. This results in 90.995 detected strings. It has to be noted that the actual number is significantly smaller due to imperfections in the string detection heuristic, which oftentimes incorrectly classifies short random byte sequences as ASCII strings. Nevertheless, the total number of true strings still amounts to tens of thousands. To our surprise, these strings often include debugging or assertion strings that reveal the name of variables and function that are otherwise unknown. Assertion and debugging strings are typically used in the software development process to find bugs, but are stripped from final release builds. It is unclear as to why this is not the case in the present Thuraya SO-2510 firmware. The firmware contains several strings that can be obviously associated to the DSP initialization routine, such as `Start DSP Image Loading...` at virtual address `0x8330F4`. Assertion strings also reveal the original source code file name of certain code parts. Most interestingly, at least four functions can be directly attributed to a source code file named `dsp_main`. This allows obtaining a coarse overview of DSP control code and greatly reduces the relevant code base and allows to search for the initialization code manually with a reasonable amount of effort.

DSP Memory References We also followed another more methodical approach that does not rely on the presence of redundant static strings. As explained in Section 2.5.1, the DSP memory space and the DSP peripherals are mapped into the ARM address range `0xE0000000 - 0xEFFFFFFF`. The peripheral registers are mapped within the address range `0xFFFB0000 - 0xFFFEFFFF`. DSP code and data has to be copied to DSP accessible memory and the DSP has to be configured and reset during the DSP startup process. As a natural consequence, the ARM code holds static memory references to the corresponding address ranges, which can again be searched for automatically in the code. We thus wrote a script that identifies all memory access instructions with references inside the mapped DSP memory range and a selected set of peripheral registers that are likely to be used in the previously described scenario. For example, we search for accesses to the `ARM_RSTCT1` register at address `0xFFFECE10`, which is responsible for resetting the DSP processor. This yields a total of 38 instructions in 16 different functions.

Further manual analysis reveals the presence of two independent DSP firmware load routines which reference two different DSP firmware images. For the sake of convenience, both functions are called `dsp_fwload_a` (address `0x00833110`, 123 ARM instructions) and `dsp_fwload_b` (address `0x015E7138`, 167 ARM instructions) in the following.

DSP Firmware Extraction

We reverse engineered the `dsp_fwload_a` function in the ARM code to deduce how the DSP firmware is assembled from the original firmware. Listing 2.2 shows a high-level C representation of the code. The loading routine takes a pointer argument to the initial DSP firmware (called `dsp_fw_img`). It consists of two loops that load the DSP image section-wise. Sections are small chunks of the DSP firmware that are located in the ARM firmware. Note that these sections do not necessarily have to be contiguous in the ARM firmware. The outer loop traverses the different sections, while the inner loop copies the corresponding data chunks of each section. As a consequence, section data is stored as one chunk in the original firmware. Furthermore, the data is byte-swapped using a 16 bit alignment because of differing endiannesses of the ARM and DSP cores. Because of sections being non-contiguous and byte swapping, it is not feasible to spot the DSP code making use of the special instruction encoding of the DSP architecture.

According to the official OMAP5910 system initialization instructions, the OMAP-5910 development tool chain provides a converter program `OUT2BOOT` that converts DSP code from the COFF object format of the DSP compiler into C arrays that can be included in the C source code as headers. The documentation [146] states

> the header file that OUT2BOOT produces consists of an array that holds the sections of code/data. A record consisting of the following fields represents each section:
>
> - The section length in 16-bit words
> - The DSP destination run-time address for the section.
> - The code/data words for the section.

The sample code that is provided in the documentation to load COFF sections described above from C array representations [146] is largely identical to the reverse engineered algorithm shown in Listing 2.2. We thus assume the code was borrowed from there.

Using the reverse engineered `dsp_fwload_a` algorithm, one can reassemble the final DSP image from the original firmware manually since references to the sections can be deduced from the code. Alternatively, it is also possible to attach a QEMU ARM emulator to IDA, let it execute only the `dsp_fwload_a` function and dump

```
void dsp_fwload_a(unsigned short *dsp_fw_img) {
  unsigned short section_length = dsp_fw_img[0] / 2;
  int next_section = 1;

  while(section_length) {
    char *section_addr = 0xC010000 + dsp_fw_img[next_section];
    int section_offset = next_section + 1;
    for(i = 0; i < section_length; i++) {
      *(short *) section_addr = (dsp_fw_img[section_offset] << 8) | (
          dsp_fw_img[section_offset + 1]);
      section_addr += 2;
      section_offset += 2;
    }

    section_length = dsp_fw_img[section_offset];
    next_section = section_offset + 1;
  }
}
```

List of Listings 2.2: Reverse engineered high-level C representation of `dsp_fwload_a`.

the contents of the final DSP image, which is then contained in the memory area starting from `0xC010000`. As described previously in Section 2.5.1, the DSP memory is mapped to `0xE0000000`. It is unclear why `dsp_fwload_a` rather loads the code to a different address. We assume that either the internal hardware-wired mapping was changed or that additional shared DSP memory is used to hold or load the DSP image. Please note that this is of no importance for the rest of the analysis. The DSP memory references are nevertheless important to spot the DSP peripheral register accesses that lead to both firmware load functions.

We also reverse engineered the second firmware load routine `dsp_fwload_b`. The corresponding high-level representation is depicted in Listing 2.3. The code bears significant resemblance to the first firmware load routine in that it also loads the DSP firmware from different sections that are appended in one large data byte array within the ARM firmware. However, every section has a section type at the beginning of the section data that determines the final load address in DSP memory. Section data is byte-swapped to take the differing endianness into account, just as in `dsp_fwload_a`. The final DSP firmware image can be obtained by reconstructing the firmware load routine using the statically stored firmware data byte array at address `0x158BE7C`. Again, it also possible to dynamically execute `dsp_fwload_b` in an emulator such as QEMU and dump the written DSP code and data starting from address `0x5000000`.

```
void dsp_fwload_b(unsigned short *dsp_fw_img) {
  unsigned short word_length = dsp_fw_img[0] / 2;

  if(!word_length)
    return;

  unsigned short next_offset = 1;
  do {
    unsigned short section_type = dsp_fw_img[next_offset];
    unsigned int section_base_address;
    switch(section_type) {
      case 1:
        section_base_address = 0x5010000;
        break;
      case 2:
        section_base_address = 0x5020000;
        break;
      case 16:
        section_base_address = 0x5100000;
        break;
      case 17:
        section_base_address = 0x5110000;
        break;
      default:
        section_base_address = 0x5000000;
    }

    unsigned short *section_addr = (unsigned short *)
      (section_base_address + (dsp_fw_img[next_offset + 1] & 0xFFFE))
        ;
    next_offset += 2;
    for(int i = 0; i < word_length; i++, section_addr += 2,
                                         next_offset += 2)
      *section_addr = dsp_fw_img[next_offset + 1] |
                      (dsp_fw_img[next_offset] << 8);

    section_size = dsp_fw_img[next_offset++];
    word_length = section_size / 2;
  } while(word_length);
}
```

List of Listings 2.3: Reverse engineered high-level C representation of `dsp_fwload_b`.

2.5.5 Crypto Code Identification

An initial rudimentary analysis of both extracted DSP firmware images using the IDA disassembler reveals that the first firmware (taken from `dsp_fwload_a`) only contains a small amount of actual DSP instructions. It contains ten different functions which are mostly responsible for processor initialization tasks. We assume that this is rather a test image. Since also the DSP initialization routine `dsp_fwload_a` greatly resembles the examples from the official documentation, it might be reasonable to assume that this is sample code that was not stripped from the final release build. For the rest of the analysis, we thus focused on the analysis of the second DSP firmware obtained from `dsp_fwload_b`.

The second DSP firmware image holds a total of 22.575 instructions organized in 384 different functions. In contrast to the ARM firmware, no strings, symbols, or similar meta data are present or available. Since the cryptographic code can only constitute a small share of the entire firmware code that cannot be identified manually without huge effort, we applied the previous work on identifying cryptographic primitives as described in Section 2.2. We confined ourselves to approaches that can be applied statically, as we had no real satphone at our disposal at the time of the analysis. If this were the case, then accessing a debugging interface on the hardware device thorugh the widespread JTAG interface would have been possible. This approach can be cumbersome nevertheless, since debugging interfaces are oftentimes disabled in release boards. Furthermore, it is infeasible to apply approaches that require the availability of concrete algorithms beforehand (e.g., signature-based searches), since we know nothing about the actual GMR-1 crypto algorithms.

Another problem that has to be addressed is the presence of compression and encoding algorithms on the firmware. Due to the nature of the algorithms, they also happen to contain very similar instructions types than cryptographic code. This does not the render related search approaches useless, but merely makes it more complicated to confirm that a given piece of code is actually part of an encryption or decryption process.

Since GMR-1 is derived from GSM, we speculated that the cipher algorithm employed in GMR-1 bears at least some resemblance to the GSM-A5/2 cipher from GSM (see Section 2.2.5). Due to the nature of this algorithm (e.g., the presence of feedback shift registers), the cipher code is as well bound to contain a lot of bit shift and XOR operations — unless it is somehow obfuscated.

We thus implemented a plugin for IDA that counts the occurrences of such instructions in each function and sets them in relation to the total number of instructions in the function; on the analogy of the ideas to spot cryptographic primitives that have been described in Section 2.3. Table 2.1 lists the six top-rated functions found when using this heuristic. The four topmost functions are rather short sequences of code that bear striking resemblance to feedback register shift operators and they

Function address	% relevant instr.
0001D038	43%
0001CFC8	41%
0001D000	41%
0001D064	37%
00014C9C	25%
00014CAC	25%

Table 2.1: Functions rated by percentage of relevant arithmetic and logical instructions.

are adjacent in memory, which suggest they are related. A disassembly and a high level C representation of one of the functions is depicted in Listing 2.4. Further analyzing the call sites of these four functions reveal an accessed memory region holding variables which equal

- the assumed key length,
- the assumed number and length of the feedback registers, and
- the assumed frame-number lengths (see Section 2.4.1).

These points, plus the fact that GSM-A5/2 also contains four LFSR clocking routines, are strong indicators that one has spotted the correct region of the crypto code and not another resembling algorithm such as encoding or speech compression. Starting from this code area, we manually reverse engineered the relevant code portions to obtain the cryptographic algorithm employed in the DSP. In our derivate implementation, the resulting C source spans over 184 lines of code and contains 9 functions. The source code is not listed in this thesis for the sake of brevity, but it can be publicly downloaded from our GMR website [29]. The correctness of the reverse engineered algorithm could be verified from an external source by decrypting previously recorded satphone traffic where the crypto key could be extracted from the SIM card of the device.

2.5.6 Cipher and Weaknesses

We now give a short summary of the layout of the extracted GMR-A5-1 cipher. More details and insights into the algorithm and its attack surfaces can be found in the corresponding paper [52]. Figure 2.9 shows the structure of the algorithm. As can be seen at first glance, the algorithm is very similar to GSM-A5/2. It also consists of four LFSRs along with three majority function elements. The authors of GMR-A5-1 mainly interchanged the bit positions that serve as inputs and outputs during clocking and the keystream generation.

```
ROM:1D038    mov      dbl(*abs16(#1EF4h)), AC1
ROM:1D03D    sftl     AC1, #-1h, AC2
ROM:1D040    mov      dbl(*abs16(#1EF4h)), AC1
ROM:1D045    xor      AC2, AC1
ROM:1D047    bfxtr    #0FFF0h, AC1, AR1
ROM:1D04B    and      #1h, AR1, AC3
ROM:1D04E    and      #1h, AC1, AC1
ROM:1D051    xor      AC3, AC1
ROM:1D053    xor      AC0, AC1
ROM:1D055    sftl     AC1, #16h, AC0
ROM:1D058    xor      AC2, AC0
ROM:1D05A    mov      AC0, dbl(*abs16(#1EF4h))
ROM:1D05F    ret
```

```
// #1EF4h -> reg3
void update_reg3(int arg) {
  int t = BIT((reg3 >> 1) ^ reg3, 5);
  reg3 = (((t ^ BIT((reg3 >> 1) ^ reg3, 1) ^ arg) << 22) ^ (reg3 >>
      1);
}
```

List of Listings 2.4: Disassembly and corresponding high-level C representation of one DSP LFSR clock routine.

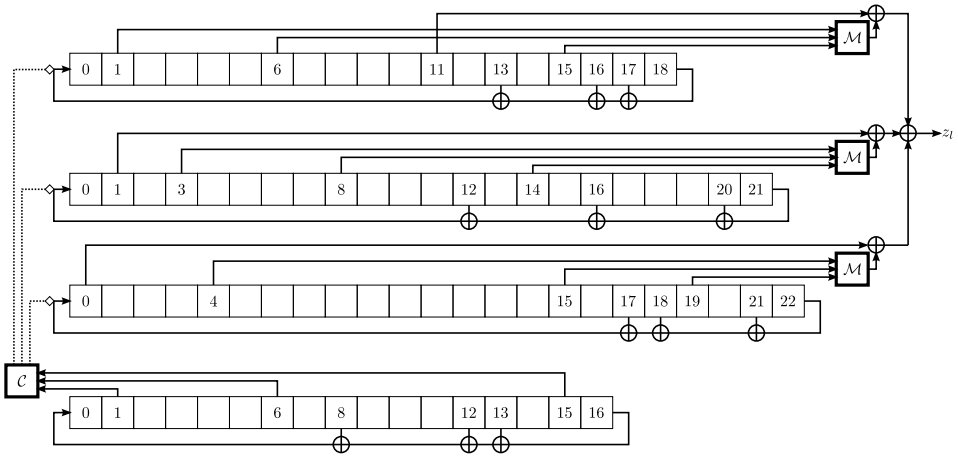


Figure 2.9: The GMR-A5-1 cipher.

The algorithm can work in two different modes of operation: *initialization* and *generation*. In initialization mode, the cipher at first performs the following four steps:

1. Set all four LFSRs to zero.
2. Compute initialization value I by combining frame number with encryption key using a certain xor scheme.
3. Clock I into all four registers
4. Set least-significant bits of all LFSRs to zero.

The algorithm then proceeds to execute 250 clocks, discards the output and is ready to operate in generation mode to produce keystream data. The number of keystream bits varies depending on the current channel context. The standard specifies different traffic and control channels (abbreviated as TCHx and SDCCH/SACCH/FACCHx respectively).

In an additional crypto analysis (which is not part of this thesis), it could be shown that traditional attacks which were presented against GSM-A5/2 can also be mounted against GMR-A5-1. More specifically, previous attacks presented by Petrovic and Fuster-Sabater [122] can be extended for a known-keystream attack. This allows to reconstruct the internal states of the keystream generation within reasonable computational effort and thus decrypt all transmitted packets without any additional knowledge about the key material.

2.6 Conclusion and Future Work

In this chapter, we presented a generic approach to reverse engineering unknown encryption algorithms from mobile devices and performed an actual analysis of a Thuraya SO-2510 phone that implements the widespread GMR-1 satellite telephony standard. GMR-1 uses a proprietary encryption scheme that is not part of the official specification documents. In order to assess the resistance of GMR-1 satphone against real world eavesdropping attacks, we reverse engineered the unknown encryption algorithms. Previous research efforts already proved that the GSM mobile communication standards (which are closely related to GMR) employ a vulnerable encryption scheme and we speculated that similar attacks can be ported to GMR-1.

We presented the individual steps from reconstructing the virtual memory layout of the device firmware to finding the unknown encryption algorithm in-detail. Analyzing mobile devices greatly differentiates from traditional desktop or server based reverse engineering due to the different hardware layout and software components adjusted accordingly. We showed how these challenges can be tackled and leveraged previous work on identifying cryptographic primitives in binary code.

Our analysis verified our concerns that GMR-1 uses an insufficient algorithm. In fact, we showed that GMR-A5-1 structurally closely resembles GSM-A5/2 which is

known to be vulnerable by modern standards. This poses a significant threat to security-sensitive scenarios in which the privacy of the transmission can make the difference between life and death. Our findings reinforce our opinion that encryption algorithms should be made available to the public so that researchers can test and verify their resistance against attacks rather than relying on the security by obscurity principle.

In terms of future work, it would be interesting to to analyze more proprietary mobile devices to confirm the general applicability of our approach. We already successfully conducted the analysis of the competing GMR-2 standard and also identified its proprietary encryption algorithm [52]. Another interesting topic for future work is to increase the automation of the reverse engineering process to accelerate and facilitate the analysis.

3

Practical Timing Side Channel Attacks Against Kernel Space ASLR

3.1 Introduction

Modern operating systems employ a wide variety of methods to protect both user and kernel space code against memory corruption attacks that leverage vulnerabilities such as stack overflows [12], integer overflows [32], and heap overflows [45]. Control flow hijacking attempts pose a significant threat and have attracted a lot of attention in the security community due to their high relevance in practice. Even nowadays, new vulnerabilities in applications, drivers, or operating system kernels are reported on a regular basis. To thwart such attacks, many mitigation techniques have been developed over the years. A few examples that have received widespread adoption include stack canaries [47], non-executable memory (e.g., No eXecute (NX) bit and Data Execution Prevention (DEP) [106]), and Address Space Layout Randomization (ASLR) [30, 117, 163].

Especially ASLR plays an important role in protecting computer systems against software faults. The key idea behind this technique is to randomize the system's virtual memory layout either every time a new code execution starts (e.g., upon process creation or when a driver is loaded) or on each system reboot. While the initial implementations focused on randomizing user mode processes, modern operating systems such as Windows 7 randomize *both* user and kernel space. ASLR introduces diversity and randomness to a given system, which are both appealing properties to defend against attacks: an attacker that aims to exploit a memory corruption vulnerability does not know *any* memory addresses of data or code sequences which are needed to mount a control flow hijacking attack. Even advanced exploitation techniques like return-to-libc [136] and return-oriented programming (ROP) [133] are hampered since an attacker does not know the virtual address of memory locations to which she can divert the control flow. As noted above, all

major operating systems such as Windows, Linux, and Mac OS X have adopted ASLR and also mobile operating systems like Android and iOS have recently added support for this defense method [34, 117, 127, 39].

Broadly speaking, successful attacks against a system that implements ASLR rely on one of three conditions:

1. In case not all loaded modules and other mapped memory regions have been protected with ASLR, an attacker can focus on these regions and exploit the fact that the system has not been fully randomized. This is an adoption problem and we expect that in the near future all memory regions (both in user space and kernel space) will be fully randomized [93, 72]. In fact, Windows 7/8 already widely supports ASLR and the number of applications that do not randomize their libraries is steadily decreasing. Legacy libraries can also be forced to be randomized using the Force ASLR feature.

2. If some kind of information leakage exists that discloses memory addresses [54, 141, 91], an attacker can obtain the virtual address of specific memory areas. She might use this knowledge to infer additional information that helps her to mount a control flow hijacking attack. While such information leaks are still available and often used in exploits, we consider them to be software faults that will be fixed to reduce the attack surface [9, 11].

3. An attacker might attempt to perform a brute-force attack [134]. In fact, Shacham et al. showed that user mode ASLR on 32-bit architectures only leaves 16 bit of randomness, which is not enough to defeat brute-force attacks. However, such brute-force attacks are *not* applicable for kernel space ASLR. More specifically, if an attacker wants to exploit a vulnerability in kernel code, a wrong offset will typically lead to a complete crash of the system and thus an attacker has only *one* attempt to perform an exploit. Thus, brute-force attacks against kernel mode ASLR are not feasible in practice.

In combination with DEP, a technique that enforces the $W \oplus X$ (**W**ritable **xor** e**X**ecutable) property of memory pages, ASLR significantly reduces the attack surface. Under the assumption that the randomization itself cannot be predicted due to implementation flaws (i.e., not fully randomizing the system or existing information leaks), typical exploitation strategies are severely thwarted.

3.1.1 Contributions

In this chapter, we study the limitations of kernel space ASLR against a local attacker with restricted privileges. We introduce a generic attack for systems running on the Intel *Instruction Set Architecture* (ISA). More specifically, we show how a local attacker with restricted rights can mount a timing-based side channel

attack against the memory management system to deduce information about the privileged address space layout. We take advantage of the fact that the memory hierarchy present in computer systems leads to shared resources between user and kernel space code that can be abused to construct a side channel. In practice, timing attacks against a modern CPU are very complicated due to the many performance optimizations used by current processors such as hardware prefetching, speculative execution, multi-core architectures, or branch prediction that significantly complicate timing measurements [109]. Previous work on side-channels attacks against CPUs [7, 79, 147] focused on older processors without such optimization and we had to overcome many challenges to solve the intrinsic problems related to modern CPU features [109].

We stress that our side channel attacks are generic and we tested our approach on both Windows and Linux systems. However, it is still necessary to tailor our attacks to the concrete ASLR implementation of the operating system (see Section 3.5). Since ASLR implementations always compromise with regards to the used randomness, it is reasonable and practically required to make use of these assumptions in the attacks. However, the kernel ASLR implementation of the most prevalent desktop operating system Windows is proprietary and no details are known publically. In this thesis, we put a special focus on proprietary Windows operating systems. Therefore, we had to reverse engineer the undocumented Windows kernel ASLR implementation. We also provide a mitigation solution at the end that retrofits the Windows operating system with an improved exception handler that thwarts our presented attacks.

We have implemented three different attack strategies that are capable of successfully reconstructing (parts of) the kernel memory layout. We have tested these attacks on different Intel and AMD CPUs (both 32- and 64-bit architectures) on machines running either Windows 7 or Linux. Furthermore, we show that our methodology also applies to virtual machines. As a result, an adversary learns precise information about the (randomized) memory layout of the kernel. With that knowledge, she is enabled to perform control flow hijacking attacks since she now knows where to divert the control flow to, thus overcoming the protection mechanisms introduced by kernel space ASLR. Furthermore, we also discuss mitigation strategies and show how the side channel we identified as part of this work can be prevented in practice with negligible performance overhead.

In summary, the contributions of this thesis are the following:

- We present a generic attack to derandomize kernel space ASLR that relies on a side channel based on the memory hierarchy present in computer systems, which leads to timing differences when accessing specific memory regions. Our attack is applicable in scenarios where brute-force attacks are not feasible and we assume that no implementation flaws exist for ASLR. Because of the

general nature of the approach, we expect that it can be applied to many operating systems and a variety of hardware architectures.

- We present three different approaches to implement our methodology. We successfully tested them against systems running Windows 7 or Linux on both 32-bit and 64-bit Intel and AMD CPUs, and also the virtualization software VMware. As part of the implementation, we reverse-engineered an undocumented hash function used in Intel Sandybridge CPUs to distribute the cache among different cores. Our attack enables a local user with restricted privileges to determine the virtual memory address of key kernel memory locations within a reasonable amount of time, thus enabling ROP attacks against the kernel.
- We discuss several mitigation strategies that defeat our attack. The runtime overhead of our preferred solution is not noticeable in practice and successfully prevents the timing side channel attacks discussed in this thesis. Furthermore, it can be easily adopted by OS vendors.

This chapter is based on a previous publication together with Willems and Holz [82].

3.2 Technical Background

We review the necessary technical background information before introducing the methodology behind our attack.

3.2.1 Address Space Layout Randomization

As explained above, ASLR randomizes the system's virtual memory layout either every time a new code execution starts or every time the system is booted [30, 94, 117, 163]. More specifically, it randomizes the base address of important memory structures such as for example the code, stack, and heap. As a result, an adversary does not know the virtual address of relevant memory locations needed to perform a control flow hijacking attack (i.e., the location of shellcode or ROP gadgets). All major modern operating systems have implemented ASLR. For example, Windows implements this technique since Vista in both user and kernel space [127], Linux implements it with the help of the PaX patches [117], and MacOS ships with ASLR since version 10.5. Even mobile operating systems such as Android [34] and iOS [39] perform this memory randomization nowadays.

The security gain of the randomization is twofold: First, it can protect against remote attacks, such as hardening a networking daemon against exploitation. Second, it can also protect against local attackers by randomizing the privileged address

space of the kernel. This should hinder exploitation attempts of implementation flaws in kernel or driver code that allow a local application to elevate its privileges, a prevalent problem [42, 142]. Note that since a user mode application has no means to *directly* access the kernel space, it cannot determine the base addresses kernel modules are loaded to: every attempt to access kernel space memory from user mode results in an access violation, and thus kernel space ASLR effectively hampers local exploits against the OS kernel or drivers.

Windows Kernel Space ASLR

In the following we describe the kernel space ASLR implementation of Windows (both 32-bit and 64-bit). The information presented here applies to Vista, Windows 7, and Windows 8. We obtained this information by reverse-engineering the corresponding parts of the operating system code.

During the boot process, the Windows loader is responsible for loading the two core components of the OS, the kernel image and the hardware abstraction layer (HAL), which is implemented as a separate module. At first, the Windows loader allocates a sufficiently large address region (the *kernel_region*) for the kernel image and the HAL. The base address of this region is constant for a given system. Then, it computes a random number ranging from 0 to 31. This number is multiplied by the page size (0x1000) and added to the base address of the reserved region to form a randomized load address. Furthermore, the order in which the kernel and the HAL are loaded is also randomized. Both components are always loaded consecutively in memory, there is no gap in between. This effectively yields 64 different slots to which the kernel image and the HAL each can be loaded (see also Figure 3.1). In summary, the formula for computing the kernel base address is as follows:

$$k\_base = kernel\_region + (r_1 * 0x1000) + (r_2 * hal\_size),$$

where $r_1 \in \{0 \dots 31\}$ and $r_2 \in \{0, 1\}$ are random numbers within the given ranges. Kernel and HAL are commonly mapped using so called *large pages* (2 MB) which improves performance by reducing the duration of page walks; both components usually require three large pages (= 6 MB). An interesting observation is that the randomization is already applied to the *physical* load addresses of the image and that for the *kernel_region*, the formula

$$virtual\_address = 0x80000000 + physical\_address$$

holds. The lower 31 bits of virtual kernel addresses are thus identical to the physical address. Again, this is only true for addresses in the *kernel_region* and does not generally apply to kernel space addresses. For the rest of the chapter, note that we assume that the system is started without the `/3GB` boot option that restricts the

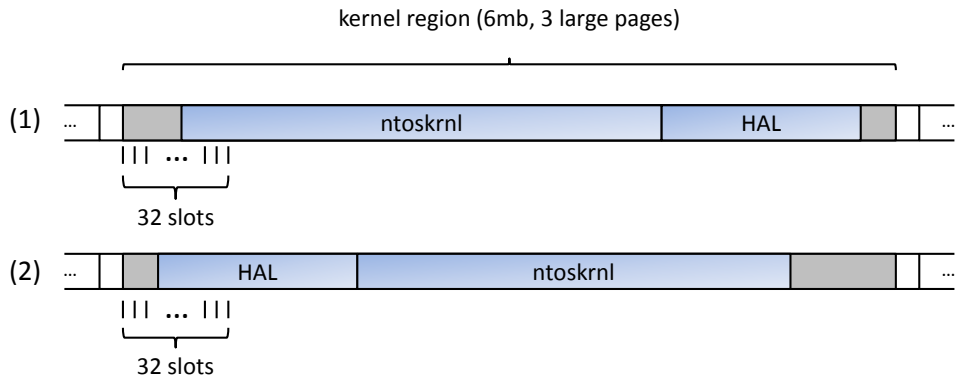


Figure 3.1: ASLR for Windows *kernel_region* (not proportional). Slot and load order (either (1) or (2)) are chosen randomly.

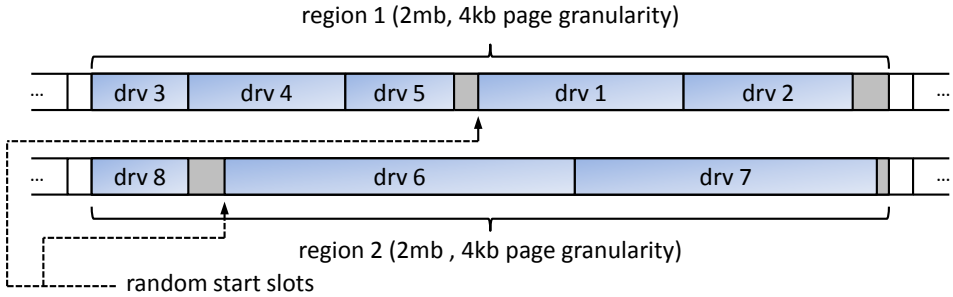


Figure 3.2: Two example driver regions randomized using Windows kernel ASLR.

kernelspace to 1 GB. In this case, the kernelspace base address would be $0xC0000000$ instead.

Once the kernel is initialized, all subsequent drivers are loaded by the kernel's driver load routine `MmLoadSystemImage`. This mechanism contains a different ASLR implementation to randomize the base address of drivers in the subroutine `MiReserveDriverPtes`. The process works as follows: the kernel first reserves a memory region of 2 MB using standard 4 KB sized pages (a *driver_region*). It then randomly chooses one out of 64 page-aligned start slots in this region where the first driver is loaded to. All subsequent drivers are then appended, until the end of the 2 MB region is hit, which is when the next driver is mapped to the beginning of the region (i.e., a wrap-around occurs). In case a region is full, a new 2MB *driver_region* with a random start slot is allocated (see Figure 3.2). For session-wide drivers such as `win32k.sys`, a similar randomization with 64 slots for each driver image is applied in a dedicated *session_driver_region*. We observed that the loading order of drivers is always the same in practice.

3.2.2 Memory Hierarchy

There is a natural trade-off between the high costs of *fast* computer memory and the demand for *large* (but inexpensive) memory resources. Hence, modern computer systems are operating on hierarchical memory that is built from multiple stages of different size and speed. Contemporary hierarchies range from a few very fast CPU registers over different levels of cache to a huge and rather slow main memory. Apparently, with increasing distance to the CPU the memory gets slower, larger, and cheaper.

We focus on the different *caches* that are used to speed up address translation and memory accesses for code and data. As illustrated in Figure 3.5, each CPU core typically contains one dedicated *Level 1 (L1)* and *Level 2 (L2)* cache and often there is an additional *Level 3 (L3)* shared cache (also called *last level cache (LLC)*). On level 1, instructions and data are cached into distinct facilities (*ICACHE* and *DCACHE*), but on higher stages unified caches are used. The efficiency of cache usage is justified by the *temporal* and *spatial locality* property of memory accesses [4]. Hence, not only single bytes are cached, but always chunks of adjacent memory. The typical size of such a *cache line* on x86/x64 is 64 bytes.

Different forms of cache organization are possible and the two most important design decisions are: *where* to store a certain block of memory in the cache (*associativity)* and *which* block to remove when the cache is full (*replacement strategy*). In a *full associative* cache, every memory block can be placed anywhere in the cache, resulting in the necessity to search the complete cache when a certain memory block is looked for. On the contrary, in a *direct mapped* cache each memory location can only reside in one single dedicated cache slot. This decreases the lookup complexity dramatically, but also increases the probability of collisions and, therefore, the cache miss rate. A typically used trade-off is an *n-way set associative* mapping, in which memory addresses can reside in n different cache locations.

In order to find out if the data from a certain memory location is currently stored in a cache, not only this data but also (a part of) its address has to be stored. This address information is called *tag* and its necessary size is determined by the mapping strategy. In a full associative cache, the complete address needs to be stored. In so called *set associative caches*, or direct mapped caches, the requested address is split into an index and a remaining tag value. The index then decides which cache slots' tag values have to be verified. Figure 3.3 illustrates this situation. As an example, consider a 32-bit address and a 256 KB cache with 64 byte cachelines. The cache is *4-way set associative* and, thus, the available slots are grouped into 1024 different sets ($= 256 * 1024/(64 * 4)$). Hence, 10 bits are necessary in order to select the appropriate set (step 1) and these form the *cache index*. The upper 16 bits of the address are then used as *tag value* and are stored with each cacheline to correctly identify (step 2) each stored address. To finally address (step 3) one of the 64 bytes

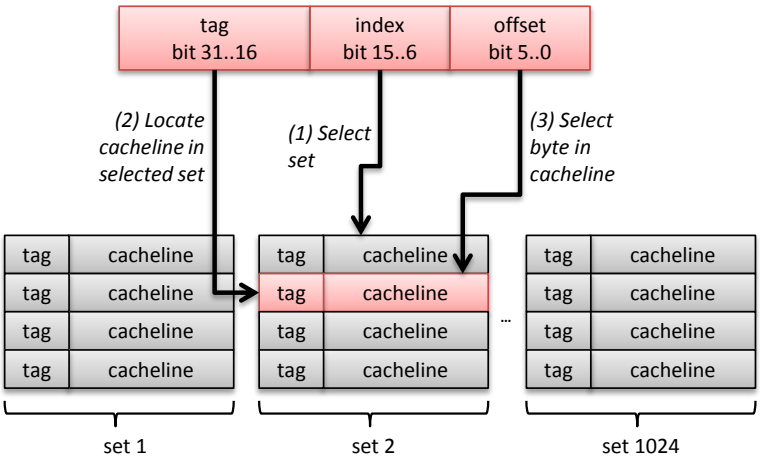


Figure 3.3: Address to cache index and tag mapping.

within each cacheline, the lower 6 address bits are used (called *(block) offset*).

An important question is where to store certain memory content in the caches and how to locate it quickly on demand. All described caches operate in an *n-way set associative* mode. Here, all available slots are grouped into sets of the size n and each memory chunk can be stored in all slots of one particular set. This target set is determined by a bunch of *cache index* bits that are taken from the memory address. As an example, consider a 32-bit address and a typical L3 cache of 8 MB that is 16-way set associative. It consists of $(8,192 * 1,024)/64 = 131,072$ single slots that are grouped into $131,072/16 = 8,192$ different sets. Hence, 13 bits are needed to select the appropriate set. Since the lower 6 bits (starting with bit 0) of each address are used to select one particular byte from each cacheline, the bits 18 to 6 determine the set. The remaining upper 13 bits form the *address tag*, that has to be stored with each cache line for the later lookup.

One essential consequence of the *set associativity* is that memory addresses with identical index bits compete against the available slots of one set. Hence, memory accesses may evict and replace other memory content from the caches. One common replacement strategy is *Least Recently Used (LRU)*, in which the entry which has not been accessed for the longest time is replaced. Since managing real timestamps is not affordable in practice, the variant *Pseudo-LRU* is used: an additional *reference bit* is stored with each cacheline that is set on each access. Once all reference bits of one set are enabled, they are all cleared again. If an entry from a set has to be removed, an arbitrary one with a cleared reference bit is chosen.

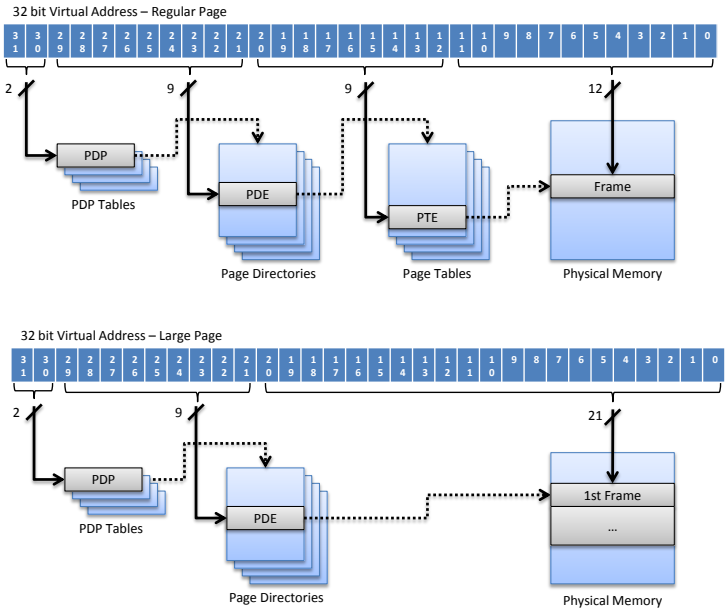


Figure 3.4: Address resolution for regular and large pages on PAE systems.

Virtual Memory and Address Translation

Contemporary operating systems usually work on *paged virtual memory* instead of physical memory. The memory space is divided into equally sized pages that are either *regular pages* (e.g., with a size of 4 KB), or *large pages* (e.g., 2 or 4 MB). When accessing memory via virtual addresses (VA), they first have to be translated into physical addresses (PA) by the processor's *Memory Management Unit* (MMU) in a *page walk*: the virtual address is split into several parts and each part operates as an array index for certain levels of *page tables*. The lowest level of the involved *paging structures* (PS), the *Page Table Entry* (PTE), contains the resulting physical *frame number*. For large pages, one level less of PS is needed since a larger space of memory requires less bits to address. In that case, the frame number is stored one level higher in the *Page Directory Entry* (PDE). In case of *Physical Address Extension* (PAE) [84] or 64-bit mode, additional PS levels are required, i.e. the *Page Directory Pointer* (PDP) and the *Page Map Level 4* (PML4) structures.

Figure 3.4 illustrates the address resolution for regular pages (upper part) and large pages (lower part) on PAE systems. Notice that in the first case, the resulting PTE points to one single frame. In the second case, the PDE points to the first one of a set of adjacent frames, that in sum span the same size as a large page.

In order to speed up this address translation process, resolved address mappings are cached in *Translation Lookaside Buffers* (TLBs). Additionally, there often are

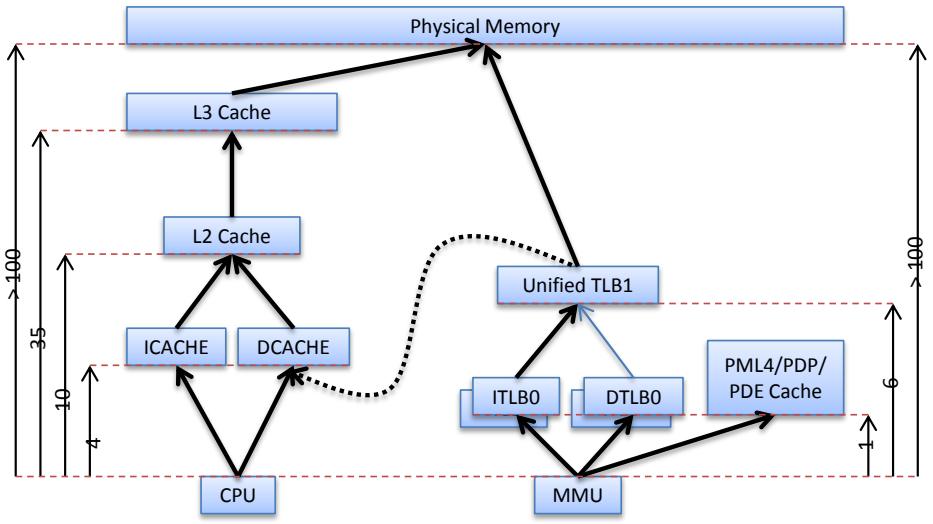


Figure 3.5: Intel i7 memory hierarchy plus clock latency for the relevant stages (based on [90, 101]).

dedicated caches for the involved higher level PS [83]. Depending on the underlying system, the implementation of these translation caches differs a lot. Current x86/x64 systems usually have two different levels of TLB: the first stage *TLB0* is split into one for data (DTLB) and another for instructions (ITLB), and the second stage TLB1 is used for both. Further, the TLBs are often split into one part for regular pages and another for large pages.

Even with TLBs and PS caches, the address translation takes some clock cycles, during which the resulting physical address is not available yet. As an effect, the system has to wait for the address translation *before* it can check the tag values of the caches. Therefore, lower caches (mostly only the L1 cache) are *virtually indexed, but physically tagged*. This means that the cache index is taken from the virtual address but the stored tag values from the physical one. With that approach, the corresponding tag values already can be looked up and then quickly compared once the physical address is available.

Figure 3.5 illustrates all the different caching facilities of the *Intel i7* processor. The vertical arrows are labeled with the amount of clock cycles that are normally required to access the particular stages [101, 90]. The dashed arrow (pointing from the TLB1 to the DCACHE) indicates that PS are not only cached in the TLB or PML4/PDP/PDE caches, but may also reside as regular data within the DCACHE or higher level unified caches.

An essential part of each virtual memory system is the *page fault handler* (PFH).

It is invoked if a virtual address cannot be resolved, i.e., the page walk encounters invalid PS. This may happen for several reasons (e.g., the addressed memory region has been swapped out or the memory is accessed for the first time after its allocation). In such cases, the error is handled completely by the PFH. Although this happens transparently, the process induces a slight time delay. Besides translation information, the PS also contain several protection flags (e.g., to mark memory as *non-executable* or to restrict access to privileged code only). After successful translation, these flags are checked against the current system state and in case of a protection violation, the PFH is invoked as well. In that case an access violation exception is generated that has to be caught and handled by the executing process. Again, a slight time delay may be observable between accessing the memory and the exception being delivered to the exception handler.

3.3 Related Work

Timing and side channel attacks are well-know concepts in computer security and have been used to attack many kinds of systems, among others cryptographic implementations [97, 152, 5], OpenSSL [6, 37], SSH sessions [138], web applications [40, 64], encrypted VoIP streams [160, 155], and virtual machine environments [125, 165, 166].

Closely related to our work is a specific kind of these attacks called *cache games* [35, 79, 119, 147]. In these attacks, an adversary analyzes the cache access of a given system and deduces information about current operations taking place. The typical target of these attacks are cryptographic implementations: while the CPU performs encryption or decryption operations, an adversary infers memory accesses and uses the obtained information to derive the key or related information. In a recent work, Gullasch et al. showed for example how an AES key can be recovered from the OpenSSL 0.9.8n implementation [79] and Zhang et al. introduced similar attacks in a cross-VM context [166].

We apply the basic principle behind cache attacks in our work and introduce different ways how this general approach can be leveraged to obtain information about the memory layout of a given system. Previous work focused on attacks against the instruction/data caches and not on the address translation cache, which is conceptually different. We developed novel approaches to attack this specific aspect of a computer system. Furthermore, all documented cache attacks were implemented either for embedded processors or for older processors such as Intel Pentium M (released in March 2003) [79], Pentium 4E (released in February 2004) [147], or Intel Core Duo (released in January 2006) [7]. In contrast, we focus on the latest processor architectures and need to solve many obstacles related to modern performance optimizations in current CPUs [109]. To the best of our knowledge, we are the first

to present timing attacks against ASLR implementations and to discuss limitations of kernel space ASLR against a local attacker.

3.4 Timing Side Channel Attacks

Based on this background information, we can now explain how time delays introduced by the memory hierarchy enable a side channel attack against kernel-level ASLR.

3.4.1 Attacker Model

We focus in the following on local attacks against kernel space ASLR: we assume an adversary who already has restricted access to the system (i.e., she can run arbitrary applications) but does not have access to privileged kernel components and thus cannot execute privileged (kernel mode) code. We also assume the presence of a user mode-exploitable vulnerability within kernel or driver code, a common problem [42]. The exploitation of this software fault requires to know (at least portions of) the kernel space layout since the exploit at some point either jumps to an attacker controlled location or writes to an attacker controlled location to divert the control flow.

Since the entire virtual address space is divided in both user and kernel space, an attacker might attempt to directly jump to a user space address from within kernel mode in an exploit, thus circumventing any kernel space ASLR protections. However, this is not always possible since the correct user space might not be mapped at the time of exploitation due to the nature of the vulnerability [93]. Furthermore, this kind of attack is rendered impossible with the introduction of the *Supervisor Mode Execution Protection* (SMEP) feature of modern CPUs that disables execution of user space addresses in kernel mode [86].

We also assume that the exploit uses ROP techniques due to the $W \oplus X$ property enforced in modern operating systems. This requires to know a sufficiently large amount of executable code in kernel space to enable ROP computations [81, 133]. Schwartz et al. showed that ROP payloads can be built automatically for 80% of Linux programs larger than 20 KB [130]. Further, we assume that the system fully supports ASLR and that no information leaks exist that can be exploited. Note that a variety of information leaks exist for typical operating systems [91], but these types of leaks stem from shortcomings and inconsequences in the actual implementation of the specific OS. Developers can fix these breaches by properly adjusting their implementation. Recently, Giuffrida et al. [72] argued that kernel information leakage vulnerabilities are hard to fix. While we agree that it is not trivial to do so, we show that even in the absence of any leak, we can still derandomize kernel space ASLR.

One of our attacks further requires that the userland process either has access to certain APIs or gains information about the physical frame mapping of at least one page in user space. However, since this prerequisite holds only for one single attack – which further turns out to be our least effective one – we do not consider it in the general attacker model but explain its details only in the corresponding Section 3.5.1.

In summary, we assume that the system correctly implements ASLR (i.e., the complete system is randomized and no information leaks exist) and that it enforces the $W \oplus X$ property. Hence, all typical exploitation strategies are thwarted by the implemented defense mechanisms.

3.4.2 General Approach

In this thesis, we present *generic* side channels against processors for the Intel ISA that enable a restricted attacker to deduce information about the privileged address space by timing certain operations. Such side channels emerge from intricacies of the underlying hardware and the fact that parts of the hardware (such as caches and physical memory) are *shared* between both privileged and non-privileged code. Note that all the approaches that we present in this thesis are *independent* of the underlying operating system: while we tested our approach mainly on Windows 7 and Linux, we are confident that the attacks also apply for other versions of Windows or even other operating systems. Furthermore, our attacks work on both 32- and 64-bit systems.

The methodology behind our timing measurements can be generalized in the following way: At first, we attempt to set the system in a specific state from user mode. Then we measure the duration of a certain memory access operation. The time span of this operation then (possibly) reveals certain information about the kernel space layout. Our timing side channel attacks can be split into two categories:

- **L1/L2/L3-based Tests:** These tests focus on the L1/L2/L3 CPU caches and the time needed for fetching data and code from memory.
- **TLB-based Tests:** These tests focus on TLB and PS caches and the time needed for address translation.

To illustrate the approach, consider the following example: we make sure that a privileged code portion (such as the operating system's system call handler) is present within the caches by executing a system call. Then, we access a designated set of user space addresses and execute the system call again. If the system call takes longer than expected, then the access of user space addresses has evicted the system call handler code from the caches. Due to the structure of modern CPU caches, this reveals parts of the physical (and possibly virtual) address of the system call handler code as we show in our experiments.

Accessing Privileged Memory

As explained in Section 3.2.2, different caching mechanisms determine the duration of a memory access:

- The TLB and PS caches speed up the translation from the virtual to the physical address.
- In case no TLB exists, the PS entries of the memory address must be fetched during the page walk. If any of these entries are present in the normal L1/L2/L3 caches, then the page walk is accelerated in a significant (i.e., measurable) way.
- After the address translation, the actual memory access is faster if the target data/code can be fetched from the L1/L2/L3 caches rather than from the RAM.

While it is impossible to access kernel space memory directly from user mode, the nature of the cache facilities still enables an attacker to *indirectly* measure certain side-effects. More precisely, she can directly access a kernel space address from user mode and measure the duration of the induced exception. The page fault will be faster if a TLB entry for the corresponding page was present. Additionally, even if a permission error occurs, this still allows to launch address translations and, hence, generate valid TLB entries by accessing privileged kernel space memory from user mode.

Further, an attacker can (to a certain degree) control which code or data regions are accessed in kernel mode by forcing fixed execution paths and known data access patterns in the kernel. For example, user mode code can perform a system call (`sysenter`) or an interrupt (`int`). This will force the CPU to cache the associated handler code and data structures (e.g., IDT table) as well as data accessed by the handler code (e.g., system call table). A similar effect can be achieved to cache driver code and data by indirectly invoking driver routines from user mode. Drivers oftentimes provide interfaces to user applications by creating special device files. If an application reads, writes, or sends device I/O control codes to such a file, then callback functions inside the driver to handle the request are triggered. This can, for example, be used to read or write network data to the TCP/IP system driver. Accordingly, a user mode application can craft special events to device files and make sure that code and data from specific drivers are present in the caches upon return to user mode.

Note that the x86/x64 instruction set also contains a number of instructions for explicit cache control (e.g., `invlpg`, `invd/wbinvd`, `clflush`, or `prefetch`) [84]. However, these instructions are either privileged and thus cannot be called from user mode, or they cannot be used with kernel space addresses from user mode.

Hence, none of these instructions can be used for our purposes. As a result, we must rely on *indirect* methods as explained in the previous paragraphs.

3.4.3 Handling Noise

While performing our timing measurements we have to deal with different kinds of noise that diminish the quality of our data if not addressed properly. Some of this noise is caused by the architectural peculiarities of modern CPUs [109]: to reach a high parallelism and work load, CPU developers came up with many different performance optimizations like hardware prefetching, speculative execution, multi-core architectures, or branch prediction. We have adapted our measuring code to take the effects of these optimizations into account. For example, we do not test the memory in consecutive order to avoid being influenced by memory prefetching. Instead, we use access patterns that are not influenced by these mechanisms at all. Furthermore, we have to deal with the fact that our tool is not the only running process and there may be a high CPU load in the observed system. The thread scheduler of the underlying operating system periodically and, if required, also preemptively interrupts our code and switches the execution context. If we are further running inside a virtual machine, there is even more context switching when a transition between the virtual machine monitor and the VM (or between different VMs) takes place. Finally, since all executed code is operating on the same hardware, also the caches have to be shared to some extent.

As mentioned above, our approach is based on two key operations: (a) set the system into a specific state and (b) measure the duration of a certain memory access operation. Further, these two operations are performed for each single memory address that is probed. Finally, the complete experiment is repeated multiple times until consistent values have been collected. While it is now possible — and highly probable — that our code is interrupted many times while probing the complete memory, it is also very likely that the low-level two step test operations can be executed without interruption. The mean duration of these two steps depends on the testing method we perform, but even in the worst case it takes no more than 5,000 clock cycles. Since modern operating systems have time slices of at least several milliseconds [1, 107], it is highly unlikely that the scheduler interferes with our measurements. Accordingly, while there may be much noise due to permanent interruption of our experiments, after a few iterations we will eventually be able to test each single memory address without interruption. This is sufficient since we only need minimal measurement values, i.e., we only need one measurement without interruption.

3.5 Implementation and Results

We now describe three different implementations of timing side channel attacks that can be applied independently from each other. The goal of each attack is to precisely locate some of the currently loaded kernel modules from user mode by measuring the time needed for certain memory accesses. Note that an attacker can already perform a ROP-based attack once she has derandomized the location of a few kernel modules or the kernel [81, 130].

The first attack is based on the concept of *cache probing*, where we try to evict the code/data of a loaded kernel module from the L3 caches in a controlled way. We achieve this by consecutively accessing certain user-controlled addresses that are mapped into the same cacheset where we assume the loaded kernel code. A practical obstacle is that the latest Intel CPUs based on the *Sandybridge* architecture do not use a single L3, but uniformly distribute the data to their different L3 *cache slices*. In order to apply this method to these CPUs, we thus first had to reverse-engineer the hash function used by the CPU in a black-box fashion since it is (to the best of our knowledge) not publicly documented.

In the following two experiments, we do not measure the time for code/data fetching, but we focus on the time needed for address translation. Depending on the used system, several caching facilities are also involved in mapping virtual to physical addresses, and these can be utilized for our means as well. Our second attack induces a *double page fault*, which allows us to obtain a complete picture of what ranges of kernel memory are allocated and which not. By examining the portions and sizes of neighboring allocations, we are able to locate certain drivers for which we have previously built *allocation signatures*.

In the third attack we again indirectly load kernel code into the caches by issuing a `sysenter` instruction. In contrast to our first method, we do not evict that loaded code from the L3, but instead we determine its virtual address directly. To that end, we execute kernel space memory at various locations, which obviously results in page faults. By measuring the time elapsed between calling into kernel space and return of the page fault we can infer if the address translation information for that very address has been cached before.

Depending on the randomness created by the underlying ASLR implementation, the first attack might still require partial information on the location for the kernel area. For the Windows ASLR implementation (see Section 3.2.1), this is not the case since only 64 slots are possible of the kernel. The first attack requires either the presence of two large pages or the knowledge of the physical address of a single page in user space. Our second attack has no requirements. However, due to the way the AMD CPU that we used during testing behaves in certain situations, this attack could not be mounted on this specific CPU. The third attack has no requirements at all.

Method	Requirements	Results	CPUs	Success
Cache Probing	*large pages* or PA of *eviction buffer*, partial information about *kernel_region* location	`ntoskrnl.exe` and `hal.sys`	all	✓
Double Page Fault	none	allocation map, several drivers	all but AMD	✓
Cache Preloading	none	`win32k.sys`	all	✓

Table 3.1: Summary of timing side channel attacks against kernel space ASLR on Windows.

We have evaluated our implementation on the 32-bit and 64-bit versions of *Windows 7 Enterprise* and *Ubuntu Desktop 11.10* on the following (native and virtual) hardware architectures to ensure that they are commonly applicable on a variety of platforms:

1. Intel i7-870 (Nehalem/Bloomfield, Quad-Core)
2. Intel i7-950 (Nehalem/Lynnfield, Quad-Core)
3. Intel i7-2600 (Sandybridge, Quad-Core)
4. AMD Athlon II X3 455 (Triple-Core)
5. VMWare Player 4.0.2 on Intel i7-870 (with VT-x)

Table 3.1 provides a high-level overview of our methods, their requirements, and the obtained results. We implemented an exploit for each of the three attacks.

For the sake of simplicity, all numbers presented in the remainder of this section were taken using Windows 7 Enterprise 32-bit. Note that we also performed these tests on Windows 7 64-bit and Ubuntu Desktop (32-bit and 64-bit) and can confirm that they work likewise. The Ubuntu version we used did not employ kernel space ASLR yet, but we were able to determine the location of the kernel image from user space. In general, this does not make any difference since the attacks also would have worked in the presence of kernel space ASLR.

In the following subsections, we explain the attacks and discuss our evaluation results.

3.5.1 First Attack: Cache Probing

Our first method is based on the fact that multiple memory addresses have to be mapped into the same cache set and, thus, compete for available slots. This can

be utilized to infer (parts of) virtual or physical addresses indirectly by trying to evict them from the caches in a controlled manner. More specifically, our method is based on the following steps: first, the searched code or data is loaded into the cache indirectly (e.g., by issuing an interrupt or calling `sysenter`). Then certain parts of the cache are consecutively replaced by accessing corresponding addresses from a user-controlled *eviction buffer*, for which the addresses are known. After each replacement, the access time to the searched kernel address is measured, for example by issuing the system call again. Once the measured time is significantly higher, one can be sure that the previously accessed eviction addresses were mapped into the same cache set. Since the addresses of these *colliding locations* are known, the corresponding cache index can be obtained and obviously this is also a part of the searched address.

For illustration assume a traditional L3 cache with 8 MB, 64 byte cachelines and a 16-way set associativity. These specifications result in 8*1024*1024/(64*16)=8192 different sets, which means that bits 18 to 6 of each physical addresses utilize as the cache index (bit 5 to 0 are used as the block offset to indicate the selected byte within each cacheline). For probing we choose a certain system service in the Windows kernel, which can be executed by calling the `sysenter` instruction with a particular value loaded into the `eax` register. The probing now works in the following way:

for $i = 0$ **to** 8191 **do**
(1) perform system call
(2) access 16 memory locations that are mapped to $cacheset_i$ (in order to replace the interrupt handler from the cache)
(3) perform system call again and measure required time
(4) $result_i \leftarrow i$ with highest execution time
end for

Several obstacles have to be addressed when performing these timing measurements in practice. First, the correct kind of memory access has to be performed: higher cache levels are unified (i.e., there are no separate data and instruction caches), but on lower levels either a memory read/write access or an execution has to be used in order to affect the correct cache type. Second, accessing the colliding addresses only once is not enough. Due to the Pseudo-LRU algorithm it may happen that not the searched address is evicted, but one from the eviction buffer. Therefore, it is necessary to access each of the colliding addresses twice. Note that it is still possible that code within another thread or on other CPUs concurrently accesses the search address in the meantime, setting its reference bit that way. To overcome this problem, all tests have to be performed several times to reduce the influence of potential measuring errors and concurrency.

More serious problems arise due to the fact that the cache indexes on higher levels

are taken from the physical instead of the virtual addresses. In our experiments, the eviction buffer is allocated from user mode and, hence, only its virtual address is known. While it is still possible to locate the colliding cacheset, no information can be gathered about the corresponding physical addresses. In general, even if the physical address of the searched kernel location is known, this offers no knowledge about its corresponding virtual address. However, the relevant parts of the virtual and physical address are identical for the *kernel_region* of Windows (see Section 3.2.1). Hence, all the relevant bits of the virtual address can be obtained from the physical address.

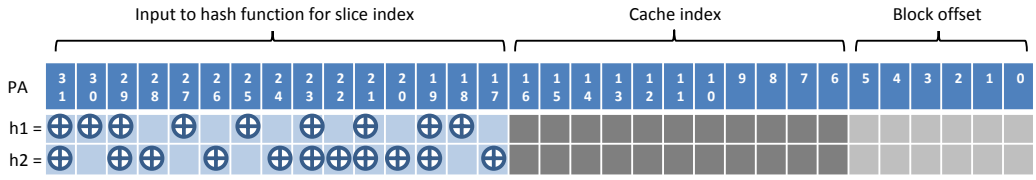
Cache probing with the latest Intel CPUs based on the Sandybridge [84] architecture is significantly harder, even if the attacker has a contiguous region of memory for which all corresponding physical addresses are known. These processors employ a *distributed last level cache* [84] that is split into equally sized *cache slices* and each of them is dedicated to one CPU core. This approach increases the access bandwidth since several L3 cache accesses can be performed in parallel. In order to uniformly distribute the accesses to all different cache slices, a hash function is used that is not publicly documented. We thus had to reconstruct this hash function in a black-box manner before cache probing can be performed, since otherwise it is unknown which (physical) addresses are mapped into which cache location. We explain our reverse-engineering approach and the results in a side note before explaining the actual evaluation results for our first attack.

Side Note: Sandybridge Hash Function

In order to reconstruct the Sandybridge hash function, we utilized the Intel i7-2600 processor. This CPU has an 8 MB L3 cache and 4 cores, resulting in 2 MB L3 slices each. Hence, the hash function has to *decide* between 4 different slices (i.e., resulting in 2 output bits). Since our testing hardware had 4 GB of physical memory, we have reconstructed the hash function for an input of 32 bits. In case of larger physical memory, the same method can be applied to reverse the influence of the upper bits as well.

We started with the reasonable assumption that L3 cachelines on this CPU still consist of 64 bytes. Hence, the lowest 6 bits of each address operate as an offset and, therefore, do not contribute as input to the hash function. Accordingly, we assumed a function $h : \{0,1\}^{32-6} \rightarrow \{0,1\}^2$.

In order to learn the relationship between the physical addresses and the resulting cache slices, we took one arbitrary memory location and an additional eviction buffer of 8 MB and tried to determine the colliding addresses within (i.e., those which are mapped into the same cacheset of the same cache slice). Since the L3 cache operates on physical addresses, the eviction buffer had to be contiguous. Therefore, we used our own custom driver for this experiment.



$$h = (h_1, h_2)$$
$$h_1 = b_{31} \oplus b_{30} \oplus b_{29} \oplus b_{27} \oplus b_{25} \oplus b_{23} \oplus b_{21} \oplus b_{19} \oplus b_{18}$$
$$h_2 = b_{31} \oplus b_{29} \oplus b_{28} \oplus b_{26} \oplus b_{24} \oplus b_{23} \oplus b_{22} \oplus b_{21} \oplus b_{20} \oplus b_{19} \oplus b_{17}$$

Figure 3.6: Results for the reconstruction of the undocumented Sandybridge hash function.

Performance optimization features of modern CPUs like hardware prefetching, speculative execution, and branch prediction make it impossible to directly identify single colliding addresses. Therefore, we performed a two-step experiment: (1) we identified eviction buffer *regions* of adjacent memory addresses that collide with the probe address and then (2) we located the particular colliding addresses within these regions. We have performed these tests several hundred times with different physical addresses in order to gain variety in the test data. As a result of each single test we got a tuple $(p, CA = \{ca_1, ca_2, ...\})$ whereas p is the used probe address and each ca_i is a colliding address from our eviction buffer. By manually comparing those tuples (p, CA) and (p', CA') with a hamming distance of *one* between p and p', we were able to learn the influence of particular bits on the colliding addresses from CA and CA'.

In the end we were able to fully reconstruct the hashing function h that decides which cache slice is used for a given address. It turned out that only the bits 31 to 17 are considered as input values. Each cache slice operates as a separate smaller 2 MB cache, whereas the address bits 16 to 6 constitute as the cache index (11 bits are necessary to address all sets of such a 2 MB cache). Figure 3.6 shows how the 32 bits of a physical address are used as inputs to the hash function, cache index, and block offset.

Evaluation Results

We evaluated cache probing on all of our testing machines. We assume that the base address of the *kernel_region* (see *kernel_base* from Section 3.2.1) is known. This is a reasonable assumption in practice since this information can be reliably extracted using the method presented in Section 3.5.2. In Windows this address actually is constant for a particular system.

Figure 3.7 shows the correlation of the different parts of the virtual and physical address inside the *kernel_region*. In essence, bits 16 to 12 of the kernel's base

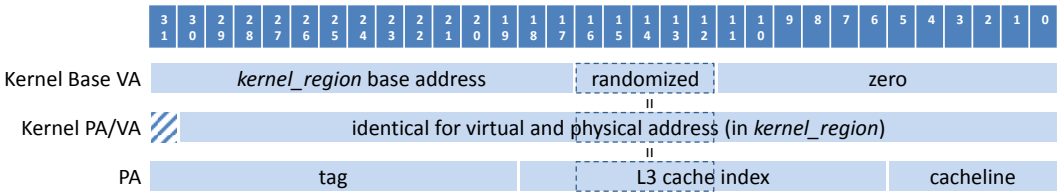


Figure 3.7: Correlation of different memory addresses.

address are randomized in Windows' ASLR implementation and must be known by an attacker. Since the PA and VA for bits 30 to 0 are identical in the *kernel_region*, it is also sufficient to know bits 16 to 12 of the physical address. This bit range overlaps with the L3 cache index. In other words: if the L3 cache index is known, then an attacker can tell the virtual base address of the kernel.

We used cache probing to extract parts of the physical address of the system call handler `KiFastCallEntry`. The offset from this function to the kernel's base address is static and known. If we know the address of this function, then we also know the base address of the kernel (and HAL).

We performed the following steps for all cache sets i:

1. Execute `sysenter` with an unused syscall number.
2. Evict cache set i using the eviction buffer.
3. Execute `sysenter` again and measure the duration.

The unused syscall number minimizes the amount of executed kernel mode code since it causes the syscall handler to immediately return to user mode with an error. Step 1 makes sure that the syscall handler is present in the caches. Step 2 tries to evict the syscall handler code from the cache. Step 3 measures if the eviction was successful. If we hit the correct set i, then the second `sysenter` takes considerably longer and from i we can deduce the lower parts of the physical address of the syscall handler. Along with the address of the *kernel_region*, this yields the *complete virtual address* of the syscall handler, and thus the base of the entire kernel and the HAL.

We performed extensive tests on the machine powered by an Intel i7-870 (Bloomfield) processor. We executed the cache probing attack 180 times; the machine was rebooted after each test and we waited for a random amount of time before the measurements took place to let the system create artificial noise. Figure 3.8 shows the cache probing measurements. The x-axis consists of the different L3 cache sets ($8,192$ in total) and the y-axis is the duration of the second system call handler invocation in CPU clocks, after the corresponding cache set was evicted. The vertical dashed line indicates the correct value where the system call handler code resides. There is a clear cluster of high values at this dashed line, which can be used to extract the correct cache set index and thus parts of the physical (and possibly

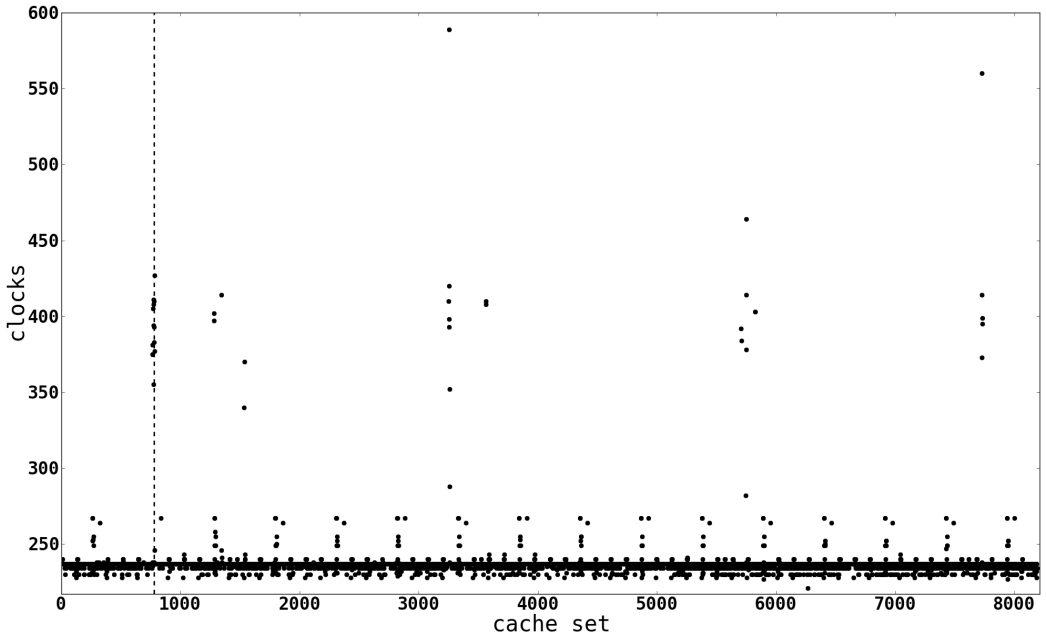


Figure 3.8: Cache probing results for Intel i7-870 (Bloomfield).

virtual) address. We were able to successfully determine the correct syscall handler address in each run and there were no false positives. The test is fast and generally takes less than one second to finish.

Discussion

For successful cache probing attacks, an adversary needs to know the physical addresses of the eviction buffer, at least those bits that specify the cache set. Furthermore, she somehow has to find out the corresponding virtual address of the kernel module from its physical one. This problem is currently solved by using *large pages* for the buffer, since under Windows those always have the lowest bits set to 0. Therefore, their first byte always has a cache index of 0 and all following ones can be calculated from that. However, this method does not work with Sandybridge processors, since there we need to know the complete physical address as input to the hash function that decides on which cache slice an address is mapped. Furthermore, allocating large pages requires a special right under Windows (`MEM_LARGE_PAGES`), which first has to be acquired somehow. One possible way to overcome this problem is to exploit an application that already possesses this right.

In case of non-Sandybridge processors, large pages are not needed per se. It is only necessary to know the physical start address of the eviction buffer. More

generically, it is only necessary to know parts of the physical base address of *one* user space address, since this can then be used to align the eviction buffer. Our experiments have shown that these parts of the physical base address of the common module `ntdll`, which is always mapped to user space, is always constant (even after reboots). Though the concrete address varies depending on the hardware and loaded drivers and is thus difficult to compute, the value is deterministic.

3.5.2 Second Attack: Double Page Fault

The second attack allows us to reconstruct the allocation of the entire kernel space from user mode. To achieve this goal, we take advantage of the behavior of the TLB cache. When we refer to an *allocated* page, we mean a page that can be accessed without producing an address translation failure in the MMU; this also implies that the page must not be paged-out.

The TLB typically works in the following way: whenever a memory access results in a successful page walk due to a TLB miss, the MMU replaces an existing TLB entry with the translation result. Accesses to non-allocated virtual pages (i.e., the present bit in the PDE or PTE is set to zero) induce a page fault and the MMU does *not* create a TLB entry. However, when the page translation was successful, but the *access permission* check fails (e.g., when kernel space is accessed from user mode), a TLB entry is indeed created. Note that we observed this behavior only on Intel CPUs and within the virtual machine. In contrast, the AMD test machine acted differently and *never* created a TLB entry in the mentioned case. The double page fault method can thus not be applied on our AMD CPU.

The behavior on Intel CPUs can be exploited to reconstruct the entire kernel space from user mode as follows: for each kernel space page p, we first access p from user mode. This results in a page fault that is handled by the operating system and forwarded to the exception handler of the process. One of the following two cases can arise:

- p refers to an *allocated* page: since the translation is successful, the MMU creates a TLB entry for p although the succeeding permission check fails.
- p refers to an *unallocated* page: since the translation fails, the MMU does *not* create a TLB entry for p.

Directly after the first page fault, we access p again and measure the time duration until this second page fault is delivered to the process's exception handler. Consequently, if p refers to an allocated kernel page, then the page fault will be delivered *faster* due to the inherent TLB hit.

Due to the many performance optimizations of modern CPUs and the concurrency related to multiple cores, a single measurement can contain noise and outliers. We thus probe the kernel space multiple times and only use the observed minimal access time for each page to reduce measurement inaccuracies. Figure 3.9 shows

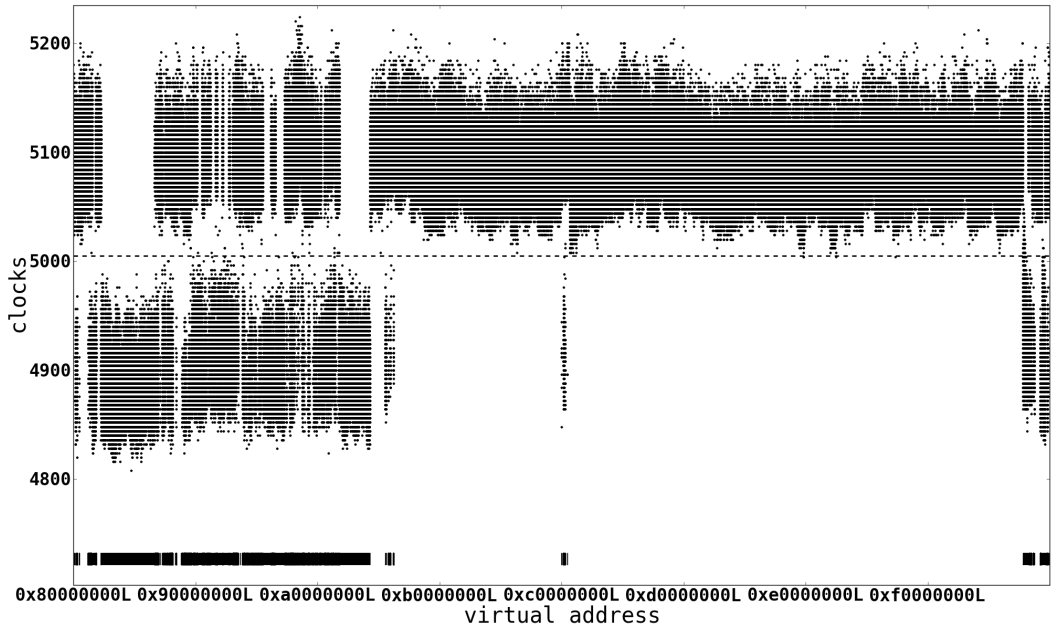


Figure 3.9: Example of double page fault measurements for an Intel i7-950 (Lynnfield) CPU.

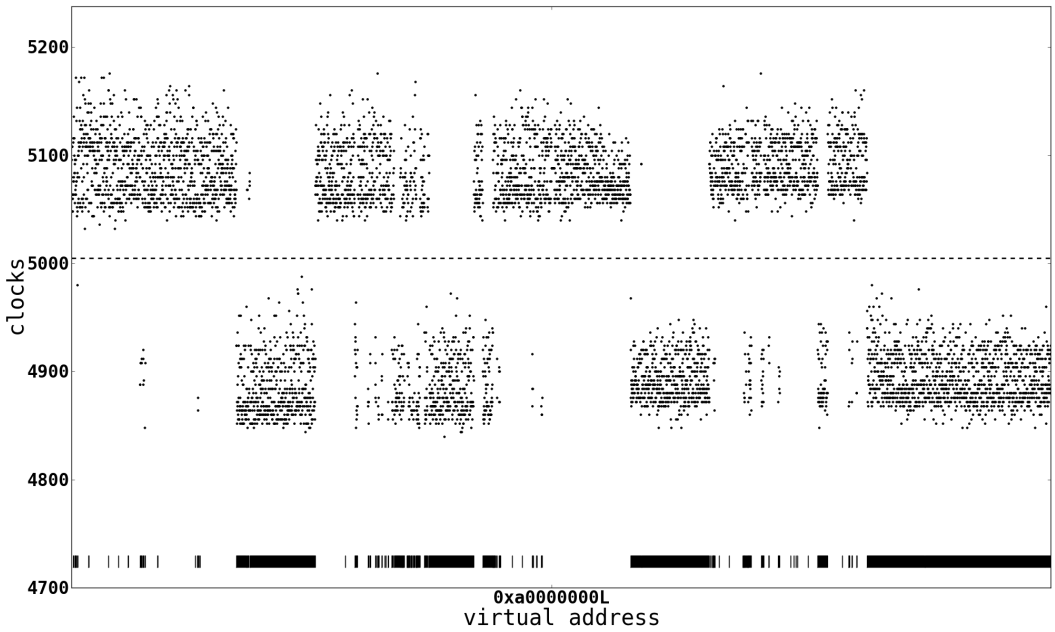


Figure 3.10: Zoomed-in view around `0xa0000000` (Intel i7-950 (Lynnfield) CPU).

measurement results on an Intel i7-950 (Lynnfield) CPU for eight measurements, which we found empirically to yield precise results. The dots show the minimal value (in CPU clocks) observed on eight runs. The line at the bottom indicates which pages are actually allocated in kernel space; a black bar means the page is allocated. As one can see, there is a clear correlation between the timing values and the allocation that allows us to infer the kernel memory space. Figure 3.10 shows a more detailred zoomed-in version around the memory address `0xa0000000`.

We developed an algorithm that reconstructs the allocation from the timing values. In the simplest case, we can introduce a threshold value that differentiates allocated from unallocated pages. In the above example, we can classify all timing values below $5,005$ clocks as allocated and all other values as unallocated as indicated by the dashed line. This yields a high percentage of correct classifications. Depending on the actual CPU model, this approach might induce insufficient results due to inevitable overlap of timing values and thus other reconstruction algorithms are necessary. We implemented a second approach that aims at detecting transitions from allocated to unallocated memory by looking at the pitch of the timing curve, a straightforward implementation of a *change point detection* (CPD) algorithm [27].

Figure 3.11 shows the double page fault measurements on an Intel i7-870 (Bloomfield) processor. It is not possible to use a simple threshold value to tell apart allocated from unallocated pages without introducing a large amount of faulty results. In the zoomed version in Figure 3.12, one can see that it is still possible to distinguish unallocated from unallocated pages. Note that this figure uses lines instead of dots to stress the focus on transitions from high to low values (or vice versa). We therefore use a change point detection (CPD) algorithm [27] in this case.

Figure 3.13 shows our CPD algorithm in pseudo code. The *kernelspace_pages* array holds one address for each kernel space page. The *timing_vector* array returns a list of the timing measurements for a specific address. If there are x iterations, then the list is x elements long and contains the values for each iteration. The value of the *threshold* variable has to be adjusted for each CPU model, but this can be done by an attacker beforehand. It is also possible to automatically determine this value form user mode.

Evaluation Results

We evaluated our double page fault based approach on the three Intel CPUs and the virtual machine, Table 2 shows a summary of the results. We employed the threshold algorithm on CPU (1) and the CPD algorithm on platforms (2)–(4). The numbers shown in the table are the average out of ten runs for each machine. Between each run, we rebooted the operating system to make sure that the kernel space allocation varies. We took a snapshot of the allocation with the help of a custom driver *before* we started the measurements to obtain a ground truth of the memory layout. Since

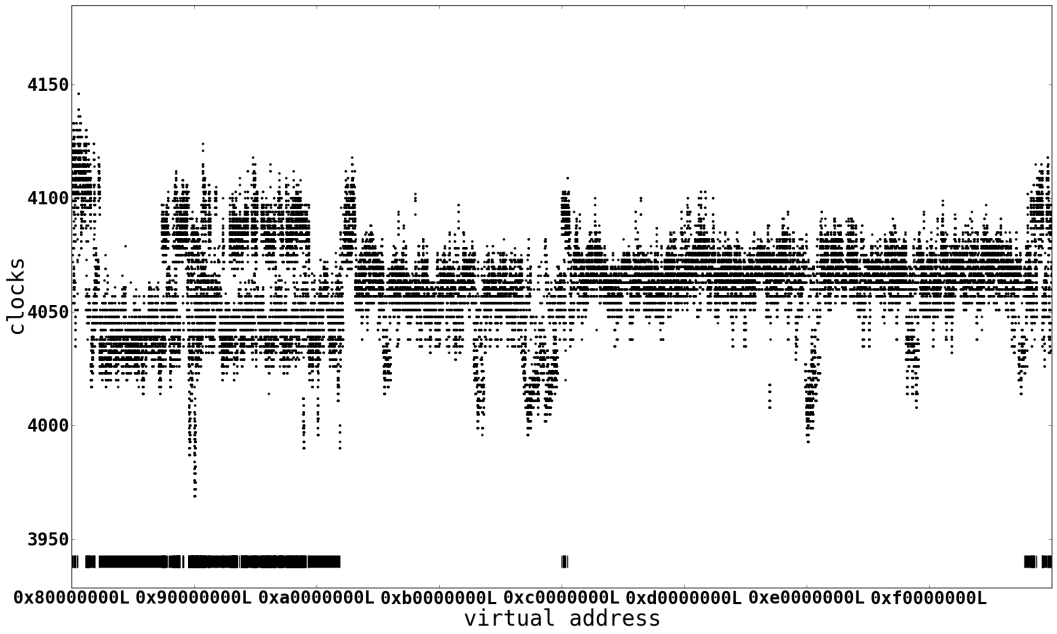


Figure 3.11: Double page fault measurements on Intel i7-870 (Bloomfield) processor.

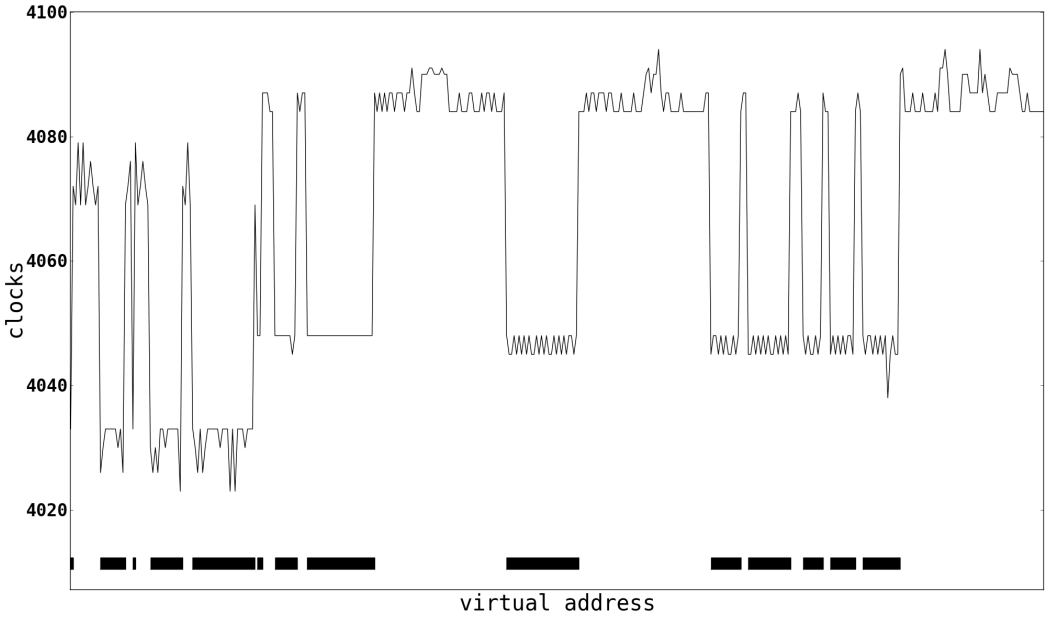


Figure 3.12: Zoomed-in view of Figure 3.11.

```
alloc ← []
last_vec ← []
for all addr in kernelspace_pages do
  vec ← timing_vector[addr]
  diffs ← []
  for all i in 0 to len(vec) do
    diffs ← diffs ∪ (vec[i] − last_vec[i])
  end for
  diffs ← sort(diffs)
  cut ← len(diffs)/4
  diffs ← diffs[cut ... len(diffs) − cut]
  average ← sum(diffs)/len(diffs)
  if average < −threshold then
    alloc ← allocation ∪ true
  else
    if average > threshold then
      alloc ← alloc ∪ false
    else
      alloc ← alloc ∪ alloc[len(alloc) − 1]
    end if
  end if
  last_vec ← vec
end for
```

Figure 3.13: Pseudo code of our CPD reconstruction algorithm.

CPU model	Correctness	Runtime
(1) i7-870 (Bloomfield)	96.42%	17.27 sec (8 it.)
(2) i7-950 (Lynnfield)	99.69%	18.36 sec (8 it.)
(3) i7-2600 (Sandybr.)	94.92%	65.41 sec (32 it.)
(4) VMware on (1)	94.98%	72.93 sec (32 it.)

Table 3.2: Results for double page fault timings.

the allocation might change while the measurements are running, the correctness slightly decreases because of this effect. Nevertheless, we were able to successfully reconstruct the state of at least 94.92% of all pages in the kernel space on each machine. With the help of *memory allocation signatures* (a concept we introduce next) such a precision is easily enough to exactly reconstruct the location of many kernel components.

The average runtime of the measurements varies between 18 and 73 seconds and is therefore within reasonable bounds. One iteration is one probe of the entire kernel space with one access per page. As noted above, we empirically found that more than eight iterations on Nehalem CPUs do not produce better results. For Sandybridge and VMware, more iterations yielded more precise results, mainly due to the fact that there was more noise in the timings.

Memory Allocation Signatures

The double page fault timings yield an estimation for the allocation map of the kernel space, but do not determine at which concrete base addresses the kernel and drivers are loaded to. However, the allocation map can be used, for example, to spot the *kernel_region* (i.e., the memory area in which the kernel and HAL are loaded) due to the large size of this region, which can be detected easily.

One could argue that, since the virtual size of each driver is known, one could find driver load addresses by searching for allocated regions which are exactly as big as the driver image. This does not work for two reasons: first, Windows kernel space ASLR appends most drivers in specific memory regions and thus there is usually no gap between two drivers (see Section 3.2.1). Second, there are gaps of unallocated pages inside the driver images as we explain next.

In contrast to the *kernel_region*, Windows drivers are not mapped using large pages but using the standard 4 KB page granularity. Code and data regions of drivers are unpageable by default. However, it is possible for developers to mark certain sections inside the driver as pageable to reduce the memory usage of the driver. Furthermore, drivers typically have a discardable `INIT` section, that contains the initialization code of the driver which only needs to be executed once. All code pages in the `INIT` section are freed by Windows after the driver is initialized. Code

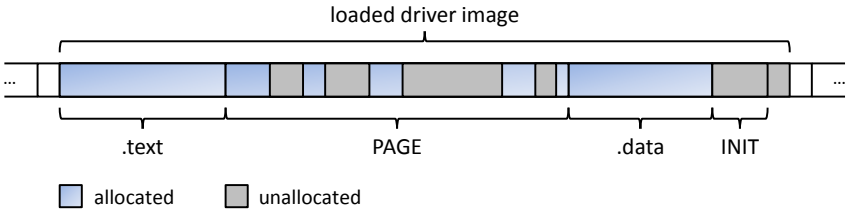


Figure 3.14: Example of an allocation signature.

CPU model	Matches	Code size
(1) i7-870 (Bloomfield)	21	7,431 KB
(2) i7-950 (Lynnfield)	9	4,184 KB
(3) i7-2600 (Sandybridge)	5	1,696 KB
(4) VMware on (1)	18	7,079 KB
(1) with signatures of (2)	9	2,312 KB

Table 3.3: Evaluation of allocation signature matching.

or data in pageable sections that are never or rarely used are likely to be unallocated most of the time. Along with the size and location of the `INIT` section, this creates a *memory allocation signature* for each driver in the system. We can search for these signatures in the reconstructed kernel space allocation map to determine the actual load addresses of a variety of drivers. Figure 3.14 shows an example of a driver allocation signature.

We evaluated the signature matching on all three Intel CPUs and the virtual machine. At first, we took a snapshot of the kernel space with the help of a custom driver. Then we created signatures for each loaded driver. A signature essentially consists of a vector of boolean values that tell whether a page in the driver was allocated (true) or paged-out (false). Note that this signature generation step can be done by an attacker in advance to build a database of memory allocation signatures.

In the next step, we rebooted the machine, applied the double page fault approach, and then matched the signatures against the reconstructed kernel space allocation map. To enhance the precision during the signature matching phase, we performed two optimizations: first, we rule out signatures that contain less than five transitions from allocated to paged-out memory to avoid false positives. Second, we require a match of at least 96% for a signature, which we empirically found to yield the best results.

The results are shown in Table 3. On machine (1), the signature matching returns the exact load addresses of 21 drivers (including big common drivers such as `win32k.sys` and `ndis.sys`); 141 drivers are loaded in total and 119 signatures were ruled out because they held too few transitions. Hence only one signature had a

too low match ratio. All identified base addresses are correct, there are no false positives. Most of the other drivers could not be located since they are too small and their signatures thus might produce false positives. The 21 located drivers hold 7,431 KB of code, which is easily enough to mount a full ROP attack as explained previously [81, 130]. Similar results hold for the other CPUs.

To assess whether the signatures are also portable across different CPUs, we took the signatures generated on machine (2) and applied them to machine (1). The operating system and driver versions of both machines are identical. This yields 9 hits with 2,312 KB of code. This experiment shows that the different paging behavior in drivers is not fundamentally affected by differing hardware configurations.

Discussion

Although the double page fault measurements only reveal which pages are allocated and which are not, this still can be used to derive precise base addresses as we have shown by using the memory allocation signature matching. Furthermore, the method can be used to find large page regions (especially the *kernel_region*).

3.5.3 Third Attack: Address Translation Cache Preloading

In the previous section we have described an approach to reconstruct the allocation map of the complete kernel space. While it is often possible to infer the location of certain drivers from that, without driver signatures it only offers information about the fact that there is *something* located at a certain memory address and not *what*. However, if we want to locate a certain driver (i.e., obtain the virtual address of some piece of code or data from its loaded image), we can achieve this with our third implementation approach: first we flush all caches (i.e., address translation and instruction/data caches) to start with a *clean* state. After that, we preload the address translation caches by indirectly calling into kernel code, for example by issuing a `sysenter` operation. Finally, we intentionally generate a page fault by jumping to some kernel space address and measure the time that elapses between the jump and the return of the page fault handler. If the faulting address lies in the same memory range as the preloaded kernel memory, a *shorter* time will elapse due to the already cached address translation information.

Flushing all caches from user mode cannot be done directly since the `invlpg` and `invd/wbinvd` are privileged instructions. Thus, this has to be done indirectly by accessing sufficiently many memory addresses to evict all other data from the cache facilities. This is trivial for flushing the address translation and L1 caches, since only a sufficient number of virtual memory addresses have to be accessed. However, this approach is not suitable for L2/L3 caches, since these are *physically* indexed and we do not have any information about physical addresses from user mode. Anyway, in

practice the same approach as described above works if the eviction buffer is chosen large enough. We have verified for Windows operating systems that large parts of the physical address bits of consecutively allocated pages are in successive order as well. Presumably this is done for performance reasons to optimally distribute the data over the caches and increase the effectiveness of the hardware prefetcher. As our experiments have shown, even on Sandybrige CPUs one virtually consecutive memory buffer with a size twice as large as the L3 cache is sufficient to completely flush it.

During our experiments we tried to locate certain system service handler functions within `win32k.sys`. To avoid cache pollution and obtain the best measuring results, we chose the system service `bInitRedirDev`, since it only executes 4 bytes of code before returning. As a side effect, we also located the *System Service Dispatch/Parameter Tables* (SSDT and SSPT) within that module, since these tables are accessed internally on each service call.

In our implementation we first allocated a 16 MB eviction buffer and filled it with `RET` instructions. Then for each page p of the complete kernel space memory (or a set of selected candidate regions), we performed three steps:

1. Flush all (address translation-, code- and unified) caches by calling into each cacheline (each 64th byte) of the eviction buffer.
2. Perform `sysenter` to preload address translation caches.
3. Call into some arbitrary address of page p and measure time until page fault handler returns.

Evaluation Results

The steps described above have to be repeated several times to diminish the effects of noise and measuring inaccuracies. It turned out that the necessary amount of iterations strongly depends on the underlying hardware. Empirically we determined that around 100 iterations are needed on Nehalem, 60 on AMD, and only 30 on Sandybridge to reliably produce precise results. Inside the virtual machine, we had to further increase the number of iterations due to the noise that was generated by the virtual machine monitor. Nevertheless, by increasing it to 100 (the VM operated on the Sandybridge processor) this timing technique also worked successfully inside a virtualized environment.

We learned that the noise could be additionally reduced by taking different addresses randomly from each probed page for each iteration. In addition, we found out that using relative time differences was less error-prone than using absolute values. Therefore, we enhanced our testing procedure by performing the measuring twice for each page: the first time like shown above and the second time *without* performing the syscall in between. By calculating the relative time difference between

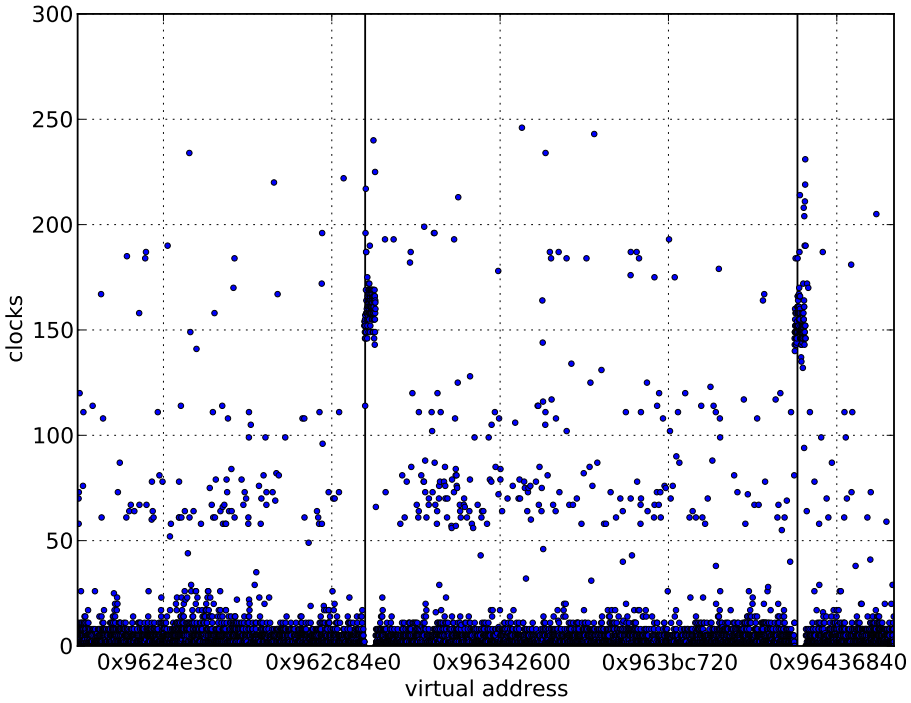


Figure 3.15: Extract of cache preloading measurements.

both timing values, we were able to measure the speedup of address translation caches for our particular scenario. Figure 3.15 shows an extract of our measuring results for the Intel i7-950 (Lynnfield) CPU. The x-axis displays the probed virtual address, while the y-axis displays the relative time difference in clock cycles. The two vertical lines indicate those locations where the searched system service handler function resp. the SSDT/SSPT were located. As one can easily see those memory regions have much higher timing difference values than the others. Though there was a lot of noise within the data, our algorithms were able to locate those regions correctly on all of our testing environments.

While this method only reveals the memory *page* of the searched kernel module, it is still possible to reconstruct its full virtual address. This can be achieved by obtaining the relative address offset of the probed code/data by inspecting the image file of the module. As the measuring operates on a *page granularity*, it is best suited to locate kernel modules that reside in regular pages. Nevertheless, with the described *difference* technique, also large page memory regions can be identified

that contain certain code or data. Obviously, the exact byte locations within such regions cannot be resolved and, therefore, we have used it to locate `win32k.sys` in our experiments. Due to its size, this module is sufficient to perform arbitrary ROP attacks [81, 130].

Discussion

Our third proposed method has no remarkable limitations. However, depending on the size of the probed memory range and the amount of necessary test iterations, it may take some time to complete. The probing of a 3 MB region (this is the size of `win32k.sys`) for *one* iteration takes around 27 seconds. Therefore, if an attacker has employed the *double page fault* method to identify an appropriate candidate region and then performs 30 iterations on a Sandybridge processor, it takes 13 minutes to perform the complete attack. However, since the relative offset of the searched kernel function can previously be obtained from the image file, the probed memory region can be reduced drastically, enabling to perform the test in a minute or less. If the location of candidate regions is not possible, our attack will still work but take longer time. Furthermore, the technique operates on page granularity. Hence, drivers residing in large pages can be located, but their exact byte offset cannot be identified without additional techniques.

3.6 Mitigation Approaches

Since the methods presented in the previous section can be used to break current ASLR implementations, mitigation strategies against our attacks are necessary. To that end, there are several options for CPU designers and OS vendors.

The root cause of our attacks is the *concurrent usage* of the same caching facilities by privileged and non-privileged code and data, i.e., the memory hierarchy is a shared resource. One solution to overcome this problem would be to split all caches and maintain isolated parts for user and kernel mode, respectively. Obviously, this imposes several performance drawbacks since additional checks had to be performed in several places and the maximum cache size would be cut in half for both separate caches (or the costs increase).

A related mitigation attempt is to forbid user mode code to resolve kernel mode addresses. One way to achieve this is to modify the *global descriptor table* (GDT), setting a *limit* value such that the segments used in non-privileged mode only span the user space. However, doing so would render some CPU optimization techniques useless that apply when the *flat memory model* is used (in which all segments span the complete memory). Furthermore, the complete disabling of segmentation on 64-bit architectures makes this mitigation impossible. Another option would be to suppress the creation of TLB entries on successful address translation if an access

violation happens, like it is done with the tested AMD CPU. Nevertheless, the indirect loading of kernel code, data, or address mappings through system calls still cannot be avoided with this method.

Current ASLR implementations (at least under Windows) do not fully randomize the address space, but randomly choose from 64 different memory slots. By utilizing the complete memory range and distributing all loaded modules to different places, it would be much harder to perform our attacks. Especially when dealing with a 64-bit memory layout, the time needed for measuring is several magnitudes higher and would increase the time needed to perform some of our attacks. Nevertheless, scattering allocated memory over the full address range would significantly degrade system performance since much more paging structures would be needed and spatial locality would be destroyed to a large extent. Furthermore, we expect that our *double page fault* attack even then remains practical. Due to the huge discrepancy between the 64-bit address space and the used physical memory, the page tables are very sparse (especially those one higher levels). Since page faults can be used to measure the depth of the valid page tables for a particular memory address, only a very small part of the complete address space actually has to be probed.

We have proposed a method to identify mapped kernel modules by comparing their memory allocation patterns to a set of known signatures. This is possible because parts of these modules are marked *pageable* or *discardable*. If no code or data could be paged-out (or even deallocated) after loading a driver, it would be impossible to detect them with our signature approach. Again, applying this protection would decrease the performance, because unpageable memory is a scarce and critical system resource.

One effective mitigation technique is to modify the execution time of the page fault handler: if there is no correlation between the current allocation state of a faulting memory address and the observable time for handling that, the timing side channel for address translation vanishes. This would hinder our attacks from Sections 3.5.2 and 3.5.3. We have implemented one possible implementation for this method and verified that our measuring no longer works. To that end, we have hooked the page fault handler and *normalized* its execution time if unprivileged code raises a memory access violation on kernel memory. In that case we enforce the execution to always return back to user mode after a *constant* amount of clock cycles. For that purpose we perform a bunch of timing tests in advance to measure the timing differences for memory accesses to unallocated and allocated (for both regular and large) pages. Inside the hooked page fault handler we delay execution for the appropriate amount of time, depending on the type of memory that caused the exception. Since this happens only for software errors – or due to active probing – there is no general impact on system performance. Note that modifying the page fault handler renders our attack infeasible, but there might be other side channels an attacker can exploit to learn more about the memory layout of the kernel.

Even with normalizing the page fault handling time, our *cache probing attack* remains feasible. However, cache probing has one fundamental shortcoming: it only reveals information about *physical* addresses. If the kernel space randomization is only applied to virtual addresses, then knowing physical addresses does not help in defeating ASLR.

The kernel (or an underlying hypervisor) may also try to detect suspicious access patterns from usermode to kernelspace, for example by limiting the amount of user-mode page faults for kernel space addresses. Such accesses are necessary for two of the previously described methods. While our current implementations of these attacks could be detected without much effort that way, we can introduce artificial sleep times and random access patterns to mimicry benign behavior. In the end, this would lead to an increased runtime of the exploits.

In case the attacks are mounted from within a VMM, the hypervisor might also provide the VM with incorrect information on the true CPU model and features, for example by modifying the `cpuid` return values. However, this might have undesirable side-effects on the guest operating system which also needs this information for optimizing cache usage. Furthermore, the architectural parameters of the cache (such as size, associativity, use of slice-hashing, etc.) can be easily determined from within the VM using specific tests.

Finally, the most intuitive solution would be to completely disable the `rdtsc` instruction for usermode code, since then no CPU timing values could be obtained at all. However, many usermode applications actually rely on this operation and, hence, its disabling would cause significant compatibility issues.

3.7 Conclusion and Future Work

In this chapter, we have discussed a generic, timing-based side channel attack against kernel space ASLR. Such side channels emerge from intricacies of the underlying hardware and the fact that parts of the hardware (such as caches and physical memory) are shared between both privileged and non-privileged mode. We have presented three different instances of this methodology that utilize timing measures to precisely infer the address locations of mapped kernel modules. We successfully tested our implementation on four different CPUs and within a virtual machine and conclude that these attacks are feasible in practice. As a result, a local, restricted attacker can infer valuable information about the kernel memory layout and bypass kernel space ASLR for proprietary Windows operating systems.

As part of our future work, we plan to apply our methods to other operating systems such as Mac OS X and more kinds of virtualization software. We expect that they will work without many adoptions since the root cause behind the attacks lies in the underlying hardware and not in the operating system. We further plan

to test our methods on other processor architectures (e.g., on ARM CPUs to attack ASLR on Android [34]). Again, we expect that timing side channel attacks are viable since the memory hierarchy is a shared resource on these architectures as well.

Another topic for future work is the identification of methods to obtain the physical address of a certain memory location from user mode. One promising method would be to identify certain data structures that are always mapped to the same physical memory and use the technique of cache probing with them. First experiments have shown that certain parts of mapped system modules are constant for a given system (e.g., the physical base address of `ntdll.dll`). Another possibility is to instrument the characteristics of the *Sandybridge* hash function to locate colliding memory locations and infer the bits of their physical address.

4

Providing Control Flow Integrity for Proprietary Mobile Devices

4.1 Introduction

Ever since the Morris worm emerged as the first large scale Internet security incident, researchers have been aware of the possible consequences that software vulnerabilities pose to Internet security. More than two decades later, software vulnerabilities still constitute a major security problem and are subject to intense research in the academic sector. As a consequence, various different defense mechanisms have been proposed and implemented that aim at mitigating vulnerability exploitation in a preferably generic way. By doing so, software vulnerabilities may still exist but they cannot be exploited because the targeted program is terminated before malicious attackers can make the system execute arbitrary operations.

Software exploits usually work by compromising the control flow of the vulnerable application at some stage of the execution. There exists a multitude of starting points for an attacker to divert the control flow. Traditionally, the first software exploits (such as the Morris worm) abused buffer overflows on either the stack or the heap [12, 13]. More advanced methods take advantage of errors in the handling of format string functions [71], integer overflows [32], or use-after-free bugs [10]. On a more abstract level, all these vulnerabilities lead to a diversion of the control flow in the targeted process to an attacker-controlled arbitrary memory location. This eventually allows the attacker to execute arbitrary computations in the context of the exploited process.

Despites massive efforts to come to grips with the problem of software vulnerabilities, researchers constantly reveal new holes in security critical applications. This is partly still caused by the prevalent use of unsafe low-level programming languages (such as C) that may encourage developers to unknowingly create unsafe code. However, low-level programming languages are still an essential part of pro-

gram development because of compatibility issues and since they emit significantly faster code than safe languages do.

One promising approach to mitigate software exploits makes use of the fact they, at some stage, inherently divert the application's control flow to an attacker-controlled memory location. This invalid transition can be regarded as a violation of the set of valid and allowed control flow transitions that can occur in the execution of a program. Frameworks that ensure control flow integrity (CFI) [2] can successfully detect and thwart control flow hijacking attacks and thus almost all kinds of software exploits. CFI is a generic approach and – if implemented correctly – constitutes an efficient protection. It is even possible to detect highly sophisticated attacks such as ROP exploits that bypass W $\oplus$ X based protection schemes. The approach is also flexible and more fine-grained than similar approaches such as ASLR since it works on the control flow transitions level within the protected program. Note that ASLR systems can also suffer from an insufficient entropy or from data leakage issues, as described in Chapter 3. The flexibility of the approach allows CFI frameworks to be used as the foundation for other analysis or protection-based schemes. In fact, we extended our framework to support the specification of individual API policy rules per application to greatly increase the user's privacy in an iOS system.

CFI frameworks must construct the so-called CFG in advance. This is performed statically once and yields a graph (usually one per subroutine) that contains all the *legitimate* code paths that the instruction pointer may follow during execution. The CFG is passed on to the dynamic CFI engine when the secured application is started, which then takes the role of a supervisor at runtime by comparing control flow transitions against the legitimate paths. If at some point a CFG violation is detected, the program is terminated so that any exploitation attempt is shut down before it can do damage. The CFI runtime engine has to be implemented wisely as it is the only part of the process's code base that must be trusted at runtime. Furthermore, CFI naturally comes with a significant performance penalty as every transition requires an additional check. One major requirement for a CFI engine thus is to achieve reasonable performance so that the slow down does not affect the end user in any negative way.

One problem of many CFI solutions is that they rely on the presence of the source code of the respective protected program. This facilitates the detection of control flow transitions in the code during the compilation stage, since a variety of high-level abstractions, such as the abstract syntax tree (AST) of the program, are available. Relevant program flow transitions and valid target labels can be easily extracted using this meta information. As opposed to this, the final binary representation contains close to no additional meta information apart from the plain program code and data. This also means the developing a CFI solution for proprietary systems comes with additional challenges that must be tackled appropriately. It is also

easier to introduce the actual CFI check code into existing source code during the compilation phase rather than providing a runtime library that modifies existing code. Doing so is highly involved and requires rewriting many parts of the actual program code without breaking any semantics.

Abadi et al. developed a CFI framework for x86-based processors on Windows operating systems [2]. At the time of writing, there exists no comparable framework for smartphone devices, which have been gaining widespread popularity among users recently. It is unclear as to how an existing *proprietary* CFI approach can be ported to mobile devices as they run on fundamentally different hardware architectures than traditional desktop/server systems.

This chapter discusses the design and implementation of the runtime engine of the Mobile CFI (MoCFI) framework for iOS operating systems. MoCFI is tailored to the ARM architecture which is the prevalent CPU architecture on mobile devices. At the time of writing, we are the first to develop such a framework for this specific architecture in particular, but also proprietary mobile devices in general. We chose to target iOS as an operating system for two reasons: first, applications are developed using Apple's Objective-C programming language, which compiles to native code and has an unsafe typing system. This makes iOS applications especially susceptible to control flow attacks. Second, iOS is a proprietary system. Note that in fact, some parts of the OS X operating system that iOS relies on are available as open source [16]. However, the entire application runtime system that is built around the App Store is proprietary.

4.1.1 Contributions

In this chapter, we describe the design and implementation of the *runtime components* of the MoCFI framework. We had to develop new technical approaches because of the proprietary nature of the targeted iOS operating system. We thus cannot rely on compile-time information and have to modify the binary instead of the source code to insert our checks. Furthermore, MoCFI is the first CFI framework for mobile devices, which are typical developed for RISC ARM processors which can be substantially different to x86 CISC processors. We evaluated our framework using several popular apps and show that the performance overhead remains within reasonable bounds.

MoCFI was also extended in a second tool called PSiOS which provides fine-grained policy enforcement for iOS devices. PSiOS allows the user to specify individual policy rule sets for specific applications. The tool greatly enhances user privacy in a proprietary environment which otherwise grants extensive access to privacy-critical resources to every application. Similar solutions have only been proposed for open source platforms such as Android.

This chapter is based on previous publications together with Davi, Dmitrienko,

Egele, Fischer, Holz, Nürnberger, and Sadeghi [50], and Werthmann, Davi, Sadeghi, and Holz [154].

4.1.2 Outline

The chapter is structured in the following way: at first, we introduce CFI, discuss the related work in this area, and then provide the technical background with regards to iOS and the ARM architecture in Section 4.2. We then proceed to describe the design of our prototype in Section 4.4 and the corresponding implementation details in Section 4.5. Section 4.6 discusses possible limitations of our approach and how they can be addressed. The evaluation is provided in Section 4.7. Section 4.8 presents the policy enforcement extension PSiOS and Section 4.9 concludes the chapter and discusses possible future work.

4.2 Technical Background

In this section, we provide the necessary technical background. We first explain the concept of CFI and then proceed to present various technical aspects of iOS and the ARM architecture.

4.2.1 Control Flow Integrity

CFI is a security concept in which all control flow transitions of executed programs are being monitored and validated against a set of allowed transitions. Typically, the CFI engine holds a list of allowed target transitions for each indirect control flow transition within the code and checks whether the execution follows along these allowed transitions. If at any point this is not the case, a control flow violation is detected which is equivalent to a security violation. As a consequence, the execution is usually stopped or other corresponding measures are taken. Every control flow violation is considered as a malicious exploitation attempt from an attacker or program bug, as the program code would never produce a violation under normal circumstances. Since almost every software vulnerability exploit relies on attempting a controlled invalid control flow transition at some stage, such exploits are successfully thwarted by CFI solutions.

The set of valid control flow transitions is dictated by the CFG of the program. The CFG contains all valid transitions of the code. It is typically constructed once beforehand in a static analysis. The CFG can be build either during the compilation phase (thus relying on the presence of the source code of the protected program/library) or using only the compiled binary code. The former option is easier to implement due to the availability of high level program information which can greatly facilitate the CFG construction. The latter option is more involved and

can also be more error-prone but allows to even protecting proprietary applications for which no source code is available.

Consequently, CFI solutions performs two major individual steps: (1) the construction of the CFG and (2) the CFI enforcement at runtime. In case of binary CFI solutions, the CFG construction involves disassembling the program and correctly identifying all control flow transition instructions and their legitimate branch target addresses. Special care has to be taken concerning the correctness of the disassembly. Code for CISC architectures such as x86/x64 is notoriously cumbersome to disassemble due to the presence of unaligned instructions and the diverse instruction set. In contrast, RISC architectures emit code that is easier to disassemble since all instructions are of the same size and thus aligned in memory. The MoCFI framework that we present in this thesis operates on ARM CPUs which is a typical RISC architecture.

In case of a successful disassembly, the next step is to identify all control flow instructions in the protected code. To that end, the code is typically divided into basic blocks (BBs). By definition, a BB is a contiguous code portion that has exactly one entry point and exactly one exit point. As a consequence, the program will never branch to another location while executing instructions *within* (i.e., between the first and last BB instruction) the BB and only the last instruction of the BB constitutes a control flow transition. Thus, once the basic blocks of the code have been extracted, only the last instruction of each BB is relevant for the CFG analysis.

Control flow transitions at the end of each BB can be either *static* or *dynamic*. Static transitions have the target address hardcoded in the instruction encoding. Branches (either unconditional or conditional) to a static memory address are typical examples. In case the operating system provides the W $\oplus$ X security property and thus prevents overwiting of code instructions, CFI frameworks can ignore static branches within the protected code since the target address is hardcoded in the instruction which cannot be overwritten by an attacker. The iOS operating system provides such a security protection. This means that MoCFI only has to consider dynamic control flow transitions. Typical examples of such transitions are dynamic branches (e.g., for using a function pointer that is only known at runtime or for implementing switch statements) and subroutine returns. Please note that for returns, it is difficult to determine valid control flow transitions in the static analysis since the return addresses are highly dynamic by nature. There are several approaches to address this issue. We chose to implement a shadow stack at runtime, as explained in Section 4.5.

Figure 4.1 shows an exemplary CFG with five different basic blocks. The connecting edges indicate that BB1 can reach BB2. Similarly, BB2 may transition to BB3 and BB4, and BB3 and BB4 may jump to BB5 at runtime. Each basic block contains and arbitrary set of instructions. The only restriction is that none of the instructions may lead to a control flow transition other than the subsequent instruc-

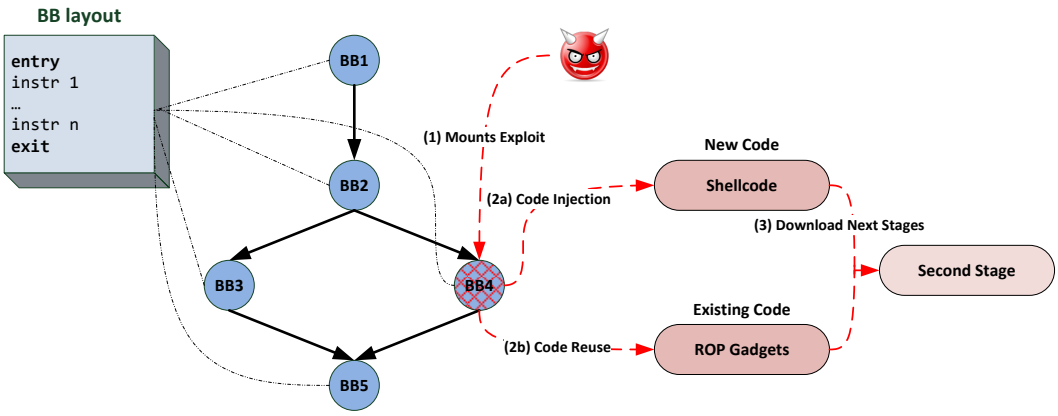


Figure 4.1: Schematic overview of control flow attacks.

tion. Any further semantics of each BB are irrelevant for CFI guarantees and are thus not considered by the framework.

The figure also shows how an attacker could redirect the control flow transition after exploiting a vulnerability at the end of BB4 (1). The two dashed lines (2a) and (2b) constitute two different kinds of exploit techniques: The former method (2a) jumps into an attacker supplied shellcode that was injected into the memory space previously. These kind of attacks are thwarted by the W $\oplus$ X protection on iOS. More sophisticated attacks, such as (2b), thus try to abuse existing code portions in the exploited process's address space. After that, an attacker usually downloads second stage code and data from an external server as indicated by (3). CFI protects against both types of exploits since both control flow transitions at the end of BB4 constitute CFG violations that are detected by the runtime engine.

The second part of a CFI solution is the *runtime engine* that monitors the execution of the protected program and validates occurring control flow transitions against the rules defined by the CFG. Binary CFI frameworks usually do this by inserting checks at code places that otherwise hold control flow transitions. Therefore, the code at these specific locations is rewritten so that execution is diverted into a CFI *validation routine*. In the following, we will call every control flow transition instruction that has to be rewritten a *check site*. The instruction that overwrites the check site to insert the check to the validation routine is called *dispatcher* instruction. This routine usually resides in a dedicated dynamic library that holds the CFI runtime engine. The rewriting happens at the very beginning of the process execution before any protected code is executed.

4.2.2 ARM Architecture

ARM features a 32 bit processor and sixteen general-purpose registers `r0` to `r15`, where `r13` is used as stack pointer (`sp`) and `r15` as program counter (`pc`). Furthermore, ARM maintains the so-called *current program status register* (`cpsr`) to reflect the current state of the system (e.g., condition flags, interrupt flags, etc.). In contrast to Intel x86, machine instructions are allowed to directly operate on the program counter `pc` (`EIP` on x86).

In general, ARM follows the *Reduced Instruction Set Computer* (RISC) design philosophy, e.g., it features dedicated load and store instructions, enforces aligned memory access, and offers instructions with a fixed length of 32 bits. However, since the introduction of the ARM7TDMI microprocessor, ARM provides a second instruction set called THUMB which usually has 16 bit instructions, and hence, is suitable for embedded systems with limited memory space.

The *ARM architecture procedure call standard* (AAPCS) document specifies the ARM calling convention for function calls [22]. In general, a function can be called by a `BL` (**B**ranch with **L**ink) or `BLX` (**B**ranch with **L**ink and e**X**change) instruction. `BLX` additionally allows indirect calls (i.e., the branch target is stored in a register), and the exchange ("*interworking*") from ARM to THUMB code and vice versa. Both instructions have in common that they store the return address (which is simply the instruction succeeding the `BLX`/`BL`) in the link register `lr` (`r14`). In order to allow nested function calls, the value of `lr` is usually pushed on the stack when the called function is entered.

Function returns are simply accomplished by loading the return address to `pc`. *Any* instruction capable of loading values from the stack or moving `lr` to `pc` can be used as return instruction. In particular, ARM compilers often use "load multiple" instructions as returns meaning that the instruction does not only enforce the return, but also loads several registers, e.g., `POP {R4-R7,PC}` loads `R4` to `R7` and the program counter with new values from the stack.

4.2.3 Apple iOS

Apple introduced iOS in 2007 as the mobile counterpart of its desktop operating system Mac OS X. It is a proprietary closed source operating system that runs on all of Apple's mobile devices including iPhones, iPads, and iPods. Currently, iOS only supports ARM based processors as all mobile Apple devices exclusively use this architecture. Please note that, since iOS is heavily based on Mac OS X, both operating systems partly share the same source code base. Parts of the Mac OS X kernel, and therefore also parts of iOS, are available as open source software [16]. However, many security relevant features implemented in iOS are not part of the publicly available source code. iOS is still a proprietary closed platform: third party

software may only be installed using the Apple App Store, which yields Apple full control over which application can be used by users. Every App Store application goes through an automatic review process before it can be made available in the App Store [15]. This enforcement is guaranteed by using binary signing: every application has to be signed by Apple before it can be executed on an iOS device.

In terms of software security, the App Store signing process provides a barrier for malware since it is at least unlikely that obvious malicious programs will be available through the App Store. The signature can also be revoked in retrospect by Apple. However, this does not provide any additional security against malicious attackers exploiting software vulnerabilities in benign applications. Therefore, Apple introduced several additional software security schemes which we explain in the following.

The previously discussed W $\oplus$ X memory protection technique has been implemented since iOS version 2. As a consequence, memory regions can never be both executable and writable which thwarts the injection and execution of machine instruction-based shellcode during an exploit. Conventional exploits injecting shellcode on to the stack (or the heap) [12] are thus not viable on iOS. It is noteworthy to mention that the W $\oplus$ X implementation of iOS is significantly stronger than most similar protections of other operating systems due to the presence of an additional feature called code signing enforcement (CSE) [167]. CSE prevents applications from *dynamically* allocating executable memory unless its signature includes a special entitlement (as explained later on in this section). On a technical level, applications thus may not call the memory allocation function `mprotect` using the `PROT_EXEC` flag. The primary idea of CSE is to prevent applications from dynamically downloading new code at runtime and executing it, thus undermining the application review process which only covers static application code. As a side effect, CSE also effectively prevents the use of multi-stage exploits that use an initial return-to-libc or ROP stub to download a second traditional shellcode. However, it does not prevent the execution of full ROP shellcode. Apple still has the possibility to grant special applications a so-called entitlement (which is part of the application's signature) that allows the use of dynamically allocated code. For example, applications that incorporate just in time (JIT) compilers, such as Apple's web browser Safari, must include such an entitlement. The bottom line also is that CFI solutions on iOS do not have to cope with dynamically allocated code.

Another security building block of iOS is the use of the Stack Smashing Protector (SSP). SSP protects against stack-based buffer overflows by introducing stack canaries in the pro- and epilogs of compiled subroutines, as described in Section 4.3. Moreover, SSP features bounds checking for selected critical functions (like `memcpy` and `strcpy`) to ensure that their arguments will not lead to stack overflows. SSP only protects stacks and does not affect other memory allocation facilities such as the heap manager.

Starting from iOS v4.3, ASLR randomizes the base addresses of loaded system libraries and other memory areas (such as stack and heap). However, Apple's implementation has proven to be particularly inconsequent and thus vulnerable. For example, the base image of the application is not randomized. This usually allows to mount arbitrary ROP attack despite the presence of ASLR on system libraries. Schwartz et al. [130] showed that typically a few kilobytes of code are sufficient.

A recent extension of the iOS operating system is the introduction of the App Sandbox [14]. The App Sandbox allows developers to assign and revoke certain access permissions to their applications. The system thereby works at the granularity of *entitlements*. Entitlements are security capabilities that are determined at compile time and incorporated into the signature of the application. Examples of typical App Sandbox entitlements are access to certain folder such as the music folder, access to hardware devices such as camera, and access to personal databases such as the address book. The combination of entitlements and sandboxing allow containing the damage of an exploited application. However, many applications hold a variety of different entitlements by default. Moreover, the entitlements are enforced by the signature which cannot be modified in hindsight. App developers may choose to apply for a temporary excception entitlement that disables the App Sandbox entirely. Apple therefore lists a number of app behaviors and use cases which are incompatible with the App Sandbox and thus need an exception [21].

4.2.4 Objective-C

Almost all iOS applications are written in the object-oriented programming language *Objective-C*. The inherent structure of this language has several implications on the design and implementation of our framework. In this section, we shed light on the technical background of Objective-C.

The Objective-C programming language extends standard ANSI C and supports the same basic syntax. Objective-C provides a syntax for defining classes and methods, as well as other constructs that promote dynamic extension of classes. One outstanding characteristics of Objective-C is that all object processing related tasks involve heavy use of a *runtime system* [17]. This runtime sytem is implemented in several shared libraries and plays a pivotal role in the execution of Objective-C applications.

One of the key Objective-C peculiarities is the method dispatch system. In Objective-C, messages are the equivalent of method calls in C++/Java. In order to call a method in an object or class, the system dispatches a message through the runtime system and does not call the method directly. Every message is dispatched using a dedicated runtime system function called `objc_msgSend`. The general layout of an Objective-C message in the Objective-C source code looks as follows:

```
[object method:arg1 param_name1:arg2 ... param_nameN-1:argN]
```

The targeted object is called the *receiver* and the method signature that consists of the method and parameters names is called the *selector*. In conclusion, an Objective-C message consists of the receiver object, the selector, and the argument values.

Dynamic Binding

One of the key aspects of the runtime implementation of Objective-C is that messages are compiled to a call to the generic message dispatching routine `objc_msgSend`, which resides in one of the shared libraries of the Objective-C subsystem. This technique is called *dynamic binding*. Thus *every* object message, including messages between application internal objects, is implement as an `objc_msgSend` call. Receiver, selector, and argument values are passed as arguments to this generic function.

Consider the following Objective-C message example:

```
[scanner setScanLocation:2]
```

As explained above, **scanner** constitutes the receiver object, **SetScanLocation** is the selector, and the only passed argument is a static value of 2. This message is translated by the compiler into the following piece of ARM assembler:

```
MOVS    R2, #2
LDR     R0, =(selRef_setScanLocation_ - 0x41CA)
LDR     R1, [R0]
LDR     R0, [SP, #0x3C+var_1C]
BLX     _objc_msgSend
```

The receiver is passed as the first argument in `r0`. In our case, it is a local variable referenced through the stack pointer `sp`. The second parameter is the *compiled* selector. Every selector is assigned a compiled selector of the type `SEL` that is registered to the runtime system at application load time. Compiled selectors are typically stored within the application binary in a specific section that is described by the Mach-O headers. The runtime system then only works with compiled selectors to refrain from parsing selector strings, which would be a time-consuming task.

Dynamic binding exacerbates the implementation of a sound CFI solution: attackers may be able to modify the target object or selector parameters and thereby perform invalid control flow modifications to unexpected objects and methods. This may eventually lead to the execution of arbitrary attacker-controlled computations. Naive CFI solutions will not detected this since the transition occurs inside the Objective-C runtime library.

We added support for dynamic binding to MoCFI as this is an especially important topic since almost all applications use the Objective-C API. Thus, the majority of function calls within iOS applications are typically `objc_msgSend` calls. We track the correct object types and selectors during the static analysis. Our runtime engine

later handles these messages and ensures that valid objects and selectors are being used by the application. A similar approach was already followed by some existing analyzers [57, 53]. However, these tools only work statically.

4.3 Related Work

In this section, we discuss the relevant related work in the domain of control flow integrity solutions. At first, we discuss several security policy enforcement frameworks, since they are related to CFI. We then present several exploit mitigation solutions that employ CFI at some stage. Finally, we also discuss orthogonal defense mechanisms that strive for similar goals as CFI.

Security Policy Enforcement

The underlying principle of monitoring control flow has originally been employed to enforce execution policies on targeted applications. Such systems do not provide the comprehensive security guarantees of control flow integrity solutions, which monitor *every* control flow transition in the target. However, the principle of security policy enforcement is still very similar to CFI.

Schneider [128] introduced the general concept of execution monitoring (EM) for enforcing security policies on targeted code. His paper presents a formal characterization of EM enforcement mechanisms, an automata-based formalism for specifying policy rules, and discusses how this security automata can be used for policy enforcement. By definition, CFI solutions do not fall under the category of EMs since they typically modify the target application's code, which is not allowed for EM enforcements. Apart from that, both approaches do strive for the same goals.

SASI [59] is a reference monitor developed by Erlingsson and Schneider that also enforces security policies. The framework was implemented for both the x86 and Java JVML platform. Although SASI works on the machine level, the x86 implementation still relies on the assembler output of the gcc compiler in the last compilation step. The engine therefore makes assumptions about the structure of the assembler output which only apply to code emitted by the gcc compiler. Therefore, the framework cannot be applied generically to proprietary binary programs since these might be transformed into binary code using various compilers.

A similar system called Naccio [63] was introduced by Evans and Twyman. Naccio is a framework that allows the specification and enforcement of security policies to place arbitrary constraints on resources. It uses code rewriting to insert the check code. The Java VM as well as the Windows Win32 platform are supported. However, in case of the latter Naccio only supports the somehow obscure DEC Alpha CPU architecture whose development was phased out in 2004.

Polymer [28] is another security policy framework that redirects security-sensitive operations to a validation routine which then verifies the operation against a list of pre-defined policies. The system provides a sophisticated policy specification system in which policy rules can be arranged in hierarchies. In order to insert the policy checks, the application code is rewritten at runtime. Polymer supports Java VM applications.

Exploit Mitigation

The first work that introduced the concept of control flow integrity for the purpose of mitigating software vulnerability exploitation was presented by Kiriansky et al. as *program shepherding* [96]. Their system provides three buildings blocks that ensure that shepherding can

- restrict execution privileges depending on origins,
- restrict control transfer based on instruction class, source, and target, and
- guarantee that inserted check points will never be bypassed by the target code.

The first point ensures that malicious code cannot conceal itself as data rather than code. Note that this technique has become obsolete by the recent introduction of W $\oplus$ X protection schemes in operating systems, which do not permit data to be executable. The second building block represents the typical core principals of control flow integrity, while the last point ensures that the system cannot be outsmarted by a malicious attacker. The authors implemented their approach in a reference system that is built on top of the x86 dynamic instrumentation platform DynamoRIO [24], the framework thus only runs on x86 architectures. Therefore, the target code is instrumented and check code is inserted at appropriate check sites during the instrumentation – no source code access is required. In order to achieve reasonable performance, the system makes aggressive use of caching on the BB level to instrument reoccurring BBs as rarely as possible. Please note that the paper follows a different approach than MoCFI. Instead of using instrumentation, we rely on binary rewriting which is a less complicated and faster approach.

Abadi et al. [2] introduced the term control flow integrity (CFI) for exploit mitigation. The authors provide a general formal foundation of the concept of CFI and also present a practical implementation for x86 Windows systems that operates on top of the instrumentation engine Vulcan [56]. The framework does not rely on source code access and can be integrated into proprietary software systems. However, it does require the presence of debug information that is generated during the compilation process and is typically stripped from final release versions.

As pointed out by Abadi et al., CFI can also serve as a supplementary measure to secure higher-level protection schemes. For example, it can ensure the invulnerability of software fault isolation (SFI) [149] systems against malicious attackers by protecting security-critical components in the framework from being bypassed or tampered with. One example of this is XFI [3], which extends a CFI engine to allow for additional integrity constraints on memory accesses.

Recent work has employed CFI to harden security-critical system components such as hypervisors. For example, HyperSafe [150] is a system that protects x86 hypervisors by enforcing lifetime code and control flow integrity of the hypervisor code. The framework constructs the CFG and inserts runtime checks to verify the validity of control flow transitions. HyperSafe requires access to the hypervisor source code and makes some simplifying assumptions in validating valid return addresses for subroutine calls which cannot be transferred to the domain of MoCFI.

Orthogonal Defenses

Both the academic and private sector have developed various different defense mechanisms independent on CFI solutions. These systems eventually strive for the same goal: to mitigate the exploitation of software vulnerabilities within application code. One technique that has received widespread adoption into many common compiler frameworks is the concept of *stack canaries*. StackGuard [47] and StackGhost [68] are compiler modifications that push a randomly generated number (the so-called stack canary) on the stack in the prolog of each subroutine. Before the subroutine returns, the epilog checks whether the exact same random number is still present on the stack. If a buffer overflow occurred in the meantime that targets return addresses on stack, then this random number is bound to be overwritten by the exploit too. Since the attacker cannot know the value of the stack canary beforehand, another value will be written to the stack. Consequently, StackGuard detects this modification and triggers a security violation before any malicious code can be executed. Stack canaries have, among others, been implemented in Microsoft's Visual Studio compilers, the GCC tool chain and Apple's Clang compiler. The concept of canaries has also been ported to other data structures such as heap objects. While they constitute an efficient hurdle to thwart software exploits, stack canaries can only protect against buffer overflow exploits. The approach also fails to detect overwriting of local function pointers since these are located below the canary on the stack and thus cannot be protected.

RAD [41] is a compiler extensions that protects against stack-based overflows by outsourcing return addresses to a dedicated and specially protected stack. PointGuard [48] encrypts pointer values and only decrypts them at runtime as soon as they are loaded into CPU registers. Both approaches postulate modifications in the compiler and thus require recompilation to apply the protection. Furthermore, they

introduce a non-negligible performance overhead.

ASLR is a scheme that leverages the fact that attackers lack the knowledge of the concrete memory layout of the exploited target process. By randomizing the loading addresses of the application and all loaded libraries, exploitation attempts are thwarted since malicious code can never be executed unless concrete or roughly estimated memory addresses are known. Chapter 3 shows that ASLR, in certain circumstances, can be prone to side channel attacks. Furthermore, shortcomings in ASLR implementations such as data leakages [91] or insufficient randomization [134] can undermine the entire approach.

Another protection scheme that has found widespread adoption is the W $\oplus$ X principle (also called Data Execution Prevention (DEP)). W $\oplus$ X enforces that memory can never be both writable and executable. The idea behind is to prevent attackers from injecting and subsequently executing shellcode in vulnerable applications since doing so requires a memory area that is writable *and* executable. W $\oplus$ X can be considered to be merely a supplementary protection scheme since its presence alone was proven to be insufficient. Return-to-Libc [136] and ROP [133] attacks are able to successfully and reliably circumvent W $\oplus$ X by abusing already existing code in the system instead of injecting own machine instructions. This allows an attacker to execute arbitrary computations despite the presence of W $\oplus$ X.

Binary Rewriting

Binary rewriting is a technique that has been employed by numerous binary security or optimization frameworks [126, 62, 132, 148, 56] and is also used by MoCFI. However, all of these frameworks yet require compiler-generated information such as debug meta data that is typically not shipped with release versions. They can thus not be applied in our scenario and are inappropriate for effective retrofitting of binary software.

More related to our work is SecondWrite [115], a generic binary rewriting framework that does not require additional debug information and aims to provide a stable fundament for retrofitting security-related logics into binary programs. The system is built on top of the intermediate language toolset LLVM and thus a entirely different design philosophy.

4.4 Framework Design

In this section, we describe the unique challenges that have to be addressed when implementing a CFI framework for the proprietary iOS operating system. At first, we name the various different challenges and explain why existing CFI approaches cannot be adapted to our scenario. In the second part of the section, we describe how we addressed these challenges and how our framework is structured.

4.4.1 Technical Challenges

Section 4.3 described various different CFI-related solutions that have been proposed by academic researchers. However, none of these approaches can be transferred to our scenario due design restrictions and technical peculiarities that we describe in the following.

Proprietary Software Environment

Almost all existing CFI frameworks require the presence of additional compile-time meta data that is stripped from release versions. For example, the most current CFI solution by Abadi et al. for Windows x86-based systems requires the presence of debug information. Such information is almost never available on proprietary systems that work with binaries. Distributing debug information with release version is usually undesirable since it allows to draw various inferences about the original source code: debug information contains source code filenames, variable and subroutine symbols up to concrete source code lines. This greatly facilitates the reverse engineering of binary code.

We implement MoCFI for a closed operating system, which cannot be modified in order to include our solution in the standard application load process. Second, end user devices and even App Store maintainers have no access to the application source code. Hence, a compiler-based CFI solution is not practicable from the end user's perspective.

Due to the bonding of iOS with centralized software distribution facilities, it also features application encryption and signing. Many existing CFI approaches apply patches to the static binary files. Similar approaches are unfeasible in the presence of code signing since modification of application files breaks the signature. Our system therefore has to make sure that all patches are applied at runtime, *after* the signature checks, and *before* the program execution.

CPU Architecture

The ARM architecture provides a second instruction set called THUMB. Both ARM and THUMB code can be interleaved arbitrarily, there are no restrictions on the application as to when it employs which instruction set. Application may thus switch between both instruction set modes at any point of execution. The runtime enforcement thus has to support both modes in order to function properly.

As mentioned in Section 4.2.2, ARM does not provide dedicated return instructions. Instead, any branch instruction that modifies the instruction pointer can be used as a return. Moreover, returns are usually accomplished using load multiple register (LDM) instructions that have side effects on other registers than the instruction pointer, meaning that the return does not only enforce the return to the caller,

but also restores several registers at the same time. This has several implications at runtime when handling return instructions with respect to the corresponding side effects.

Lastly, the ARM program counter `pc` is a general purpose register which can thus be directly accessed by a number of instructions, e.g., arithmetic instructions are allowed to load the result of an arithmetic operation directly into `pc`. Furthermore, a load instruction may use the current value of `pc` as base address to load a pointer into another register. This complicates a CFI solution on ARM, since we have to consider and correctly handle all possible control flow changes, and also preserve the accurate execution of load instruction that use `pc` as base register.

4.4.2 General Framework Design

Figure 4.2 shows the general design of MoCFI. The entire framework is separated into two different components: the *static* analysis and the *dynamic* runtime enforcement. The static analysis only has to be performed once as an initial step. Therefore, the application has to be decrypted and disassembled. In the next step, the static analysis of MoCFI generates two kinds of output information: first, it identifies all relevant branches within the target's code. This includes all dynamic branches, subroutine calls, and subroutine returns. As explained in Section 4.2.1, static branches do not have to be considered due to the active W $\oplus$ X protection. Addresses and meta information of relevant branches, which we call *check sites* in the following, are saved in the so-called *patchfile*. The second task of the static analysis is to reconstruct the CFG of the application and save it in another file. The output files of the static analysis process only need to be generated once for each application.

The second component of MoCFI is the dynamic runtime enforcement which is implemented as a shared system library (called the *runtime engine* in the following). The runtime engine adjusts the protected application's code after process initialization as specified by the previously generated patchfile. At every check site, our system ensures that the destination of the control flow transition is a valid target in the pre-generated CFG.

Please note that this thesis only covers the runtime enforcement components of MoCFI. The interested reader can refer to our paper [50] for more details on the static analysis. In the following, we explain three core design decisions of the MoCFI runtime engine that distinguish our solution from similar frameworks.

Binary Rewriting

One of the core design decisions of a CFI framework is how the framework modifies the target application so that control flow is redirected at the check sites into a

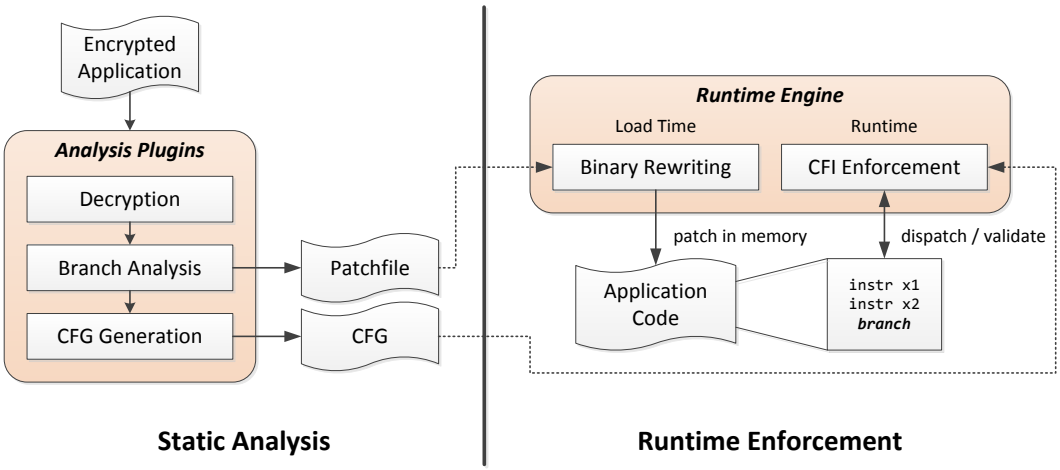


Figure 4.2: Design of the MoCFI framework.

validation routine. Previous x86-based enforcement frameworks [2] are built on top of binary instrumentation engines such as PIN [103] and Vulcan [55]. This decision stems from the fact that x86 employs a variable-length instruction set encoding. Overwriting instructions at check sites to implement a jump to the validation routine would be highly complicated since the overwriting instruction might be longer than the overwritten instruction, thereby affecting subsequent instructions as well. In combination with the possibility of PC-relative instructions, this requires to implement a complex instruction parsing engine. Furthermore, x86 provides a variety of different instructions due to its CISC nature and thus many corner cases would have to be handled properly. This problem is solved by implementing the CFI engine on top of existing binary instrumentation frameworks.

However, the design of a mobile CFI framework needs to be approached from a different angle. Firstly, there exists no binary instrumentation engine for ARM processors at the time of writing this thesis. Binary instrumentation frameworks are cumbersome to implement and typically induce a significant performance overhead. Thirdly, almost all relevant mobile processor architectures (including ARM) to this date follow the RISC design philosophy which also means that there exists a fixed instruction length[1].

We therefore chose to take a different approach in MoCFI called *binary rewriting* [159]. Instead of relying on an instrumentation engine, we directly overwrite the target code at check sites (typically one single instruction) and provide a static jump into our validation routine (the *dispatcher* instruction). This static jump can be

[1]Please note that THUMB mode can constitute an exception to this rule. We explain how we tackle this problem in Section 4.5.

implemented using one single instruction. In some corner cases in THUMB mode, it might be necessary to overwrite at most two instructions (see Section 4.5).

An important requirement for our framework is that the rewriting has to happen dynamically *after* the process was initialized. Static patches are not feasible since this would break the mandatory digital signature of the binary; the operating system would thus refuse to load the application. Furthermore, the rewriting must take place *before* any application or other library code was executed. We chose to implement the runtime engine in a dedicated shared library that is included in the protected application's load process. At load time, the runtime engine applies binary rewriting to the check sites, then it starts the actual application. At runtime, the validation routine validates relevant control flow transitions against the CFG.

Validation Routine and Dispatching

The actual enforcement in the validation routine is one of the core components in the system. In order to reduce the code base and complexity of the runtime engine, we favor to use a single *generic* validation routine that is responsible for all check sites.

There are several obstacles one faces when using generic validation routines (see Section 4.5.3). We solve these issues using so-called *trampolines*. Trampolines are short instruction sequences that prepare the entrance to the validation routine and also perform the transition back to the target code after the validation is completed. Every check site has an individual trampoline and dispatcher instructions thus target corresponding trampolines and not directly the validation routine. We stress that we are the first to use the concept of trampolines for CFI enforcement.

Shadow Stack

The static analysis cannot fully pre-compute all valid control flow transitions beforehand. Subroutine calls are highly dynamic by nature and thus any return from a subroutine can possible target a large variety of call sites. We chose to implement a *shadow stack* [41] to address this case. At every subroutine call, we save the return address in a dedicated, safe shadow stack. When the subroutine returns, we then compare this saved value against the requested return address. If both addresses do not match, then we detect a control flow validation. The advantage of shadow stacks is that they are straightforward to implement and require a low performance overhead.

4.5 Implementation Details

We developed a prototyp MoCFI implementation that supports iOS versions 4.3.1 up to 5.1.1. At the time of development, no stable jailbreak available for iOS 6.x was available, which is why we did not test the framework with this version. However, we expect that only minor changes are needed to port MoCFI and plan to do this as future work. In this section, we describe the implementation details of the *runtime engine* of MoCFI, which is implemented as a shared library. The library itself was developed using Xcode 4 and the code base amounts to 1,430 lines of code (LOC). We opted to use a preferably small code base to minimize the risk of implementation bugs in the framework.

The section is structured on the following way: we describe the load time phase of the runtime engine, elaborate on various aspects of the runtime CFI enforcement, explain the use of trampolines in detail, and conclude by providing several examples of rewritten check sites.

4.5.1 Load Time Initialization

The load time phase is performed *once* during process initialization. MoCFI carries out all of the preparatory measures to ensure CFI validation at runtime. The main task of the load time phase thus is to initialize the MoCFI runtime engine and rewrite the code of the protected application.

Shared Library Injection

Most UNIX-based operating systems support the injection of libraries by providing the environment variable `LD_PRELOAD` that is checked by the OS loader upon initialization of each new process. The loader ensures that the library is loaded before any other dependency of the actual program binary. iOS provides an analogous method through the `DYLD_INSERT_LIBRARIES` environment variable [18]. Setting this variable to point to MoCFI enables us to transparently rewrite arbitrary applications. To force loading MoCFI into every application started though the touch screen, we only need to set this environment variable for the initial *SpringBoard* process.

We stress that MoCFI is initialized *before* any other dependency of the program and *after* the signature of the application was verified (i.e., the program binary itself is unaltered). Subsequently, MoCFI implements the CFI enforcement by rewriting the code of the application in memory.

Binary Rewriting

Upon initialization of MoCFI, the load time module first locates the correct patchfile. Afterwards, it rewrites the binary according to the information stored in the

patchfile. Since Apple iOS enforces $W \oplus X$, code cannot be writable in memory. Therefore, the code pages must be set to writable (but not executable) first. This is usually accomplished using the POSIX-compliant `mprotect` system call. Our experiments revealed that iOS does not allow the code pages to be changed at all (`mprotect` returns a permission denied error). However, this problem can be overcome by re-mapping the corresponding memory areas with `mmap` [19] first. When all relevant instructions of a page are patched, the page permissions are set back to executable but not writable. Note that the presence of `mmap` does not give an adversary the opportunity to subvert MoCFI by overwriting code. In order to do so, she would have to mount an attack beforehand that would inherently violate CFI at some point, which in turn would be detected by MoCFI.

4.5.2 CFI Enforcement

In the following, we explain the implementation details of the CFI enforcement of MoCFI, i.e., the part of the implementation that is continuously triggered while the protected application executes. We present the structure of our generic validation routine, explain several technical challenges that arise due to design choices, and name implications that result from the Objective-C dynamic binding system. We chose to explain the solutions to the technical challenges in own sections afterwards.

Generic Validation Routine

We use a single generic validation routine (residing in our MoCFI shared library) that is responsible for validating control flow transitions from all check sites. This generic validation routine is thus responsible for validating all kinds of different check sites (internal calls, external calls, returns, Objective-C messages, etc.). This minimizes the amount of new code that is introduced during runtime which in turn reduces cache consumption and thus also the overhead. Using multiple validation routines would also mean that every single routine allocates additional space in the code and data caches which can greatly reduce the overall performance of the system.

Technical Challenges

The use of a generic validation routine leads to implications that have to be addressed accordingly. In the following, we explain several problems that arise in using a generic validation routine.

Argument preparation. Our binary rewriting engine overwrites the relevant control flow instructions (i.e., the check sites) with so-called *dispatcher instructions*. Dispatchers, however, cannot simply redirect the program flow into the validation

routine since certain arguments need to be prepared and overwritten instructions might have to be re-executed at a different location *before* the validation routine, as we explain later on. The validation routine requires the following information to perform the validation:

1. The source address of the check site,
2. the requested destination address of the control flow transition, and
3. the CFG of the application.

The CFG is loaded during the initialization of our runtime engine and is then available as a global variable. The source and destination addresses of the control flow transition have to be extracted and arranged properly.

Dispatcher range. The dispatcher instruction itself is implemented as a simple static branch that directly redirects the program flow to the validation. In ARM, static direct branches that are encoded directly within the instruction can target a relative memory interval ranging from -/+ 32MB. This means that the destination address must not be further away than 32 MB from the dispatcher. However, we cannot make sure that the runtime engine is loaded close to the protected application since the operating system does not provide such an option. In fact, the distance between shared libraries in memory is usually always bigger than 32 MB. As a consequence, dispatchers cannot directly jump into the generic validation routine and thus another pre-step is necessary.

Instruction emulation. The dispatcher instruction is always a 32bit instruction, in both ARM and Thumb mode[2]. If the protected code is compiled using the Thumb instruction set (which is usually the case for iOS applications), then we might need to overwrite an additional instruction before or after the actual check site, since the check site can only be 16bit long. This means that we have to emulate this overwritten instruction (see Section 4.5.5 for an example). Notwithstanding the above, the check site instruction itself also has to be emulated at the end of the validation. This is a non-trivial task since control flow transition (and thus check sites) can occur in a variety of different instructions with multiple side-effects (such as the ARM load multiple instruction `LDM` [23]).

Objective-C Dynamic Binding

As explained in the background (see Section 4.2), all Objective-C messages are implemented as a call to the external `objc_msgSend` function in the Objective-C

[2]Thumb mode allows for mixed 16bit and 32bit instructions and also supports 32bit direct branches.

runtime library. Since we strive for validating Objective-C message to ensure that a concrete check site calls the correct method in the correct object, these calls must be handled specifically. Otherwise an attacker might have the ability to modify the object or selector of an `objc_msgSend` call to tamper with the control flow. The object, selector, and argument values of an Objective-C message are passed as arguments to `objc_msgSend`. MoCFI detects `objc_msgSend` calls and handles them appropriately. The additional Objective-C information (i.e., the object class and selector) are stored by the static analysis and embedded in the CFG information. Hence, MoCFI uses this information to check whether the called object and selector indeed match with the designated ones.

4.5.3 Trampolines

In the last section, we listed several technical challenges in using a generic validation routine. We solve the aforementioned problems by employing so-called *trampolines*. Dispatcher instructions thus redirect the program flow to trampolines, which in turn transfer the execution to the generic validation routine in our MoCFI runtime engine. Hence, the trampolines are used as bridges between the application we aim to protect and our MoCFI library. Trampolines are essentially dynamically allocated short pieces of assembler code and we opted to build them as small as possible to reduce to amount of additionally executed instructions and thus reduce the performance impact. As we explain in the following, there exist multiple trampolines that are used depending on the type of the corresponding check site instruction. In the following, we explain how trampolines solve the problems mentioned in Section 4.5.2.

Argument preparation. One of the fundamental tasks of a trampoline is to extract the source and destination address of every control flow validation request. Therefore, the trampoline at first saves all relevant register values; both the source and the destination addresses can be inferred from these register values. Furthermore, the trampoline prepares these arguments before it jumps into the generic validation routine.

Dispatcher range. Trampolines also solve the problem of the limited dispatcher branch range since they are allocated dynamically during load time using the `mmap` system call. `mmap` allocates memory ranges and also allows to specify at which virtual memory address the allocated memory should start. We therefore call `mmap` and choose a memory address directly adjacent to the protected application image to ensure that trampolines are allocated within the 32MB range of the code of the protected application. Note that this approach is not feasible in case the image size surpasses the 32MB boundary because the distance between the first (or last) code

instructions and the trampoline becomes too big. However, we have not come across an application which violates this criterion during our evaluation. Even if this were the case, we could still fix the problem by overwriting the subsequent instruction of the check site – a technique which we already employ for the Thumb instruction set (see Section 4.5.5).

Instruction Emulation. Another advantage of trampolines is that we can largely refrain from using instruction emulation, which is an error-prone and complex task due to the complexity of assembler instruction encodings. Instead, we provide custom trampolines for check sites that would otherwise require instruction emulation. Trampolines allow us to copy overwritten instructions at the beginning or end of the trampoline so that they do not have to be emulated in the validation routine. The only exception in which we have to apply emulation is when we overwrite a pc-relative control flow instruction, since the instruction then resides in the trampoline and not the original memory location. In this case, we perform a rudimentary emulation. Note that such scenarios are very rare in practice.

Trampoline Types

Our system uses three different types of trampolines that are used depending on the type of the check site:

- one generic internal trampoline,
- one generic external trampoline, and
- multiple custom trampolines.

We opt to use generic trampolines (i.e., trampolines which are used by multiple check sites) whenever possible and custom trampolines when we would otherwise need instruction emulation. Custom trampolines are thus always assigned to one concrete check site. All trampolines are allocated adjacent to each other using the `mmap` system call, as described previously in this section. Specifically, we allocate custom trampolines for each indirect branch (i.e., indirect jumps / calls and returns), one generic trampoline for direct internal function calls (i.e., calls within the same code segment), and one generic trampoline for external function calls. Figure 4.3 illustrates the decision process and shows when which kind of trampoline type is employed.

The generic internal trampoline is used for dispatching internal subroutine calls, the generic external trampoline is used for dispatching external subroutine calls, i.e., calls to API functions that reside in system shared libraries. The reason for this distinction lies in the fact that we must use two different shadow stacks for

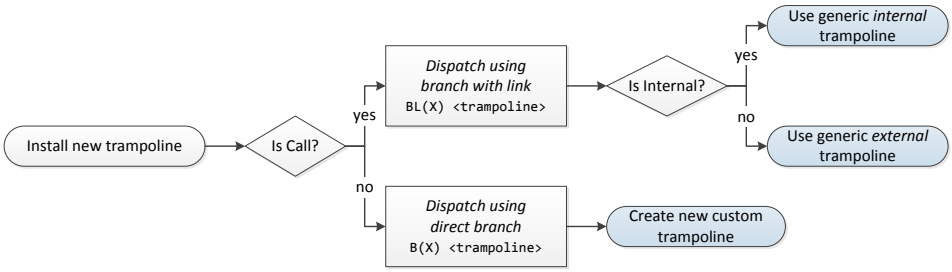


Figure 4.3: Flow chart of the trampoline decision process.

internal and external subroutine calls. By intuition, a shadow stack works in the following way: whenever a subroutine call occurs, we push the return address onto the shadow stack. Whenever a return occurs, we compare the return address against the last saved address in the shadow stack. If both values differ, a CFG violation is detected. However, there exists one corner case in practice that has to be taken into account.

Consider the following scenario: the target application calls an external API function; the API then performs a callback into another application subroutine before returning. When this callback subroutine returns, MoCFI tries to validate the return address using the shadow stack. However, since MoCFI did not see the corresponding call into the subroutine, which lies in an external shared library that is not protected by the runtime engine, this will always produces a violation. The solution to this dilemma is to use an additional stack (the *branch location stack*) that keeps track of possible callback routines. We chose to implement the stack handling directly at the trampoline stage for convenience and performance reasons in form of the generic external trampoline.

Detailed Structure

Figure 4.4 shows a detailed instruction listing for each four different trampoline types that exist. Listing (a) shows the instructions of the generic internal trampoline. The trampoline saves all relevant registers in the beginning so that the effects of the subsequent validation routine, which inherently alters several registers, can be revoked. The following instructions prepare arguments and branch into the validation routine in the runtime engine (a pointer to this routine is stored statically at the end of the trampoline). The validation routine takes one argument: a pointer to the previously saved registers. This is necessary to reconstruct the original arguments (which are passed in R0-R3) of the requested subroutine call in the application code. The validation routine is called using a `BLX` instruction and thus returns back into the next trampoline instruction; the control flow destination

address is returned in register R0 as dictated by the standard ARM calling convention (see Section 4.2.2). This value is stored temporarily on the stack, the original register values are restored, and the trampoline proceeds to the destination address.

The generic external trampoline shown in Listing (b) is to some extent similar to the internal trampoline. The first five instructions are thus identical between both types. Instead of branching to the destination address, the external call trampoline branches into the external library and sets the return address to point into trampoline. As soon as the external call returns into the trampoline, register values are saved again and we branch into the branch location stack handler in MoCFI. After that the trampoline proceeds to return to the protected application.

We provide two different custom trampoline templates: one for Thumb and one for ARM check sites. Both templates are very similar as can be seen in Listings (c) and (d). In case of Thumb trampolines (Listing (c)), it might happen that the dispatcher instruction must overwrite an additional previous instruction. In this case, the overwritten instruction is copied at the beginning of the custom trampoline. The trampoline then branches into the validation routine. Note that we also provide an additional second parameter for custom trampolines. It contains a direct pointer into a check site metadata structure. This stems from the fact that we have an individual trampoline anyway and can thus individual pointers to metadata that would otherwise have to be searched for by the validation routine first. We thus increase the overall performance by using this additional second parameter. Our ARM trampoline template (Listing (d)) is very similar, except that there exists no room for previous instructions. This is due to the fact that ARM dispatcher instructions never overwrite more than one instruction and thus no additional space in the trampoline is needed.

4.5.4 Dispatching Through Exception Handling

In rare corner cases, our approach of overwriting the check site with a dispatcher instruction is not viable. This can happen when we need to overwrite a 16-bit Thumb instruction with a 32-bit dispatch instruction but cannot overwrite any previous or subsequent instruction, e.g., if the instruction preceding the branch references the program counter or is itself a branch. Consider the following instruction sequence:

```
...
LDR R2, [PC, 16]
POP {R4-R7, PC}
```

The first instruction `LDR R2, [PC, 16]` uses the current value of `pc` to load a pointer. Such an instruction cannot be simply moved into a custom trampoline due to the PC relative addressing of the second operand. In such scenarios, we use an entirely different approach: upon initialization, we register an iOS exception handler for

```
STMFD    SP!, {R0-R12, LR}
MOV      R0, SP
LDR      R1, [PC, 12]
BLX      R1 ; -> validate routine
STR      R0, [SP, -4]
LDMFD    SP!, {R0-R12, LR}
LDR      PC, [SP, -60]
<Ptr. to validation routine>
```

(a) Generic internal call trampoline

```
STMFD    SP!, {R0-R12, LR}
MOV      R0, SP
LDR      R1, [PC, 40]
BLX      R1
   ; -> validate routine
STR      R0, [SP, -4]
LDMFD    SP!, {R0-R12, LR}
MOV      LR, PC
LDR      PC, [SP, -60];
   ; -> external library
STMFD    SP!, {R0-R3}
LDR      R1, [PC, 16]
BLX      R1
   ; -> handle branch loc. stack
MOV      LR, R0
LDMFD    SP!, {R0-R3}
BX       LR
<Ptr. to validation routine>
<Ptr. to branch loc. stack
   handler>
```

(b) Generic external call trampoline

```
<Previous instruction or NOP>
PUSH.W {R0-R12, LR}
MOV    R0, SP
LDR.W  R1, [PC, 12]
LDR.W  R2, [PC, 12]
BLX    R2 ; -> validate routine
POP.W  {R0-R12, LR}
<Original return instruction>
<Ptr. to check site metadata>
<Ptr. to validation routine>
```

(c) Custom Thumb trampoline template

```
STMFD    SP!, {R0-R12, LR}
MOV      R0, SP
LDR      R1, [PC, 12]
LDR      R2, [PC, 12]
BLX      R2 ; -> validate routine
LDMFD    SP!, {R0-R12, LR}
<Original return instruction>
<Ptr. to check site metadata>
<Ptr. to validation routine>
```

(d) Custom ARM trampoline template

Figure 4.4: Trampoline templates.

illegal instructions. The dispatcher instruction is then simply an illegal 16-bit instruction that will trigger our exception handler in the runtime engine of MoCFI. Since this technique induces additional performance overhead (the CPU has to process the entire exception chain first), we only use it for exceptional cases. To further reduce the use of the exception handler, one could calculate the address from which `pc` is loaded in the static analysis phase and replace the relevant load instruction with a new memory load instruction which could be placed at the beginning of the trampoline.

Note that our exception handler forwards all exceptions not caused by MoCFI. Furthermore, by monitoring the exception handling API, we can ensure the position of our handler within the exception chain is not changed by the protected application.

4.5.5 Examples

Figure 4.5 shows a detailed overview of the control flow in any of the four different dispatching scenarios that can occur during a validation. The code depicted on the top left is the untouched original instruction sequence that is rewritten during load time by MoCFI into the sequence that is shown in the middle of the figure. The first case (the black lines) shows how custom trampolines are employed. The preceding instruction of the check site is overwritten in order to allocate sufficient room for a 32-bit branch instruction into a custom trampoline. In the beginning of the custom trampoline, we can see that the previously overwritten instruction `MOV R1, R2` is executed at first. After that, the trampoline saves all register values and branches to the generic validation routine in the runtime engine, being located in the shared library. Upon successful validation, MoCFI returns to the custom trampoline where previously saved registers values are restored and the original return instruction `POP {R4-R7, PC}` is executed.

The second case (the green lines) illustrates an exception handler example in which MoCFI uses the operating system exception handler to dispatch into the validation routine. The exception handler is registered at load time and remains active throughout the entire execution of the protected application. As soon as the illegal instruction is executed, the CPU generates an exception that is passed on to the operating system which in turn forwards the error handling to our registered handler. We then reconstruct the registers values that were active at the time the exception was triggered, call the validation routine, and then cleanup the exception state and continue the application at the destination address of the control flow request. Please note that we pass on exception handling if MoCFI recognizes that the occurred exception is in fact not associated with the rewriting.

In the third case (the orange lines), the figure shows how internal function calls are dispatched into the generic internal trampoline. This trampoline is in fact used

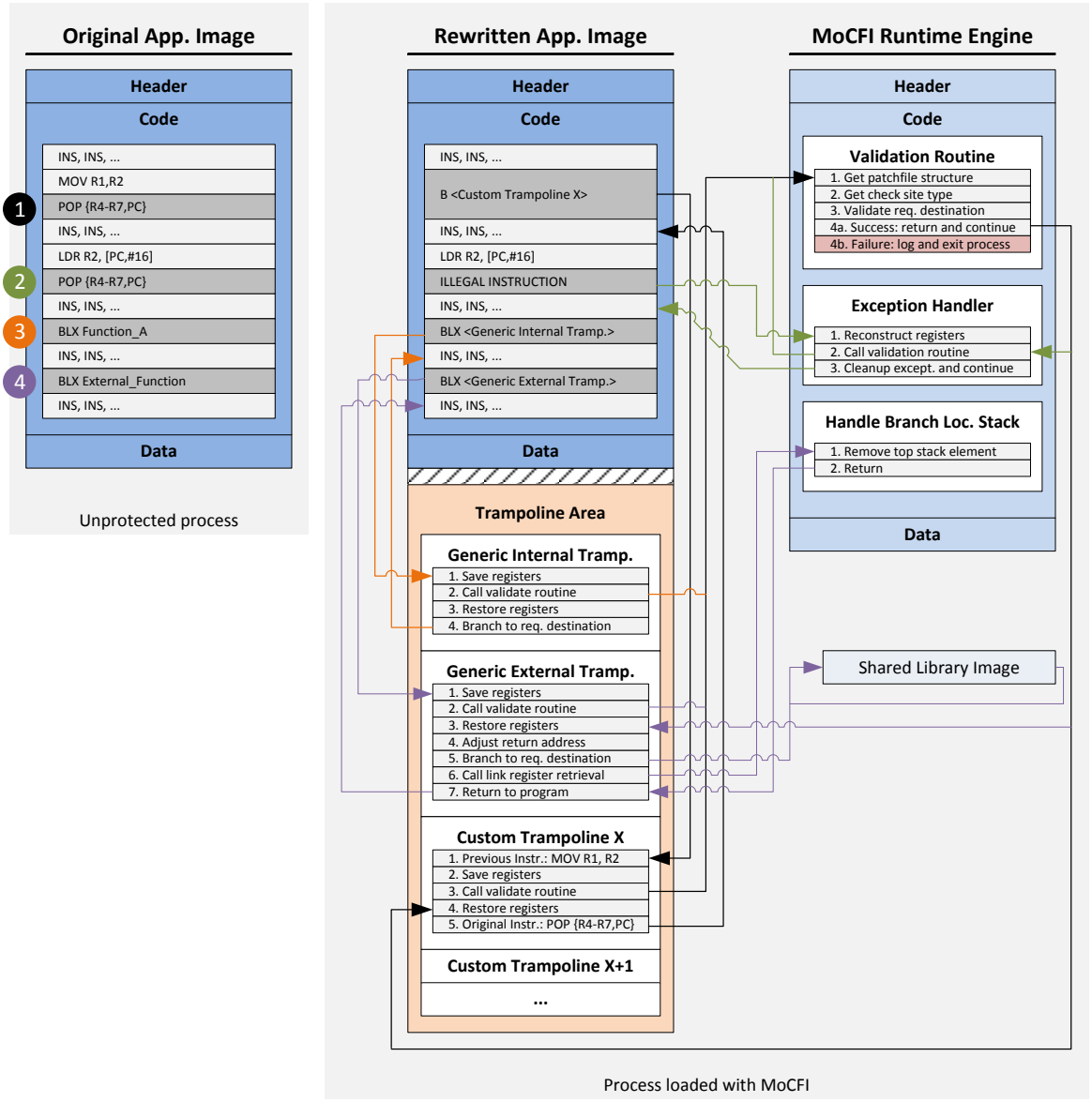


Figure 4.5: Summary and examples of all trampoline types.

by *all* internal function calls. Since these calls are always 32-bit long, there exists no need for a custom trampoline. The trampoline functions very similar to the custom trampolines except that no custom instructions are executed at the beginning or the end.

The final case (the purple lines) illustrated how external function calls are redirected into the generic external trampoline. The external trampoline at first validates the control flow request just as in the internal trampoline. However, it then modifies the return address in the `LR` register to point into the trampoline. As a consequence, the called function will return into the trampoline rather than the application code directly. This is done to then call another subroutine in MoCFI that is responsible for updating the branch location stack accordingly. We refrain from describing the details of this subroutine for the sake of brevity.

4.6 Discussion and Limitations

In this section, we discuss possible weak spots of our approach. We show in which scenarios certain limitations may become relevant and what additional changes would be necessary in order to increase the effectiveness and usability of MoCFI.

Integration into iOS. In order to install MoCFI on an iOS device, currently a jailbreak is required. This is merely necessary to modify the environment variable that injects our library in every process. No other modifications that presume a jailbreak exist. Since MoCFI performs binary rewriting after the iOS loader has verified the application signature, our scheme is compatible to application signing. On the other hand, our runtime engine is not directly compatible to the iOS CSE (code signing enforcement) runtime model (see Section 4.2.3). CSE prohibits any code generation (including code rewriting) at runtime on non-jailbroken devices, except if an application has been granted the *dynamic-signing* entitlement. To tackle this issue, one could assign the *dynamic-signing* entitlement to applications that should be executed under the protection of MoCFI. On the one hand, this is a reasonable approach, since the general security goal of CFI is to protect benign applications rather than malicious ones. Further, the *dynamic-signing* entitlement will not give an adversary the opportunity to circumvent MoCFI by overwriting existing control flow checks in benign applications. In order to do so, she would have to mount a control flow attack beforehand that would be detected by MoCFI.

On the other hand, when *dynamic-signing* is in place, benign applications may unintentionally download new (potentially) malicious code, or malicious applications may be accidentally granted the *dynamic-signing* entitlement (since they should run under protection of MoCFI) and afterwards perform malicious actions. Furthermore, disallowing code generation at runtime also obviates the need for corre-

sponding support in MoCFI.

Providing CFI for runtime generated code is a cumbersome and error-prone task. To address these problems, one could constrain binary rewriting to the load time phase of an application, so that the *dynamic-signing* entitlement is not needed while the application is executing. Further, new sandbox policies can be specified that only allow the MoCFI library to issue the `mmap` call to replace existing code, e.g., the internal page flags of the affected memory page are not changed, or their values are correctly reset after MoCFI completed the binary rewriting process.

Error-proneness of the implementation. Finally, special care must be taken that an adversary cannot tamper with the MoCFI library and thus bypass MoCFI. Since our library is small in size, the probability for exploitable vulnerabilities is very low. Given the small code base, we could also apply code verification tools. We leave this task as future work.

Insecure exception handlers. Similar to other solutions (e.g., Abadi et al.), our current implementation does not detect attacks exploiting exception handlers: an adversary can overwrite pointers to an exception handler and then deliberately cause an exception (e.g., by corrupting a pointer before it is de-referenced). This is possible because the current compiler implementation used by the iOS development framework pushes these pointers on the stack on demand. We stress that this is rather a shortcoming of the compiler and the iOS operating system as it is the task of the operating system to provide a resilient exception handling implementation. Similar problems have already been solved on other platforms, such as on Windows with SafeSEH [108]. Therefore, we highly encourage Apple to port these techniques to iOS.

Shared library support. As already mentioned in Section 4.5, MoCFI does currently not protect shared libraries, which an adversary may exploit to launch a control flow attack. However, extending MoCFI accordingly is straightforward, because we already overcame the conceptional obstacles. It has to be noted that Apple explicitly prevents developers from employing shared libraries in their application packages. Applications that contain shared libraries do not receive a valid signature and thus cannot be published in the official App Store. Hence, all loaded libraries are typically from the operating system and Objective C frameworks. Consequently, we currently disable the return check if an external library calls a function that resides in the main application. Therefore, MoCFI registers when execution is redirected to a library and disables the return address check for functions that are directly invoked by the shared library. However, note that this can be easily fixed by either applying our trampoline approach to function prologs (i.e., pushing the

App. name	Code size	# Calls	# Returns	# Ind. jumps
Facebook	2.3MB	33,647	5,988	20
TexasHoldem	2.8MB	62,576	4,864	1
Minesweeper	0.7MB	12,485	1,882	20
Gowalla	0.7MB	11,969	1,647	0

Table 4.1: Tested iOS applications and occurrences of specific check sites.

return address on the shadow stack at function entry) or by also applying MoCFI to shared libraries.

4.7 Evaluation

We performed several tests in our evaluation of MoCFI to assess its effectiveness and usability. The evaluation can thus be separated into two main parts: at first we used our framework to protect a variety of real and widespread iOS applications from the official App Store. Furthermore, we then conducted several performance benchmark tests to estimate the induced runtime overhead of MoCFI. Performance tests are an important factor of CFI frameworks since the underlying technique typically comes with a significant slowdown that can drastically affect the user experience.

4.7.1 Qualitative Tests

We used our framework to protect several popular iOS applications from the official Apple App Store. We therefore evaluated Facebook, Minesweeper, TexasHoldem, Gowalla, WhatsApp, ImageCrop, BatteryLife, Flashlight, ImageCrop, InstaGram, MusicDownloader, MyVideo, NewYork (Game), Quickscan, LinPack, Satellite TV, and the Audi App. We could not perceive a notable negative overhead when working with any of the applications. Table 4.1 shows an overview of four of the tested applications and the number of respective calls, returns, and indirect jumps that each binary contains. Please note that the rewriting overhead also remains between reasonable bounds. In any of the binaries, we did not notice an overhead larger than half a second for the initial rewriting.

For all of these applications, we tried to execute all of their functionality, for example by surfing several facebook profiles and modifying our own profile in the Facebook application. We noticed that MoCFI largely works unnoticed by the user. However, we could also find one compatibility problem in which the framework mistakenly detects control flow violations. After analyzing the binary we found out that this is due to the way the GCC compiler implements large switch tables. At the point of the switch case, the compiler outsources the switch condition evaluation into

a separate subroutine and then branches to that subroutine using the `BLX` instruction. At the end of that subroutine, the compiler does not insert a typical return but rather jumps to the switch handler (lying in the original subroutine) directly. This yields a CFG violation since there is no corresponding return to the subroutine call and the shadow stack thus corrupts. We are unsure as to why the compiler chooses this kind of optimization. We stress that this is not a fundamental issue of our approach but rather a specific compiler peculiarity and plan to support such special cases in future revisions of our framework.

We also tested our framework with a self-developed vulnerable application that can be exploited from externally using a typical buffer overflow. As expected, MoCFI successfully detects the invalid control flow transition and terminates the program.

4.7.2 Performance Tests

We performed two different tests to evaluate the performance overhead induced by MoCFI:

1. Measure the overhead on the different tests of the iOS benchmark tool *Gensystek Lite*[3].

2. Measure an upper bound overhead using a self-developed quicksort implementation.

Gensystek. Gensystek provides a number of different benchmark checks and runs on every iOS driven device. Again we want to stress that we only protect the main application binary. Some of the benchmarks perform API calls at some stages that divert into unprotected libraries and thus the performance measurements would be slightly worse in a fully protected system. However, the majority of the computations in the tests indeed occur within the main binary files. The results of the measurements are shown in Figure 4.6. The overhead ranges between 3.85x and 5x with the PI Calc benchmark being the slowest test. Please note that these are very CPU intensive tasks that are not performed by typical applications, which rather react to user interactions with small computations. We thus think that the overhead is still reasonable. On the other hand, the overhead for image processing benchmarks (which are in our opinion more practically relevant) is almost negligible and amounts to 1% to 21%.

[3] `http://www.ooparts-universe.com/apps/app_gensystek.html`

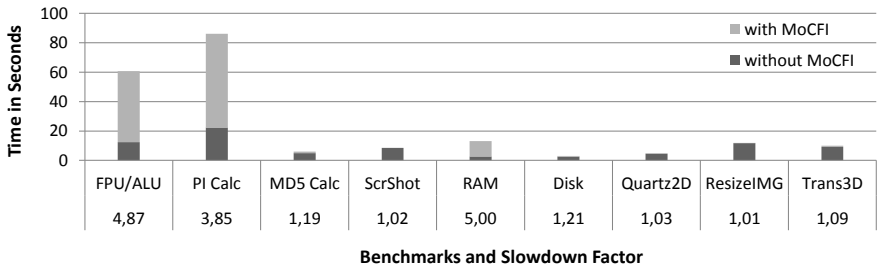


Figure 4.6: Gensystek Lite Benchmarks.

n	Without MoCFI	With MoCFI
100	0.047 ms	0.432 ms
1000	0.473 ms	6.186 ms
10000	6.725 ms	81.163 ms

Table 4.2: Quicksort measurements.

Quicksort. In order to approximate an upper boundary for performance penalties, we evaluated MoCFI by running a quicksort algorithm. The quicksort algorithm that we chose to implement makes use of a recursive comparison function. Thus, many control flow validations must be performed since many subroutine calls occur during execution. The comparison subroutine is also very small in size and the precentage of check sites hence high. Table 4.2 shows three different measurements for different array sizes (n=100, n=1000, and n=10000). The measurements show that in the worst case, MoCFI induced a slowdown factor of approx. 12x. Again, we stress that this is a worst case scenario that does not apply to the majority of mobile applications.

4.8 Enhancing Privacy Through Policy Enforcement

In this chapter, we introduced MoCFI as an efficient framework to mitigate control flow attacks against proprietary iOS applications. In this scenario, attackers exploit vulnerabilities from the outside of the application to gain access to it and the underlying operating system. From a broader perspective, external attackers are not the only threat for a smartphone end user. Mobile devices carry a plethora of personal information. The address book, history logs, location data, photos, and similar data sources allow drawing detailed conclusions about the social life of the owner of the device. Such information represents financial value to data harvesters. It is one of the device vendor's tasks to provide an execution environment in which sensitive data on the device is protected from illicit access. Specifically, applications should

not be granted dispensable access to private data. Unfortunately, the approach that iOS follows is insufficient.

Every application in the App Store has to pass a review phase which aims at ruling out unsolicited apps. According to the guidelines, applications must ask for the user's consent before accessing sensitive information. However, this is merely a formal restriction and no additional technical measures restrain developers from secretly accessing sensitive data. In fact, several incidents in the past showed that the guidelines and the review phase are insufficient [131].

Several researchers already presented privacy-extending frameworks for mobile platforms. Hornyack et al. [80] introduced a framework for Android that allows replacing real sensitive data with shadow data and mark certain data sources for device-only use so that it cannot be transmitted over network connections. Apex [111] is another policy enforcement framework for Android that gives the user enhanced control over how sensitive data can be process by applications. Both systems were developed for open platforms. However, when implementing such a policy enforcement framework for a proprietary system such as iOS, the situation is significantly different. We cannot simply extend existing libraries and API layers since we do not have access to the source code of the corresponding components. Furthermore, concepts such as dynamic binding introduce various implications and greatly exacerbate the design and implementation of a sound policy enforcement framework for iOS systems. All applications under iOS are compiled in binary assembler code and do not use intermediate language representations such as the Java VM on Android. This also means that applications are free to directly access various API layers in the system (the Objective-C API, the POSIX UNIX API, etc.).

We developed a novel tool called PSiOS which is built on top of MoCFI. PSiOS uses our CFI framework as the basis to provide additional policy enforcement checks whenever the application calls an external API function. Since our tool is able to intercept any control flow transition from within the application to any shared library, we allow for very fine-grained restriction policies that are significantly more powerful than iOS sandboxing profiles.

4.8.1 Technical Background

In the following, we discuss two important aspects of the technical background of PSiOS. At first, we discuss the different API layers that must be supported by a policy enforcement framework on iOS to support fine-grained rules. We then introduce the iOS sandbox and name its shortcomings over our approach.

API Layers

The iOS operating system provides multiple APIs for developers. The Objective-C Core Services can be regarded as the most important set of API functions that are used by the majority of applications. These services are organized in different system libraries that are tagged as either *private* or *public* frameworks. Private frameworks are only accessible from system libraries; public frameworks are also open to the application. Please note that this is merely a guideline provided by Apple. Since private frameworks lie in the same process memory as the application, there exists no technical barrier that holds back the application from calling private framework APIs nevertheless.

Another important observation is that iOS yet supports an orthogonal set of APIs in form of, e.g., the POSIX API and Mach API. These libraries are completely independent of Objective-C runtime components and are implemented as direct system calls into the corresponding system libraries. PSiOS supports all kinds of APIs.

Shortcomings of the App Sandbox

As mentioned previously in this section, iOS already includes an application sandbox that can restrict the access to system resources on a per-process basis. The sandbox is implemented within the iOS kernel and monitors the syscall interface. Apple therefore introduced so-called *entitlements* [20]. Entitlements assign certain rights to certain applications. At the time of writing, there exist 25 different entitlements which control access to several resources, e.g., access to the music folder, microphone, establishing network connections, and so on. These entitlements are declared within the digital signature of the application and are enforced by the iOS sandbox.

The iOS sandbox has the following shortcomings:

1. Entitlements are irrevocable since they are part of the digital signature. Thus, they cannot be changed in hindsight. Users cannot use individual policies.

2. The iOS sandbox is rather coarse-grained since it works on the syscall level. It is not possible to provide function-specific granularity.

3. Insecure applications can still be exploited by an adversary who then gains access to the device's resources on the same level as the application.

4. The App Sandbox is incompatible with many practical use cases and therefore exception entitlements are assigned that disable the sandbox [21].

4.8.2 Design and Implementation

PSiOS extends the underlying MoCFI framework so that the user may provide arbitrary policy rules for each individual application on the device. Rules are specified using an XML-based format. They are fine-grained and allow applying constraints to individual parameters of Objective-C messages or API function calls. For example, this allows restricting the access to certain files by specifying corresponding constraints on file open library calls (such as `fopen`). PSiOS is implemented as an academic prototype to demonstrate the technical feasibiliy of the approach. This also means that there is no convenient graphical user interface and rules have to be written manually at the time of writing. We leave this open for future work.

We implemented a new policy enforcement component at runtime on top of MoCFI, which validates calls to external shared libraries against the policy rule set. Please note that all CFI checks are still enabled in the background by MoCFI. Thus attackers cannot exploit software vulnerabilities, take over the execution, and hence access system resources in the context of the exploited application. Figure 4.7 shows an overview on how PSiOS validates external library calls.

PSiOS allows specifying three different response types in case a policy violation was detected:

1. *Exit*: the program is terminated.

2. *Log*: the incident is logged and the program continues execution.

3. *Replace*: returns forged data to the application and continues execution.

The replace option is especially interesting and allows returning forged data to the application in case a termination of the application process is undesirable. The current implementation of PSiOS replaces the corresponding method implementation which then returns an empty or arbitrary data structure. This might lead to compatibility problems in case the application does not expect empty objects to be returned. We plan to provide alternative replacement strategies for providing shadow data in the future.

4.8.3 Evaluation

We evaluated PSiOS in several scenarios. First of all, we applied PSiOS to the data harvesting application SpyPhone [110] to show that our system is capable of constraining access to various sensitive data sources. We also created individual policy rules for several popular iOS applications to restrict access to various data sources that are being accessed at runtime.

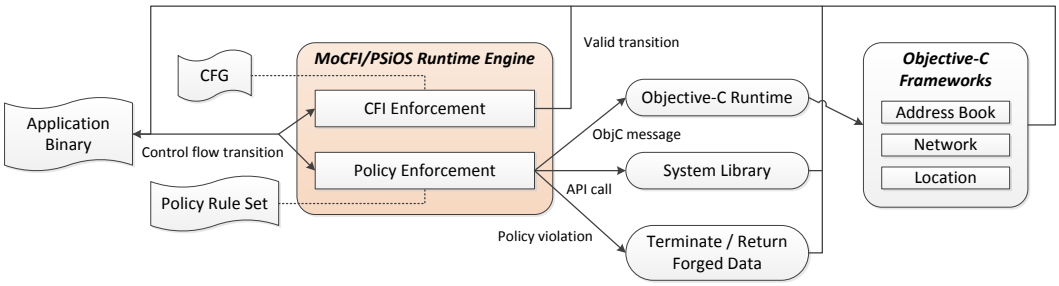


Figure 4.7: Schematic design of PSiOS.

SpyPhone

SpyPhone is a proof-of-concept application that demonstrates how iOS application can secretly access various sensitive data sources on the device per default. The collected data includes, amongst others, e-mail account information, WLAN network connection data, unique device or chip identifiers such as the UUID, address book data, location data, various history logs form web browsers and the phone application, photos, and even the keyboard cache. We created a comprehensive policy rule set to restrict SpyPhone from accessing any privacy-sensitive data source and we could confirm in our tests that PSiOS indeed blocks access to every single resource being access by SpyPhone. We therefore used the *Replace* response mechanism to return empty data.

Popular Applications

PSiOS was also applied to a number of popular iOS applications to demonstrate that it is capable of protecting real world applications. Namely, we tested Facebook, WhatsApp, ImageCrop, BatteryLife, Flashlight, ImageCrop, InstaGram, MusicDownloader, MyVideo, NewYork (Game), Quickscan, and the Audi App. An individual policy rule set was provided for each application, depending on what data the application accesses. We could successfully restrict the access to the address book (for Quickscan, Facebook, and Whatsapp), to personal photos (for ImageCrop and InstaGram), and to the iOS universal unique identifier, short UUID (for Quickscan, BatterLife, Flashlight, MusicDownloader, MyVideo, NewYork, and Audi).

4.9 Conclusion and Future Work

In this chapter, we described the design and implementation of the mobile CFI framework MoCFI. The framework runs on proprietary iOS Apple operating systems and allows to secure binary applications, i.e., no access to source code or debug

information is needed. This effectively allows us to retrofit security mechanisms into iOS that to successfully thwart software attacks that can otherwise be mounted by an attacker. We addressed a number of unique challenges that stem from the fact that iOS is a proprietary closed and mobile operating system. We introduced new techniques to implement a reliable runtime engine that enforces the control flow transition constraints as dictated by the CFG. The evaluation shows that MoCFI is able to successfully protect various popular and large iOS applications. Moreover, the performance overhead induced by the framework amounts to 3.85x to 5x for CPU intensive tasks. While we think that there is still room for optimizations, it has to be noted that typical mobile applications do not perform extensive calculations as this might heavily affect the device's battery service life. The practical effect of the performance slowdown is thus lower.

In a second step, we extended MoCFI in the policy enforcement framework PSiOS. PSiOS closes a privacy gap left by the insufficient sandbox system of iOS. Users can specify individual policy rules for each installed applications, thereby restricting access to sensitive data sources that are otherwise accessible to each application by default. The evaluation showed that PSiOS can be applied to popular iOS applications and successfully thwarts access to sensitive data when using the proof-of-concept data harvester SpyPhone.

Interesting future work advancements for MoCFI are to provide shared library support. This allows to also protecting critical system libraries such as the Objective-C runtime and several API libraries. Furthermore, we still expect that there is room for further performance optimizations to reduce the overhead of the CFI checks. Support for the newest iOS version 6 is also a task for future work.

5

Behavior-Graph-Based C&C Detection for Proprietary Operating Systems

5.1 Introduction

Malicious programs (abbrev. *malware*) have become a significant threat for end user and server computer systems. Researchers generally separate malware into several different classes such as worms, viruses, and trojan horses. One of the the most prevalent forms of malware that leverages the rising interconnectedness of computers are *bots*. Bots differentiate from other malware in that they possess a networking component that establishes connection to one or more so-called command and control (C&C) servers. This allows to control the bot from remotely by sending commands that control the its behavior or to provide bot updates. Bots are usually part of a bigger botnet that might be composed of millions of compromised machines in extreme cases [46, 70, 124, 49]. The person that controls the botnet in the background is commonly referred to as the *botmaster*. Bots can be used for a variety of nefarious purposes. For example they can abuse the infected machine's network bandwidth to participate in executing distributed denial of service (DDoS) flooding attacks. They might also be used for spying on data on the infected machine such as stealing bank account data or confidential company documents.

Researchers have proposed several countermeasures to mitigate the recent rise of the botnets. These solutions can be classified into two main categories: network-based and host-based approaches. Network-based approaches aim at creating network signatures solely by monitoring network data that is being sent and received from infected systems. Therefore, these solutions typically pass through learning phases in which bots are executed in a safe environment and produced traffic is collected. In the next step, this data is analyzed for unique and reoccurring features that seem to typify C&C connections. With the help of this data, network signatures that match these connections can be created or the C&C server addresses can

be logged or incorporated in blacklists. Network-based approaches however suffer from three major shortcomings that diminish their effectiveness in practice.

Benign connections. Not all network connections established from bots are malicious per se. Bots frequently open various other non-C&C connections, for example to check for a valid Internet connection or to synchronized the local system time with a remote network daemon. Unfortunately, telling apart legitimate non-C&C traffic from real C&C traffic is tough to achieve for network-based approaches. Thus, such solutions typically suffer from high false positive rates since they produce wrong signatures that also match benign connections from various harmless applications.

Noise injection attacks. Researchers have proposed a new kind of attack that leverages the inherent shortcomings of network-based only solutions. These attacks are closely related to the drawback described in the previous paragraph. Noise injection attacks [65, 66] produce a large chunk of (mostly useless) network traffic to conceal real C&C connections in a multitude of other connections. Since network-based approaches cannot distinguish between both types, they are incapable of producing meaningful signatures.

Encryption. Network-based solutions are completely rendered useless if the monitored bot employs traffic encryption since they rely on the semantics of the contents of network packets. If data is only transmitted in encrypted form, then these solutions have no means to draw any conclusion about the nature of the connection or to produce signatures.

These three shortcomings show that the inherent approach of network-based solutions, which is to solely rely on network packet data, is insufficient in many circumstances. Additional information that enriches the model with more semantical meaning is needed.

An orthogonal approach to bot detection is host-based solutions, which monitor and/or detect bot behavior directly on the host and not on the network-level. Typical examples of host-based solutions include anti-virus engines that either match manually generated signatures of known malicious programs or try to apply several heuristics to detect malicious behavior of unknown malware. Since these kinds of runtime solutions typically try to avoid detecting any false positives, signatures and heuristics are generated rather conservatively. This also means that it is easy to bypass any detection by malware even with techniques such as dynamic repacking in which every binary gets packed with a randomly generated encryption key. Generally speaking, host-based solutions oftentimes fail to attribute unknown binaries of existing malware to known families appropriately.

The entire bot problem is further exacerbated by the fact that most malware is written for Windows due to its tremendous market share for desktop computers. Windows is a proprietary operating system, which means one cannot simply extend or modify system libraries in order to add monitoring functionality since

corresponding source codes are not available. Furthermore, there exists a plethora of API layers that have to be monitored. These layers are eventually abstracted in Windows in a proprietary mostly undocumented native API in the background.

5.1.1 Contributions

In this chapter, we propose a new combined approach to bot detection. Our approach adopts concepts from both host- and network-based techniques to overcome the disadvantages of each respective approach. More specifically, our approach allows to reliably distinguishing malicious C&C connections from irrelevant benign and even purposely generated junk connections. Instead of relying only on network-level traffic, we monitor the bot in our special execution environment. By using data tainting techniques, we can enrich network packets with additional host-based information that allow drawing further conclusions about the semantics behind the generated packet. This allows us to determine whether sent packets contain data from system resources, and also enables one to detect in which way received data is processed by the bot.

The model we use in our approach are *behavior graphs* that characterize the exact behavior of each network connection and its interaction with the operating system. We present Jackstraws, a system that monitors bots in a special execution environment and create behavior graphs for established network connections. Behavior graphs combine host-level information with network data using taint tracking mechanisms. The behavior graphs can be further processed using by machine learning algorithms to identify subgraphs that are characteristic of malicious C&C connections. With the help of this data, it is also possible to produce generalized graph templates that match even new connection from previously unknown bots [1].

In our evaluation, we could show that Jackstraws is capable of generating templates that recognize malicious C&C traffic from known bots with high accuracy and few false positives. Moreover, our system was also able to detect C&C traffic generated by unknown bot samples that were not incorporated in the learning phase.

We stress that Jackstraws works for the proprietary Windows operating systems and we show at which key points the operating system has to be monitored to allow for the creation of generic and complete behavior graphs. More specifically, we show how to monitor the proprietary native API layer which allows us to refrain from coping with the multitude of open API layers.

This chapter is based on a previous publication together with Jacob, Kruegel, and Holz [87].

[1] Please note that the machine learning algorithms are not part of this thesis and are only introduced briefly.

5.1.2 Outline

The chapter is structured in the following way: Section 5.2 discusses related work. Section 5.3 gives an abstract overview over JACKSTRAWS and the different components that it consists of and implementation details are given in Section 5.4. The evaluation is presented in Section 5.5. We then conclude the chapter in Section 5.6 with a summary and a discussion of possible future work topics.

5.2 Related Work

Given the importance and prevalence of malware, it is not surprising that there exists a large body of work on techniques to detect and analyze this class of software. The different techniques can be broadly divided into host-based and network-based approaches, and we briefly describe the related work in the following.

5.2.1 Host-Based Detection

Host-based detection techniques include systems such as traditional anti-virus tools that examine programs for the presence of known malware. Other techniques work by monitoring the execution of a process for behaviors e.g., patterns of system calls [67, 100, 118]) that indicate malicious activity. Host-based approaches have the advantage that they can collect a wealth of detailed information about a program and its execution. Unfortunately, collecting a lot of information comes with a price; it incurs a significant performance penalty. Thus, detailed but costly monitoring is typically reserved for malware analysis, while detection systems, which are deployed on end-user machines, resort to fast but imprecise techniques [144]. As a result, current anti-virus products show poor detection rates [43].

A suitable technique to model the host-based activity of a program is a behavior graph. This approach has been successfully used in the past [44, 98, 69] and we also apply this technique. Recently, Fredrikson et al. introduced an approach to use graph mining on behavior graphs in order to distinguish between malicious and benign programs [69]. Compared to their work, we have another high-level goal: we want to learn which network connections are related to C&C traffic in an automated way. Thus we do not only focus on host-level activities, but also take the network-level view into account and correlate both. From a technical point of view, we perform a more fine-grained analysis by applying taint analysis instead of the coarse-grained analysis performed by [69].

BOTSWAT [139] analyzes how bots process network data by analyzing system calls and performing taint analysis. The system matches the observed behavior against a set of 18 manually generated behavior patterns. From a technical point of view, BOTSWAT uses library-call-level taint analysis and, thus, might miss certain

dependencies. In contrast, the data flow analysis support of JACKSTRAWS enables a more fine grained analysis of information flow dependency among system calls.

5.2.2 Network-Based Detection

To complement host-based systems and to provide an additional layer for defense-in-depth, researchers proposed network-based detection techniques [76, 74, 77, 164, 78, 162]. Network-based approaches have the advantage that they can cover a large number of hosts without requiring these hosts to install any software. This makes deployment easier and incurs no performance penalty for end users. On the downside, network-based techniques have a more limited view (they can only examine network traffic and encryption makes detection challenging), and they do not work for malicious code that does not produce any network traffic (which is rarely the case for modern malware).

Initially, network-based detectors focused on the artifacts produced by worms that spread autonomously through the Internet. Researchers proposed techniques to automatically generate payload-based signatures that match the exploits that worms use to compromise remote hosts [99, 135, 95, 112, 102]. With the advent of botnets, malware authors changed their *modus operandi*. In fact, bots rarely propagate by scanning for and exploiting vulnerable machines; instead, they are distributed through drive-by download exploits [123], spam emails [89], or file sharing networks [92]. However, bots do need to communicate with a command and control infrastructure. The reason is that bots need to receive commands and updates from their controller, and also upload stolen data and status information. As a result, researchers shifted their efforts to developing ways that can detect and disrupt malicious traffic between bots and their C&C infrastructure. In particular, researchers proposed approaches to identify (and subsequently block) the IP addresses and domains that host C&C infrastructures [140], techniques to generate payload signatures that match C&C connections [76, 74, 162], and anomaly-based systems to correlate network flows that exhibit a behavior characteristic of C&C traffic [78, 164, 77]. In a paper related to ours, Perdisci et al. studied how network traces of malware can be clustered to identify families of bots that perform similar C&C communication [121]. The clustering results can be used to generate signatures, but their approach does not take into account that bots generate benign traffic or can even deliberately inject noise [120, 66, 65, 8]. Our work is orthogonal to this approach since we can precisely identify connections related to C&C traffic.

5.3 System Overview

Our system monitors the execution of a malware program in a dynamic malware analysis environment (such as Anubis [85], BitBlaze [137], CWSandbox [156], or

Ether [51]). The goal is to identify those network connections that are used for C&C communication. To this end, we record the activities (in our case, system calls) on the host that are related to data that is sent over and received through each network connection. These activities are modeled as *behavior graphs*, which are graphs that capture system call invocations and data flows between system calls. In our setting, one graph is associated with each connection. As the next step, all behavior graphs that are created during the execution of a malware sample are matched against templates that represent different types of C&C communication. When a graph matches a template sufficiently closely, the corresponding connection is reported as C&C channel.

In the following paragraphs, we first discuss behavior graphs. We then provide an overview of the necessary steps to generate the C&C templates.

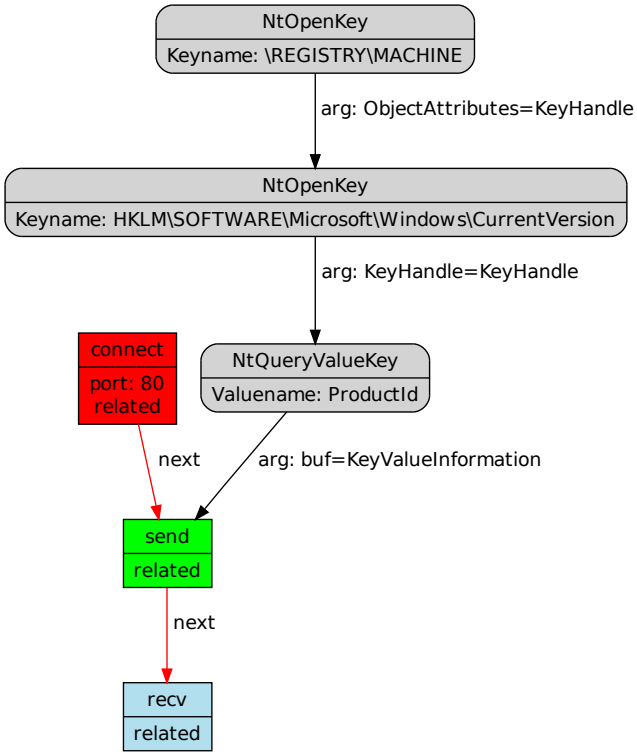
5.3.1 Behavior Graphs

A behavior graph G is a graph where nodes represent system calls. A directed edge e is introduced from node x to node y when the system call associated with y uses as argument some output that is produced by system call x. That is, an edge represents a data dependency between system calls x and y. Behavior graphs have been introduced in previous work as a suitable mechanism to model the host-based activity of (malware) programs [44, 98, 69]. The reason is that system calls capture the interactions of a program with its environment (e.g., the operating system or the network), and data flows represent a natural dependence and ordered relationship between two system calls where the output of one call is directly used as the input to the other one.

Nodes that are related to network system calls of the corresponding network connection of the graph (e.g., connect, send, recv) are connected by edges with a `next` label that represent the chronological order in which they occurred. If the tainting system indicates an information flow between the arguments of a network node and a system call, a corresponding node and edge is added to the graph. The edge is labeled accordingly with the argument names.

Figure 5.1 shows an example of a behavior graph. This graph captures the host-based activity of a bot that reads the Windows serial number (ID) from the registry and sends it to its command and control server. Frequently, bots collect a wealth of information about the infected, local system, and they send this information to their C&C servers. The graph shows the system calls that are invoked to open and read the Windows ID key from the registry. Then, the key is sent over a network connection (that was previously opened with `connect`). An answer is finally received from the server (`recv` node).

The *network* nodes correspond to `connect`, `send`, and `receive` events of the underlying connection. In the above example, one can thus see that the product ID




```
GET /bot/doit.php?v=3&id=ec32632b-29981-349-398...
```

Figure 5.1: Example of behavior graph that shows information leakage. Underneath, the network log shows that the Windows ID was leaked via the GET parameter `id`.

of the system is read from the registry. Nodes can contain additional labels that further describe the nature of the event, such as the key names in the example.

Please note that our system only includes information in the behavior graphs that is related to the specific network connection. This is in contrast to related word [69] in which the authors generate complete dependency graphs of the execution run of a given sample.

While behavior graphs are not novel *per se*, we use them in a different context to solve a novel problem. In previous work, behavior graphs were used to distinguish between malicious and benign program executions. In this work, we link behavior graphs to network traffic and combine these two views. That is, we use these graphs to identify command and control communication amidst all connections that are produced by a malware sample.

5.3.2 C&C Template Generation

As mentioned previously, the behavior graphs that are produced by our dynamic malware analysis system are matched against a set of C&C templates. C&C templates share many similarities with behavior graphs. In particular, nodes n carry information about system call names and arguments encoded as labels l_n, and edges e represent data dependencies where the type of flow is encoded as labels l_e. The main difference to behavior graphs is that the nodes of templates are divided into two classes; core and optional nodes. Core nodes capture the necessary parts of a malicious activity, while optional nodes are only sometimes present. Each C&C template represents a certain type of command and control activity and the generation of templates is separated into four subsequent steps.

In the first step, we run malware executables in our dynamic malware analysis environment, and extract the behavior graphs for their network connections. These connections can be benign or related to C&C traffic. JACKSTRAWS requires that some of these connections are labeled as either malicious or benign (for training). In our current system, we apply a set of signatures to all connections to find *(i)* known C&C communication and *(ii)* traffic that is known to be *unrelated* to C&C. Note that we have signatures that explicitly identify benign connections as such. The signatures were manually constructed, and they were given to us by a network security company. By matching the signatures against the network traffic, we find a set of behavior graphs that are associated with known C&C connections (called *malicious graph set*) and a set of behavior graphs associated with non-C&C traffic (called *benign graph set*). These sets serve as the basis for the subsequent steps.

The second step uses the malicious and the benign graph sets as inputs and performs graph mining. The goal of this step is to identify subgraphs within the behavior graphs that appear frequently in connections labeled as malicious. This suggests that this sort of behavior is common among various C&C connections.

As a third step, we cluster the graphs previously mined. The goal of this step is to group together graphs that correspond to a similar type of C&C activity. That is, when we have observed different instances of one particular behavior, we combine the corresponding graphs into one cluster.

In the fourth step, JACKSTRAWS produces a single *C&C template* for each cluster. The goal of a template is to capture the common core of the graphs in a cluster; with the assumption that this common core represents the key activities for a particular behavior.

At the end of these steps, we have extracted templates that match the core of the program activities for different types of commands, taking into account optional operations that are frequently (but not always) present. This allows us to match variants of C&C traffic that might be different (to a certain degree) from the exact graphs that we used to generate the C&C templates.

5.4 System Details

In this section, we provide an overview of the actual implementation of JACKSTRAWS and explain the different analysis steps in greater details.

5.4.1 Analysis Environment

We use the dynamic malware analysis environment Anubis [85] as the basis for our implementation, and implemented several extensions according to our needs. Note that the general approach and the concepts outlined in this paper are independent of the actual analysis environment; we could have also used BitBlaze, Ether, or any other dynamic malware analysis environment.

As discussed in Section 5.3, behavior graphs are used to capture and represent the host-based activity that malware performs. To create such behavior graphs, we execute a malware sample and record the system calls that this sample invokes. In addition, we identify dependencies between different events of the execution by making use of dynamic taint analysis [129], a technique that allows us to assess whether a register or memory value depends on the output of a certain operation. Anubis already comes with tainting propagation support. By default, all output arguments of system calls from the native Windows API (e.g., `NtCreateFile`, `NtCreateProcess`, etc.) are marked with a unique taint label. Anubis then propagates the taint information while the monitored system processes tainted data. Anubis also monitors if previously tainted data is used as an input argument for another system call.

Taint propagation is achieved by extending each relevant instruction translation from x86 into Qemu micro code. If we assume that an instruction takes two input operands i_1 and i_2, produces a result r and either i_1 or i_2 have an attached taint label, then r will have the same attached taint label. If both input arguments have

distinct taint labels, then one of them is picked. For performance reasons, every byte can only have a single label. We have not encountered major negative effects due to this simplification.

As previously mentioned, Anubis monitors the native Windows API, which resides in the library file `ntdll.dll` and constitutes the last step of a typical API call under Windows (before control is passed to the kernel). The advantage of only monitoring the last step of an API call lies in the inherent abstraction from the vast number of official Windows API functions. However, not all Windows API function invocations result in a call to the kernel. Notable examples include functions that return specific information about the operating system, underlying hardware, or time zone information (for example, `GetVersionEx`, `GetSystemInfo`, and `GetSystemTime`). These functions are relevant because bots often use them to collect system information that is then leaked to the C&C server. To capture this information, we extended the set of monitored functions to include Windows API calls that can be used to gather system information.

Native API calls by adding certain functions that are especially relevant to information leakage, since not all of the API functions Windows provides are processed by the Native API due to performance reasons. These functions reside in the first layer of Windows API libraries and must be monitored explicitly. Increasing the number of monitored system calls also results in more fine-granular and hence more meaningful results.

While Anubis propagates taint information for data in memory, it does not track taint information on the file system. In other words, if tainted data is written to a file and subsequently read back into memory, the original taint labels are not restored. This shortcoming turned out to be a significant drawback in our settings: For example, bots frequently download data from the C&C, decode it in memory, write this data to a file, and later execute it. Without taint tracking through the file system, we cannot identify the dependency between the data that is downloaded and the file that is later executed. Another example is the use of configuration data: Many malware samples retrieve configuration settings from their C&C servers, such as URLs that should be monitored for sensitive data or address lists for spam purposes. Such configuration data is often written to a dedicated file before it is loaded and used later. Restoring the original taint labels when files are read ensures that the subsequent bot activity is linked to the initial network connection and improves the completeness of the behavior graphs.

Finally, we improved the network logging abilities of Anubis by hooking directly into the Winsock API calls rather than considering only the abstract interface (`NtDeviceIOControlFile`) at the native system call level. This allows us to conveniently reconstruct the network flows, since send and receive operations are readily visible at the higher-level APIs.

5.4.2 Behavior Graph Generation

When the sample and all of its child processes have terminated, or after a fixed timeout (currently set to 4 minutes), JACKSTRAWS saves all monitored system calls, network-related data, and tainting information into a log file. Unlike previous work that used behavior graphs for distinguishing between malicious and legitimate programs, we use these graphs to determine the purpose of network connections (and to detect C&C traffic). Thus, we are not interested in the entire activity of the malware program. Instead, we only focus on actions related to network traffic. To this end, we first identify all send and receive operations that operate on a successfully-established network connection. In this work, we focus only on TCP traffic, and a connection is considered successful when the three-way handshake has completed and at least one byte of user data was exchanged. All system calls that are related to a single network connection are added to the behavior graph for this connection. That is, for each network connection that a sample makes, we obtain one behavior graph which captures the host-based activities related to this connection.

For each send operation, we check whether the sent data is tainted. If so, we add the corresponding system call that produced this data to the behavior graph and connect both nodes with an edge. Likewise, for each receive operation, we taint the received data and check if it is later used as input to a system call. If so, we also add this system call to the graph and connect the nodes.

For each system call that is added to the graph in this fashion, we also check backward dependencies (that is, whether the system call has tainted input arguments). If this is the case, we continue to add the system call(s) that are responsible for this data. This process is repeated recursively as long as there are system calls left that have tainted input arguments that are unaccounted for. That is, for every node that is added to our behavior graph, we will also add all parent nodes that produce data that this node consumes. For example, if received data is written to a local file, we will add the corresponding `NtWriteFile` system call to the graph. This write system call will use as one of its arguments a file handle. This file handle is likely tainted, because it was produced by a previous invocation of `NtCreateFile`. Thus, we also add the node that corresponds to this create system call and connect the two nodes with an edge. On the other hand, forward dependencies are not recursively followed to avoid an explosion of the graph size.

Graph Labeling

Nodes and edges that are inserted into the behavior graph are augmented with additional labels that capture more information about the nature of the system calls and the dependencies between nodes. For edges, the label stores either the names of the input or the output arguments of the system calls that are connected

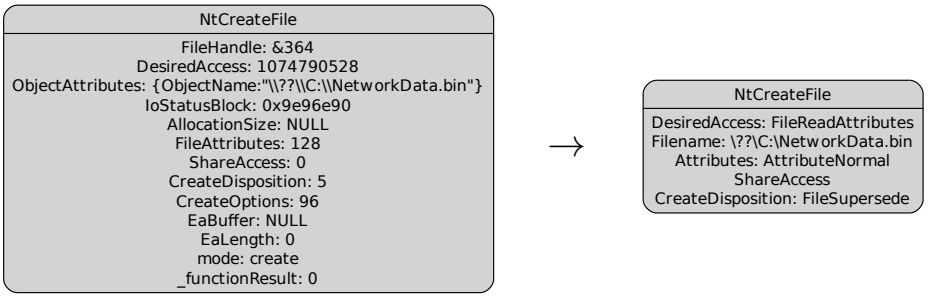


Figure 5.2: Raw behavior graph system call node on the left and its filtered version on the right.

by a data dependency. For nodes, the label stores the system call name and some additional information that depends on the specific type of call.

In the raw version of the behavior graphs generated by Anubis, the node labels contain all system call parameters and their values. Figure 5.2 shows an example of a raw `NtCreateFile` system call node on the left. The problem with storing this plethora of information is that it might influence the subsequent machine learning in a negative way by providing a large bunch of irrelevant information. Therefore, we generalize labels in an additional step by filtering out redundant information that is meaningless for the rest of our system anyway. To do so, we created a manually crafted list of filter expressions that rule out this kind of information. Note that we only need to provide these filter rules for Windows Native API system calls since our system works on this layer. It is thus not necessary to reason about creating filter rules for the various other API layers that are built on top of the Native API. We also use this step to replace numeric parameter constants with more readable strings. The right node in Figure 5.2 shows how a filtered node looks like in case of an `NtCreateFile` system call.

Simplifying Behavior Graphs

One problem we faced during the behavior graph generation was that certain graphs grew very large (in terms of number of nodes), but the extra nodes only carried duplicate information. For example, consider a bot that downloads an executable file. When this file is large, the data will not be read from the network connection by a single `recv` call. Instead, the receive system call might be invoked many times. This heavily depends on the implementation of the bot. In fact, we have observed samples that read network data one byte at a time. Since every system call results

in a node being added to the behavior graph, this can increase the number of nodes significantly. Furthermore, since our system eventually aims at producing generic behavior graph templates that catch the abstract behavior and not implementation details, it makes sense to abstract away from such peculiarities.

To reduce the number of (essentially duplicate) nodes in the graph, we introduce a post-processing step that collapses certain nodes. The purpose of this step is to combine multiple nodes sharing the same label and dependencies. More precisely, for each pair of nodes with an identical label in the behavior graph, we check whether

1. the two nodes share the same set of parent nodes, or
2. the sets of parents and children of one node are subsets of the other, or
3. one node is the only parent of the other.

If this is the case, we use so-called *parallel collapsing* which means we collapse these nodes into a single node and add a special tag *IsMultiple* to the label. Additional incoming and outgoing edges of the aggregated nodes are merged into the new node. The process is repeated until no more collapsing is possible. As an example, consider the case where a write file operation stores data that was previously read from the network by multiple receive calls. In this case, the write system call node will have many identical parent nodes (the receive operations), which all contribute to the buffer that is written. In the post-processing step, these nodes are all merged into a single system call. A beneficial side-effect of node collapsing is that this does not only reduce the number of nodes, but also provides some level of abstraction from the concrete implementation of the malware code and the number of times identical functions are called (as part of a loop, for example). Figure 5.3 shows an example of node collapsing in which multiple `NtWriteFile` system calls are collapsed to one node.

The second collapsing technique is called *sequence collapsing* in which we check whether there are multiple subsequent receive or send nodes in the behavior graph which share common labels and edges with identical labels to the same nodes. If that is the case, this sequence of nodes is collapsed and a special label with the name *IsSequence* is added to the collapsed node. Figure 5.4 shows an example of sequence collapsing.

Filesystem Tainting

We extended the Anubis tainting system to take into account special cases that occur during the behavior graph generation. Specifically, one special case we are interested in is when a monitored bot receives binary data from the Internet (such as an update), stores it on the local hard disk, and then proceeds to execute it. Therefore, the received binary data from the network is tainted and written using

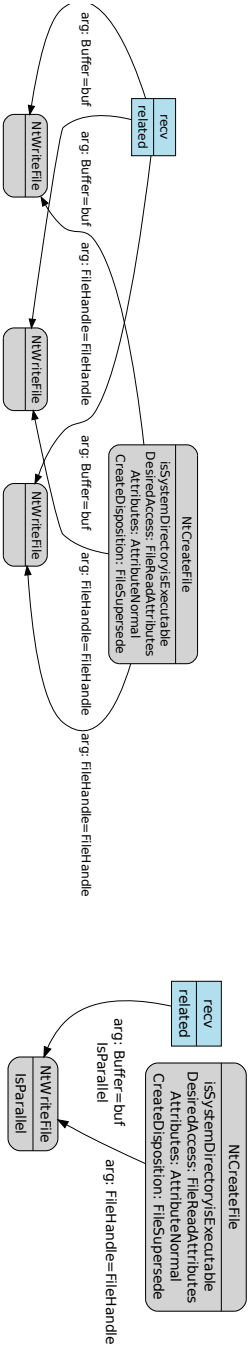


Figure 5.3: Example of a subgraph before (left) and after (right) parallel system call node collapsing.

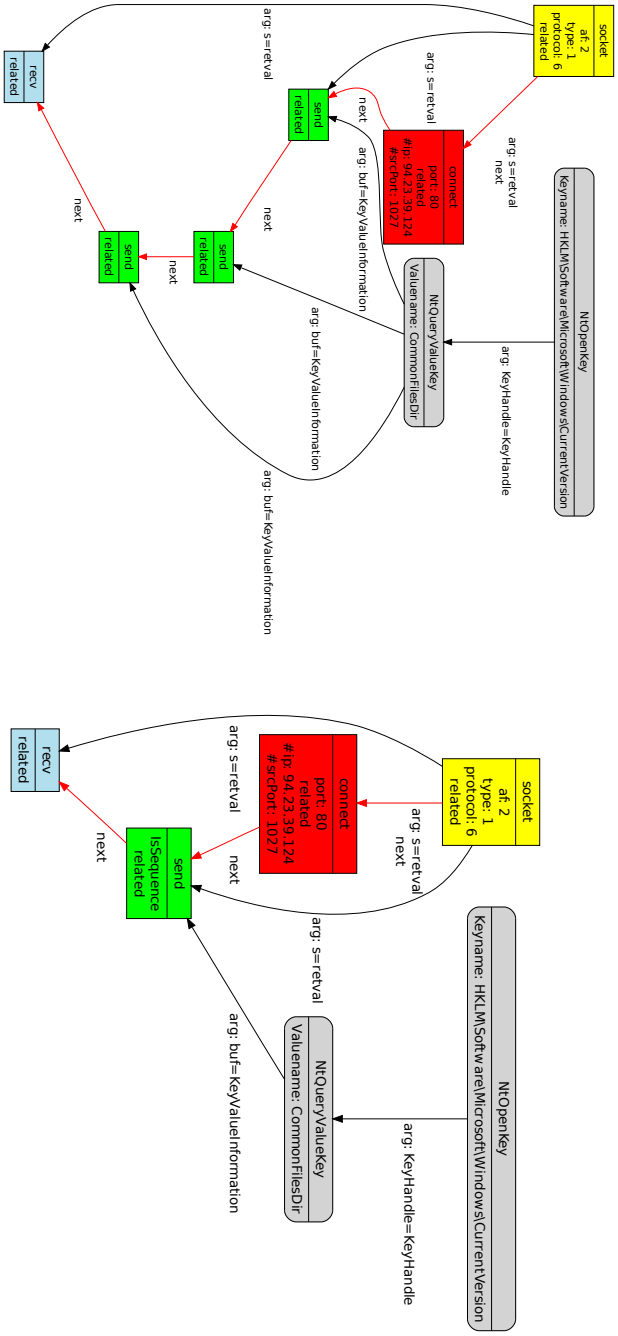


Figure 5.4: Example of a subgraph before (left) and after (right) sequence collapsing.

`NtWriteFile` and subsequently a `NtCreateProcess` system call occurs. However, it may happen that we do not see a connection between the received data and the process creation in the behavior graph because no create process call occurs within the graph since the arguments of the system call are hardcoded and not tainted. We solve this issue by extending the tainting system of Anubis to file operations. That is, whenever tainted data is written to a file, we store corresponding taint labels in a hashmap. When the system later reads from the same file, we propagate the saved taint labels into the read buffers. Taint labels thus remain intact over file write and read operations. This also allows us to detect whether the system executes a file which contains tainted network data. If this is the case, we introduce a special *process* node in the behavior graph and connect the corresponding receive nodes to it.

Summary

The output of the two previous steps is one behavior graph for each network connection that a malware sample makes. Behavior graphs can be used in two ways: First, we can match behavior graphs, produced by running unknown malware samples, against a set of C&C templates that characterize malicious activity. When a template matches, the corresponding network connection can be labeled as command and control.

The second use of behavior graphs is for C&C template generation. For this process, we assume that we know some connections that are malicious and some that are benign. We can then extract the subgraphs from the behavior graphs that are related to known malicious C&C connections and subgraphs that represent benign activity. These two sets of malicious and benign graphs form the input for the template generation process that is described in the following three sections.

5.4.3 Graph Mining, Graph Clustering, and Templating

We confine ourselves to a brief summary of the machine learning algorithms that are employed in the further steps since these results are not part of the thesis. The interested reader can find more details in the Jackstraws paper [87].

The first task is to perform graph mining on the collected malicious behavior graphs. Figure 5.5 illustrates the individual steps that occur in the mining phase. At first, frequent subgraphs in the entire set of behavior graphs are detected using frequent subgraph mining. The resulting set of subgraphs is the maximized if certain conditions are met, i.e., mined subgraphs that are themselves subgraphs of other mined subgraphs are typically discarded. In the last step of the graph mining, the resulting maximal subgraphs are compared against benign behavior graphs using subgraph isomorphism to rule out subgraphs that match benign connections.

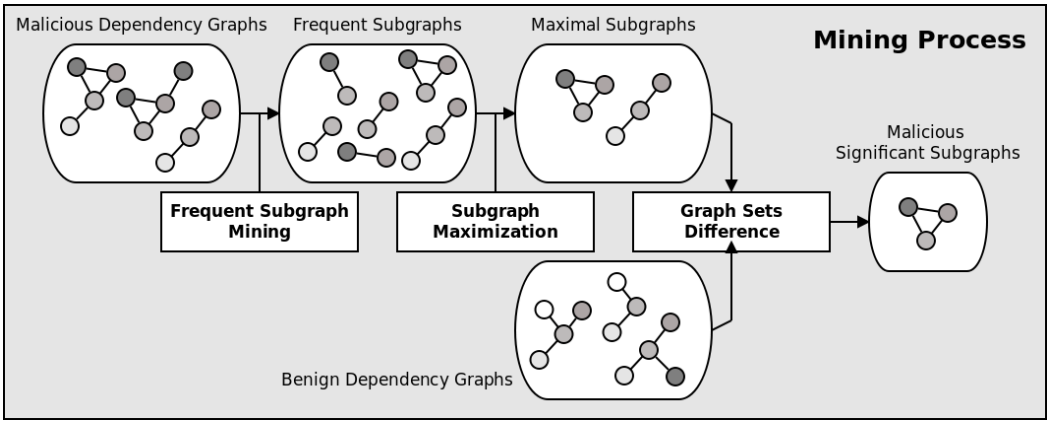


Figure 5.5: Abstract overview of the mining process.

The graph mining step produces various different malicious subgraphs. However, many of these graphs are redundant as they described identical behavior. JACKSTRAWS thus applies graph clustering to identify subgraphs with distinct behaviors out of the large body of mined subgraphs. To achieve this goal, similarity between mined subgraphs is measured by means of the maximum common subgraph and clusters are constructed using bisection clustering. All subgraphs are initially put into the same cluster and the algorithm then divides this cluster until the similarity in each split cluster surpasses a fixed threshold.

The resulting clusters are then used to build the eventual C&C graph templates. Therefore, the weighted minimal common subgraph of all graphs in the same cluster is generated, i.e., the minimal graph in which contains all the cluster's graphs. The algorithm further distinguishes between core and optional nodes. Core nodes and edges are present in all subgraphs of the cluster and the remaining other nodes and edges are marked as optional. Figure 5.6 summarizes the clustering and template generation process.

The resulting templates are matched against behavior graphs using graph isomorphism. The matching algorithm requires all core nodes to be present in the behavior graphs. Further, a score value is computed that depends on the fraction of additionally matching optional nodes.

5.5 Evaluation

Experiments were performed to evaluate JACKSTRAWS both from a quantitative and qualitative perspective. This section describes the evaluation details and results.

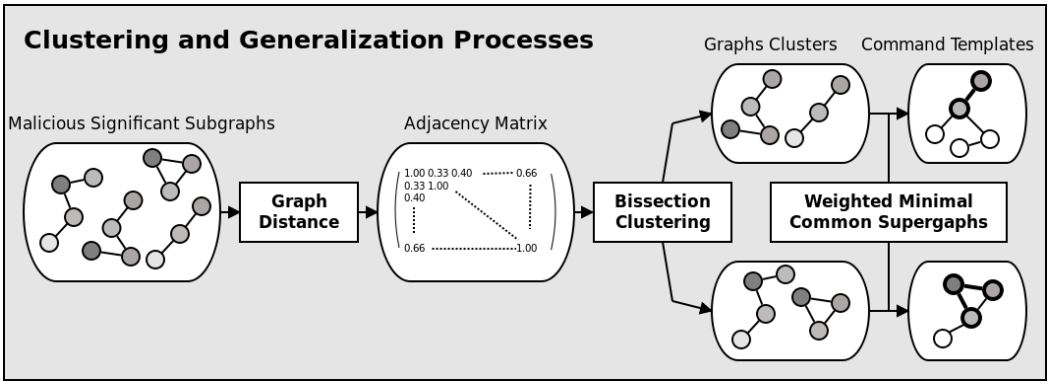


Figure 5.6: Abstract overview of the clustering and generalization processes.

5.5.1 Evaluation Datasets

For the evaluation, our system analyzed a total of 37,572 malware samples. The samples were provided to us by a network security company, who obtained the binaries from recent submission to a public malware analysis sandbox. Moreover, we were only given samples that showed some kind of network activity when run in the sandbox. We were also provided with a set of 385 signatures specifically for known C&C traffic, as well as 162 signatures that characterize known, benign traffic. The company also uses signatures for benign traffic to be able to quickly discard harmless connections that bots frequently make.

To make sure that our sample set covers a wide variety of different malware families, we labeled the entire set with six different anti-virus engines: Kaspersky, F-Secure, BitDefender, McAfee, NOD32, and F-Prot. Using several sources for labeling allows us reduce the possible limitations of a single engine. For every malware sample, each engine returns a label (unless the samples is considered benign) from which we extract the malware family substring. For instance, if one anti-virus engine classifies a sample as *Win32.Koobface.AZ*, then *Koobface* is extracted as the family name. The family that is returned by a majority of the engines is used to label a sample. In case the engines do not agree (and there is no majority for a label, we go through the output of the AV tools in the order that they were mentioned previously and pick the first, non-benign result.

Overall, we identified 745 different malware families for the entire set. The most prevalent families were *Generic* (3756), *EgroupDial* (2009), *Hotbar* (1913), *Palevo* (1556), and *Virut* (1539). 4,096 samples remained without label. Note that *Generic* is not a precise label; many different kinds of malware can be classified as such by AV engines. In summary, the results indicate that our sample set has no significant bias towards a certain malware family. As expected, it covers a rich and diverse set

of malware, currently active in the wild.

In a first step, we executed all samples in JACKSTRAWS. Each sample was executed for four minutes, which allows a sample to initialize and perform its normal operations. This timeout is typically enough to establish several network connections and send/receive data via them. The execution of the 37,572 samples produced 150,030 network connections, each associated with a behavior graph. From these graphs, we removed 19,395 connections in which the server responded with an error (e.g., an HTTP request with a 404 "Not Found" response). Thus, we used a total of 130,635 graphs produced by a total of 33,572 samples for the evaluation.

In the next step, we applied our signatures to the network connections. This resulted in 16,535 connections that were labeled as malicious (known C&C traffic, 12.7%) and 16,082 connections that were identified as benign (12.3%). The malicious connections were produced by 9,108 samples, while the benign connections correspond to 7,031 samples. The remaining 98,018 connections (75.0%) are unknown. The large fraction of unknown connections is an indicator that it is very difficult to develop a comprehensive set of signatures that cover the majority of bot-related C&C traffic. In particular, there was at least one unclassified connection for 31,671 samples. Note that the numbers of samples that produced malicious, benign, and unknown traffic add up to more than the total number of samples. This is because some samples produced both malicious and benign connections. This underlines that it is difficult to pick the important C&C connections among bot traffic.

Of course, not all of the 385 malicious signatures produced matches. In fact, we observed only hits from 78 C&C signatures, and they were not evenly distributed. A closer examination revealed that the signature that matched the most number of network connections is related to *Palevo* (4,583 matches), followed by *Ramnit* (3,896 matches) and *Koobface* (2,690 matches).

5.5.2 Template Generation

In order to increase the overall quality of the malicious and benign test mining graphs, a pre-filtering step was done. Therefore, behavior graphs that contained only network-related system calls or generally too few nodes were ruled out. This resulted in a final mining set of 10,801 malicious and 12,367 benign behavior graphs.

Both sets were then further split into a training set and a test set. To this end, we randomly picked a number of graphs for the training set, while the remaining ones were set aside as a test set. More precisely, for the malicious graphs, we kept 6,539 graphs (60.5%) for training and put 4,262 graphs (39.5%) into the test set. For the benign graphs, we kept 8,267 graphs (66.8%) for training and put 4,100 graphs (33.2%) into the test set. We used these malicious and benign training sets as input for our template generation algorithm. This resulted in 417 C&C templates that

JACKSTRAWS produced. The average number of nodes in a template was 11, where 6 nodes were part of the core and 5 were optional.

5.5.3 Detection Accuracy

In the next step, we wanted to assess whether the generated templates can accurately detect activity related to command and control traffic without matching benign connections. To this end, we ran two experiments. First, we evaluated the templates on the graphs in the test set (which correspond to known C&C connections). Then, we applied the templates to graphs associated with unknown connections. This allows us to determine whether the extracted C&C templates are generic enough to allow detection of previously-unknown C&C traffic (for which no signature exists).

Experiment 1: Known C&C connections

For the first experiment, we made use of the test set that was previously set aside. More precisely, we applied our 417 templates to the behavior graphs in the test set. This test set contained 4,262 connections that matched C&C signatures and 8,267 benign connections.

Our results show that JACKSTRAWS is able to successfully detect 3,476 of the 4,262 malicious connections (81.6%) as command and control traffic. Interestingly, the test set also contained malware families that were absent from the malicious training set. 51.7% of the malicious connections coming from these families were successfully detected, accounting for 0.4% of all detections. While the detection accuracy is high, we explored false negatives (i.e., missed detections) in more detail. Overall, we found three reasons why certain connections were not correctly identified:

First, in about half of the cases, detection failed because the bot did not complete its malicious action after it received data from the C&C server. Incomplete behavior graphs can be due to a timeout of the dynamic analysis environment, or an invalid configuration of the host to execute the received command properly.

Second, the test set contained a significant number of *Adware* samples. The behavior graphs extracted from these samples are very similar to benign graphs; after all, *Adware* is in a grey area different from malicious bots. Thus, all graphs potentially covering these samples are removed at the end of the mining process, when compared to the benign training sets.

The third reason for missed detections is malicious connections that are only seen a few times (possibly only in the test set). According to the AV labels, our data set covers 745 families (and an additional 4,096 samples that could not be labeled). Thus, certain families are rare in the data set. When a specific graph is only present a few times (or not at all) in the training set, it is possible that all of its subgraphs are below the mining threshold. In this case, we do not have a template that covers

this activity.

JACKSTRAWS also reported 7 benign graphs as malicious out of 4,100 connections in the benign test set: a false positive rate of 0.2%. Upon closer examination, these false positives correspond to large graphs where some Internet caching activity is observed. These graphs accidentally triggered four weaker templates with few core and many optional nodes.

Overall, our results demonstrate that the host-based activity learned from a set of known C&C connections is successful in detecting other C&C connections that were produced by a same set of malware families, but also in detecting five related families that were only present in the test set. In a sense, this shows that C&C templates have a similar detection capability as manually-generated, network-based signatures.

Experiment 2: Unknown connections

For the next experiment, we decided to apply our templates to the graphs that correspond to unknown network traffic. This should demonstrate the ability of JACKSTRAWS to detect novel C&C connections within protocols not covered by any network-level signature.

When applying our templates to the 98,018 unknown connections, we found 9,464 matches (9.7%). We manually examined these connections in more detail to determine whether the detection results are meaningful. The analysis showed that our approach is promising; the vast majority of connections that we analyzed had clear indications of C&C activity. With the help of the anti-virus labels, we could identify 193 malware families which were *not* covered by the network signatures. The most prevalent new families were *Hotbar* (1984), *Pakes* (871), *Kazy* (107), and *LdPinch* (67). Furthermore, we detected several new variants of known bots that we did not detect previously because their network fingerprint had changed and, thus, none of our signatures matched. Nevertheless, JACKSTRAWS was able to identify these connections due to matched templates. In addition, the manual analysis showed a low number of false positives. In fact, we only found 27 false positives out of the 9,464 matches, all of them being HTTP connections.

When comparing the number of our matches with the total number of unknown connections, the results may appear low at first glance. However, not all connections in the unknown set are malicious. In fact, 10,524 connections (10.7%) do not result in any relevant host-activity at all (the graphs only contain network-related system calls such as `send` or `connect`). For another 13,676 graphs (14.0%), the remote server did not send any data. For more than 7,360 HTTP connections (7.5%), the server responded with status code 302, meaning that the requested content had moved. In this case, we probably cannot see any interesting behavior to match. In a few hundred cases, we also observed that the timeout of JACKSTRAWS inter-

rupted the analysis too early (e.g., the connection downloaded a large file). In these cases, we usually miss some of the interesting behavior. Thus, almost 30 thousand unknown connections can be immediately discarded as non-C&C traffic.

Furthermore, the detection results of 9,464 new C&C connections for JACKSTRAWS need to be compared with the total number of 16,535 connections that the entire signature set was able to detect. Our generalized templates were able to detect almost 60% more connections than hundreds of hand-crafted signatures. Note that our C&C templates do not inspect network traffic at all. Thus, they can, by construction, detect C&C connections regardless of whether the malware uses encryption or not, something not possible with network signatures.

5.5.4 Template Quality

The previous section has shown that our C&C templates are successful in identifying host-based activity related to both known and novel network connections. We also manually examined several templates in more detail to determine whether they capture activity that a human analyst would consider malicious.

JACKSTRAWS was able to extract different kinds of templates. More precisely, out of the 417 templates, more than a hundred templates represent different forms of information leakage. The leaked information is originally collected from dedicated registry keys or from specific system calls (e.g., computer name, Windows version and identifier, Internet Explorer version, current system time, volume ID of the hard disk, or processor information). About fifty templates represent executable file downloads or updates of existing files. Additional templates include process execution: downloaded data that is injected into a process and then executed. Five templates also represent complete download and execute commands. The remaining templates cover various other malicious activities, including registry modifications ensuring that the sample is started on certain events (e.g., replacing the default executable file handler for Windows Explorer) and for hiding malware activity (e.g., clearing the MUICache).

We also found 20 "weak" templates (out of 417). These templates contain a small number of nodes and do not seem related to any obvious malicious activity. However, these templates did not trigger any false positive in the benign test set. This indicates that they still exhibit enough discriminative power with regards to our malicious and benign graph sets.

5.5.5 Template Examples

We manually examined C&C templates to determine whether they capture activity that a human analyst would consider malicious. We now present two examples that were automatically generated by JACKSTRAWS.

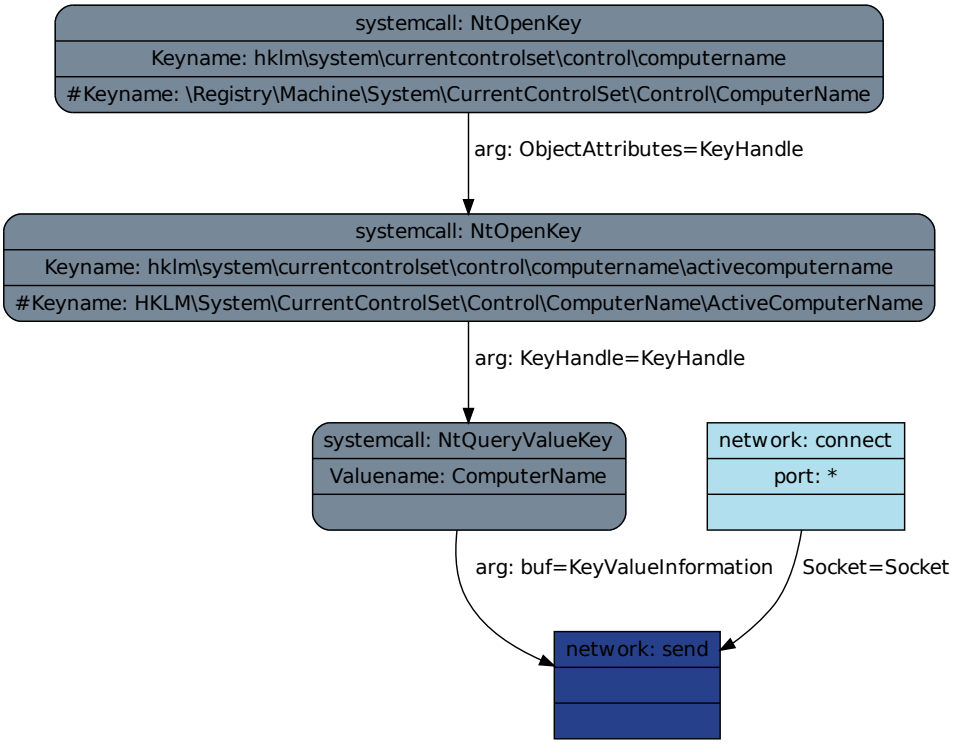


Figure 5.7: Template that describes leaking of sensitive data. Darker nodes constitute the template core, whereas lighter ones are optional.

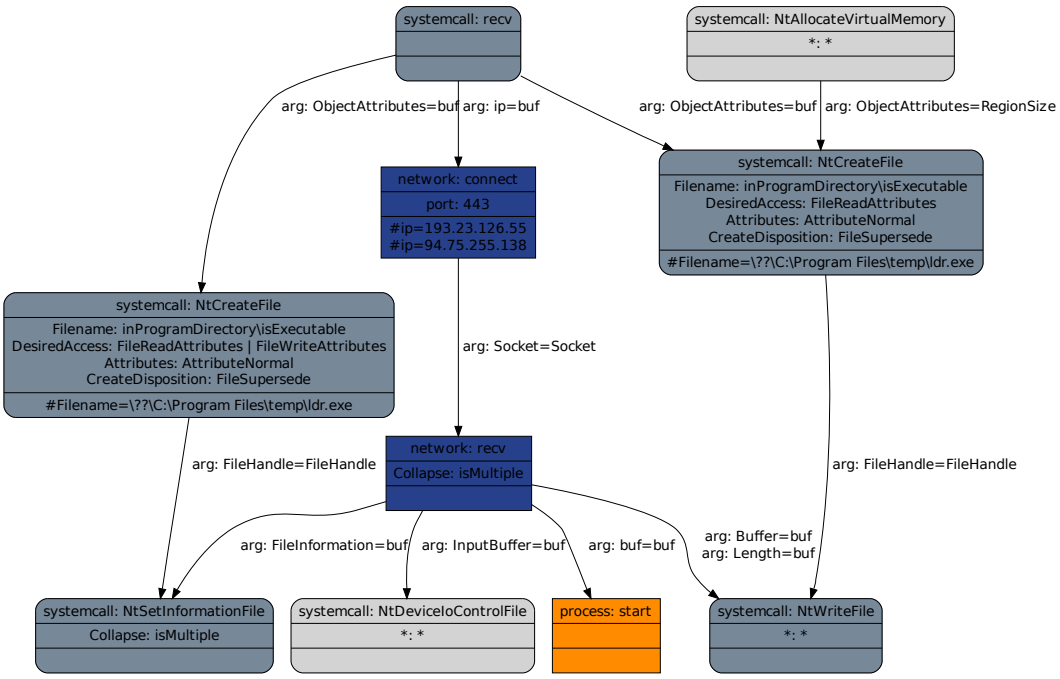


Figure 5.8: Template that describes the download and execute functionality of a bot: an executable file is created, its content is downloaded from the network, decoded, written to disk, its information is modified before being executed. In the *NtCreateFile* node, the file name *ldr.exe* is only mentioned as a comment. Comments help a human analyst when looking at a template, but they are ignored by the matching.

Figure 5.7 shows a template we extracted from bots that use a proprietary, binary protocol for communicating with the C&C server. The behavior corresponds to some kind of *information leakage*: the samples query the registry for the computer name and send this information via the network to a server. We consider this a malicious activity, which is often used by bots to generate a unique identifier for an infected machine. In the network traffic itself this activity cannot be easily identified, since the samples use their own protocol.

As another example, consider the template shown in Figure 5.8. This template corresponds to the *download & execute* behavior, i.e., data is downloaded from the network, written to disk, and then executed. The template describes this specific behavior in a generic way.

5.6 Conclusion and Future Work

In this chapter, we have presented behavior graphs as a new improved way to model network connection that allow distinguishing malicious C&C bot connections from benign connections. Therefore, we combined network-level with host-level information to introduce semantic coherences between system call and data flows between their parameters. This allowed us to overcome the weaknesses of approaches that rely either on host-based or on network-based only models. We implemented our approach using the Anubis analysis system for proprietary Windows operating systems. With the help of machine learning techniques, it is possible to use these behavior graphs by mining, clustering, and generically creating templates that allow detecting malicious C&C connections in an abstract way.

An interesting topic for future work is to compare JACKSTRAWS's results against manually created behavior graph templates for C&C connections. We are interested to see whether this allows us to achieve even better detection rates and lower false positives.

6

Conclusion

In this thesis, we presented several approaches to analyze and retrofit proprietary software systems. We therefore concerned ourselves with four subtopics in this field. We show how to extract algorithms from proprietary software and how to overcome various unique challenges that emerge when it comes to retrofitting system with improved security measures.

The first contribution is the disclosure of the secret satellite telephony GMR-1 algorithm. We presented a generic strategy to identify the unknown encryption algorithms within a satphone firmware. We then used this approach to reverse engineer the searched-for algorithms, which were later proven to be vulnerable to attacks already mounted against the related GSM mobile telephony standard. The results of our work confirm our initial thesis that communication systems should not rely on the security by obscurity principle regarding cryptographic algorithms.

In the next chapter, we showed that the assumptions of current kernelspace ASLR implementations are insufficient and can be undermined by a local attacker using side channel timing attacks. We therefore extracted the proprietary kernelspace implementation of Windows and developed three independent attacks that allow an attacker to reconstruct (parts of) the kernelspace layout, which in turn gives her the possibility to mount arbitrary ROP attacks. We also presented a mitigation approach by retrofitting the Windows exception handler to render these kinds of attacks impossible.

We also presented the design and implementation of the runtime components of the MoCFI tool that provides CFI for smartphones running the proprietary iOS operating system to thwart the exploitation of software vulnerabilities in apps. We showed how to tackle various challenges that emerge due to the lack of source code information of the protected applications. Furthermore, we presented PSiOS, an extension of the original CFI framework that allows to specify individual policy rules to restrict the access to sensitive data sources by arbitrary applications. The evaluation of both systems shows that our tools are applicable to a large variety of

popular applications and the runtime overhead remains within reasonable bounds.

Finally, we described a new approach to model malicious C&C connections from bots in the form of behavior graphs. Behavior graphs overcome the inherent weaknesses of previous similar systems which confine themselves to either network- or host-based approaches and thus produce high false positive rates or high false negative. We combine both worlds and show how behavior graphs can be generated for the Windows operating system by monitoring the proprietary native API. The behavior graph generation was embedded into the JACKSTRAWS framework which is able to generate general templates on the basis of these graphs using machine learning techniques. The evaluation shows that behavior graphs are an effective model and that they allows to successfully distinguish malicious from benign connections.

List of Acronyms

ASLR Address Space Layout Randomization . . . 4
AST abstract syntax tree . . . 76
BB basic block . . . 79
CFG control flow graph . . . 5
CFI control flow integrity . . . iii
CISC complex instruction set computer . . . 15
CPD change point detection . . . 63
CSE code signing enforcement . . . 82
DDoS distributed denial of service . . . 113
DEP Data Execution Prevention . . . 88
DSP digital signal processor . . . 4
EM execution monitoring . . . 85
ETSI European Telecommunications Standards Institute . . . 10
GMR-1 GEO Mobile Radio Interface
GSM Global System for Mobile Communications . . . 9
JIT just in time . . . 82
LOC lines of code . . . 93
MMU memory management unit . . . 16
MoCFI Mobile CFI . . . 5
PSiOS Privacy and Security for iOS devices
RISC reduced instruction set computer . . . 15
ROP return-oriented programming . . . 4

List of Figures

List of Tables

List of Listings

Bibliography

[1] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. `http://joshaas.net/linux/linux_cpu_scheduler.pdf`, 2005.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[3] M. Abadi, M. Budiu, U. Erlingsson, G. C. Necula, and M. Vrable. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[4] W. A.-K. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[5] O. Aciiçmez. Yet another MicroArchitectural Attack: exploiting I-Cache. In *ACM Workshop on Computer Security Architecture (CSAW)*, 2007.

[6] O. Aciiçmez, W. Schindler, and Çetin Kaya Koç. Improving Brumley and Boneh timing attack on unprotected SSL implementations. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[7] O. Aciiçmez, B. B. Brumley, and P. Grabher. New Results on Instruction Cache Attacks. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2010.

[8] S. Adair. Pushdo DDoS'ing or Blending In? `http://www.shadowserver.org/wiki/pmwiki.php/Calendar/20100129`, January 2010.

[9] P. Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security Symposium*, 2010.

[10] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10. USENIX Association, 2010.

[11] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium*, 2009.

[12] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 49(14), 1996.

[13] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), 2001.

[14] Apple. The App Sandbox. `http://developer.apple.com/library/ios/#documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/EnablingAppSandbox.html`, 2013.

[15] Apple. App Review Guidelines. `https://developer.apple.com/appstore/guidelines.html`, 2013.

[16] Apple. Max OS X Source Code. `http://www.opensource.apple.com/`, 2013.

[17] Apple Inc. Objective-C Runtime Programming Guide. `http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/ObjCRuntimeGuide.pdf`, 2009.

[18] Apple Inc. Manual Page of `dyld` - the dynamic link editor. `http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/dyld.1.html`, 2011.

[19] Apple Inc. Manual Page of `mmap` - allocate memory, or map files or devices into memory. `http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/mmap.2.html`, 2011.

[20] Apple Inc. Entitlement key reference. `http://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/EnablingAppSandbox.html#//apple_ref/doc/uid/TP40011195-CH4-SW1`, 2011.

[21] Apple Inc. Designing for App Sandbox. `http://developer.apple.com/library/mac/#documentation/Security/Conceptual/AppSandboxDesignGuide/DesigningYourSandbox/DesigningYourSandbox.html`, 2013.

[22] ARM Limited. Procedure Call Standard for the ARM Architecture. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf`, 2009.

[23] ARM Limited. ARM Architecture Reference Manual, 2010. `http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html`.

[24] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000.

[25] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-Only Cryptanalysis of GSM encrypted communication. In *International Crytology Conference (CRYPTO)*, pages 600–616, 2003.

[26] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. *Journal of Cryptology*, 21, March 2008.

[27] M. Basseville and I. V. Nikiforov. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, 1993.

[28] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[29] Benedikt Driessen, Ralf Hund, Carsten Willems, Christof Paar and Thorsten Holz. An analysis of the GMR-1 and GMR-2 standards, 2012. `http://gmr.crypto.rub.de`.

[30] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security Symposium*, 2003.

[31] A. Biryukov, A. Shamir, and D. Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Fast Software Encryption (FSE)*, 2000.

[32] blexim. Basic Integer Overflows. *Phrack Magazine*, 60(10), 2002.

[33] A. Bogdanov, T. Eisenbarth, and A. Rupp. A Hardware-Assisted Realtime Attack on A5/2 Without Precomputations. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[34] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev. Address Space Randomization for Mobile Devices. In *ACM Conference on Wireless Network Security (WiSec)*, 2011.

[35] J. Bonneau and I. Mironov. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2006.

[36] M. Briceno, I. Goldberg, and D. Wagner. A pedagogical implementation of the GSM A5/1 and A5/2 "voice privacy" encryption algorithms, 1999. Originally published at http://www.scard.org, mirror at `http://cryptome.org/gsm-a512.htm`.

[37] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. In *USENIX Security Symposium*, 2003.

[38] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[39] Charles Miller and Dion Blazakis and Dino Dai Zovi and Stefan Esser and Vincenzo Iozzo and Ralf-Phillipp Weinmann. *iOS Hacker's Handbook*, page 211. John Wiley & Sons, Inc., 2012.

[40] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy*, 2010.

[41] T. Chiueh and F.-H. Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[42] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[43] M. Christodorescu and S. Jha. Testing Malware Detectors. In *ACM Int. Symp. on Software Testing & Analysis (ISSTA)*, 2004.

[44] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Meeting of the European Software Engineering Conf. & the SIGSOFT Symp. Foundations of Software Engineering*, 2007.

[45] M. Conover. w00w00 on Heap Overflows, 1999.

[46] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *USENIX Workshop Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, 2005.

[47] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[48] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard™: Protecting Pointers From Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, 2003.

[49] D. Dagon, G. Gu, C. Lee, and W. Lee. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conf. (ACSAC)*, 2007.

[50] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[51] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: Malware Analysis Via Hardware Virtualization Extensions. In *ACM Conf. Computer & Communications Security (CCS)*, 2008.

[52] B. Driessen, R. Hund, C. Willems, C. Paar, and T. Holz. Don't Trust Satellite Phones: A Security Analysis of Two Satphone Standards. In *IEEE Symposium on Security and Privacy*, pages 128–142. IEEE Computer Society, 2012.

[53] J. Duart. Objective-C helper script. `https://github.com/zynamics/objc-helper-plugin-ida`, 2010.

[54] T. Durden. Bypassing PaX ASLR Protection. *Phrack Magazine*, 59(9), 2002.

[55] A. Edwards, A. Srivastava, and H. Vo. Vulcan Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[56] A. Edwards, H. Vo, A. Srivastava, and A. Srivastava. Vulcan binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[57] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Symposium on Network and Distributed System Security (NDSS)*, 2011.

[58] M. Engelberth, F. C. Freiling, J. Göbel, C. Gorecki, T. Holz, R. Hund, P. Trinius, and C. Willems. Das Internet-Malware-Analyse-System (InMAS). *Datenschutz und Datensicherheit*, 35:247–252, 2011.

[59] U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms*, NSPW '99, pages 87–95. ACM, 2000.

[60] ETSI. ETSI TS 101 376-3-2 V1.1.1 (2001-03); GEO-Mobile Radio Interface Specifications; Part 3: Network specifications; Sub-part 2: Network Architecture; GMR-1 03.002, 2001.

[61] ETSI. ETSI TS 101 376-3-9 V1.1.1 (2001-03); GEO-Mobile Radio Interface Specifications; Part 3: Network specifications; Sub-part 9: Security related Network Functions; GMR-1 03.020, 2001.

[62] A. Eustace and A. Srivastava. ATOM: a flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*. USENIX Association, 1995.

[63] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, 1999.

[64] E. W. Felten and M. A. Schneider. Timing Attacks on Web Privacy. In *ACM Conference on Computer and Communications Security (CCS)*, 2000.

[65] P. Fogla and W. Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *ACM Conf. Computer & Communications Security (CCS)*, 2006.

[66] P. Fogla, M. I. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic Blending Attacks. In *Usenix Security Symp.*, 2006.

[67] S. Forrest, S. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symp. Security & Privacy*, 1996.

[68] M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, 2001.

[69] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE Symp. Security & Privacy*, 2010.

[70] F. C. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *European Symp. Research in Computer Security (ESORICS)*, 2005.

[71] gera. Advances in Format String Exploitation. *Phrack Magazine*, 59(12), 2002. URL `http://www.phrack.com/issues.html?issue=59&id=7`.

[72] Giuffrida, Cristiano and Kuijsten, Anton and Tanenbaum, Andrew S. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12. USENIX Association, 2012.

[73] GlobalCom. Inmarsat Isatphone Pro World Wide Coverage Map, 2012. `http://www.globalcomsatphone.com/support9.html`.

[74] J. Goebel and T. Holz. Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. In *USENIX Workshop Hot Topics in Understanding Botnets (HotBots)*, 2007.

[75] F. Gröbert, C. Willems, and T. Holz. Automated Identification of Cryptographic Primitives in Binary Programs. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[76] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *USENIX Security Symp.*, 2006.

[77] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *USENIX Security Symp.*, 2008.

[78] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *Symp. Network & Distributed System Security (NDSS)*, 2008.

[79] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy*, 2011.

[80] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the Droids you're looking for: Retrofitting Android to protect data from imperious applications. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2011.

[81] R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security Symposium*, 2009.

[82] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.

[83] Intel. TLBs, Paging-Structure Caches, and Their Invalidation. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2012.

[84] Intel Corporation. Intel: 64 and IA-32 Architectures Software Developer's Manual, 2007. http://www.intel.com/products/processor/manuals/index.htm.

[85] International Secure Systems Lab. Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org, 2011.

[86] Invisible Things Lab. From Slides to Silicon in 3 Years! http://theinvisiblethings.blogspot.de/2011/06/from-slides-to-silicon-in-3-years.html, 2011.

[87] G. Jacob, R. Hund, C. Kruegel, and T. Holz. JACKSTRAWS: Picking Command and Control Connections from Bot Traffic. In *USENIX Security Symposium*. USENIX Association, 2011.

[88] Jim Geovedi and Raoul Chiesa. Hacking a Bird in the Sky. In *HITBSecConf, Amsterdam*, 2011.

[89] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. In *USENIX Symp. Networked Systems Design & Implementation (NSDI)*, 2009.

[90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, page 118. Elsevier, Inc., 2012.

[91] M. Jurczyk. Windows Security Hardening Through Kernel Address Protection. http://j00ru.vexillium.org/?p=1038, 2011.

[92] A. Kalafut, A. Acharya, and M. Gupta. A Study of Malware in Peer-to-Peer Networks. In *ACM SIGCOMM Conf. Internet Measurement*, 2006.

[93] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection Against return-to-user Attacks. In *USENIX Security Symposium*, 2012.

[94] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[95] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symp.*, 2004.

[96] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security Symposium*, 2002.

[97] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *International Crytology Conference (CRYPTO)*, 1996.

[98] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *USENIX Security Symp.*, 2009.

[99] C. Kreibich and J. Crowcroft. Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1), 2004.

[100] W. Lee, S. J. Stolfo, and K. W. Mok. A Data Mining Framework for Building Intrusion Detection Models. In *IEEE Symp. Security & Privacy*, 1999.

[101] D. Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. `http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`, 2012.

[102] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast Signature Generation for Zero-day PolymorphicWorms with Provable Attack Resilience. In *IEEE Symp. Security & Privacy*, 2006.

[103] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, 2005.

[104] G. Maral and M. Bousquet. *Satellite Communications Systems: Systems, Techniques and Technology.* John Wiley & Sons, 5 edition, 2009.

[105] D. Matolak, A. Noerpel, R. Goodings, D. Staay, and J. Baldasano. Recent progress in deployment and standardization of geostationary mobile satellite systems. In *Military Communications Conference (MILCOM)*, 2002.

[106] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[107] Microsoft. Description of Performance Options in Windows. `http://support.microsoft.com/kb/259025/en-us`, 2007.

[108] Microsoft. The /SAFESEH compiler flag. http://msdn.microsoft.com/en-us/library/9a89h429%28v=vs.80%29.aspx, 2011.

[109] K. Mowery, S. Keelveedhi, and H. Shacham. Are AES x86 Cache Timing Attacks Still Feasible? In *ACM Cloud Computing Security Workshop (CCSW)*, 2012.

[110] N. Seriot. SpyPhone. https://github.com/nst/SpyPhone, 2011.

[111] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[112] J. Newsom, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symp. Security & Privacy*, 2005.

[113] Noé Lutz. Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic. Master's thesis, ETH Zürich, July 2008.

[114] OsmocomGMR. Thuraya SO-2510, 2013. URL http://gmr.osmocom.org/trac/wiki/Thuraya_SO2510.

[115] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting security in cots software with binary rewriting. In *SEC*, IFIP Advances in Information and Communication Technology, 2011.

[116] PaX Team. http://pax.grsecurity.net/.

[117] PaX Team. Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[118] S. Peisert, M. Bishop, S. Karin, and K. Marzullo. Analysis of Computer Intrusions Using Sequences of Function Calls. *IEEE Trans. Dependable Secur. Comput.*, 4(2), 2007.

[119] C. Percival. Cache Missing for Fun and Profit. http://www.daemonology.net/hyperthreading-considered-harmful/, 2005.

[120] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. I. Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symp. Security & Privacy*, 2006.

[121] R. Perdisci, W. Lee, and N. Feamster. Behavioral Clustering of HTTP-based Malware and Signature Generation Using Malicious Network Traces. In *USENIX Symp. Networked Systems Design & Implementation (NSDI)*, 2010.

[122] S. Petrovic and A. Fuster-Sabater. Cryptanalysis of the A5/2 Algorithm. Technical report, Information Security Institute Serrano, 2000. `http://eprint.iacr.org/2000/052`.

[123] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symp.*, 2008.

[124] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Internet Measurement Conference (IMC)*, 2006.

[125] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[126] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*. USENIX Association, 1997.

[127] M. Russinovich. Inside the Windows Vista Kernel: Part 3. `http://technet.microsoft.com/en-us/magazine/2007.04.vistakernel.aspx`, 2007.

[128] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, pages 30–50, 2000.

[129] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symp. Security & Privacy*, 2010.

[130] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[131] M. J. Schwartz. iOS Social Apps Leak Contact Data. `http://www.informationweek.com/news/security/privacy/232600490`, 2012.

[132] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *In Proceedings of the 2001 Workshop on Binary Translation (WBT-2001*, 2001.

[133] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[134] H. Shacham, M. Page, B. Paff, E. jin Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[135] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *USENIX Symp. Operating Systems Design & Implementation (OSDI)*, 2004.

[136] Solar Designer. "return-to-libc" attack. Bugtraq, 1997.

[137] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Int. Conf. Information Systems Security (ICISS)*, 2008.

[138] D. X. Song, D. Wagner, and X. Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security Symposium*, 2001.

[139] E. Stinson and J. C. Mitchell. Characterizing Bots' Remote Control Behavior. In *Conf. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2007.

[140] B. Stone-Gross, A. Moser, C. Kruegel, and E. Kirda. FIRE: FInding Rogue nEtworks. In *Annual Computer Security Applications Conf. (ACSAC)*, 2009.

[141] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the Memory Secrecy Assumption. In *European Workshop on System Security (EuroSec)*, 2009.

[142] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Trans. Comput. Syst.*, 23(1), 2005.

[143] Symantec. W32.Stuxnet Dossier. `http://www.symantec.com/connect/blogs/w32stuxnet-dossier`, 2010.

[144] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[145] Texas Instruments. The OMAP 5910 Platform, 2012. URL `http://www.ti.com/product/omap5910`.

[146] Texas Instruments. System Initialization for the OMAP5910 Device, 2012. URL `http://www.ti.com/product/omap5910`.

[147] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.*, 23(2), Jan. 2010.

[148] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), Volumes 1 and 2*, 2005.

[149] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5), 1993.

[150] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[151] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[152] M. Weiss, B. Heinz, and F. Stumpf. A cache timing attack on aes in virtualization environments. In *Financial Cryptography and Data Security (FC)*, 2012.

[153] H. Welte. Anatomy of contemporary GSM cellphone hardware, 2010. URL `http://laforge.gnumonks.org/papers/gsm_phone-anatomy-latest.pdf`.

[154] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. Psios: Bring your own privacy & security to ios devices. In *8th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2013)*, May 2013.

[155] A. M. White, A. R. Matthews, K. Z. Snow, and F. Monrose. Phonotactic Reconstruction of Encrypted VoIP Conversations: Hookt on Fon-iks. In *IEEE Symposium on Security and Privacy*, 2011.

[156] C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards Automated Dynamic Binary Analysis. *IEEE Security & Privacy*, 5(2), 2007.

[157] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan. Down to the bare metal: using processor features for binary analysis. In *ACSAC*, pages 189–198, 2012.

[158] C. Willems, R. Hund, and T. Holz. CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring. Technical Report TR-HGI-2012-002, Ruhr-University Bochum, 2012.

[159] S. Winwood and M. Chakravarty. Secure Untrusted Binaries – Provably! In *3rd international workshop on formal aspects in security and trust*, 2006.

[160] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. Spot Me if You Can: Uncovering Spoken Phrases in Encrypted VoIP Conversations. In *IEEE Symposium on Security and Privacy*, 2008.

[161] D. Wright. Reaching out to remote and rural areas: Mobile satellite services and the role of Inmarsat. *Telecommunications Policy*, 19(2):105 – 116, 1995.

[162] P. Wurzinger, L. Bilge, T. Holz, J. Göbel, C. Kruegel, and E. Kirda. Automatically generating models for botnet detection. In *European Symp. Research in Computer Security (ESORICS)*, 2009.

[163] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent Runtime Randomization for Security. In *Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[164] T.-F. Yen and M. K. Reiter. Traffic Aggregation for Malware Detection. In *Conf. Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2008.

[165] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *IEEE Symposium on Security and Privacy*, 2011.

[166] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[167] D. D. Zovi. Apple iOS Security Evaluation: Vulnerability Analysis and Data Encryption. In *Black Hat USA*, 2011.