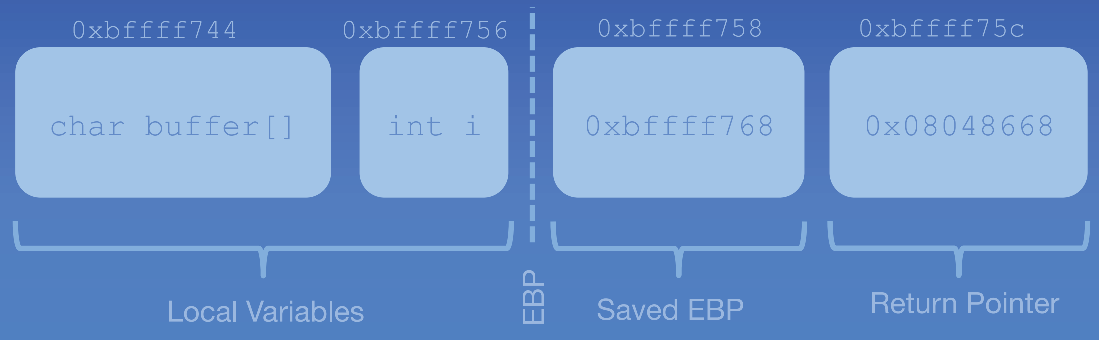


As early as 1972, it was discovered that improperly checked input data processed by a program may change its behaviour on behalf of an attacker. The Morris worm from 1988 showed that buffer overflows can be performed very purposefully as to arbitrarily change the behaviour of a program. As of 2016, this category is still one of the most prevalent types of attacks according to the MITRE Common Weakness Enumeration (CWE). It enables attackers to control input, computations and output to achieve their desired goal, such as changing a victim's online banking transaction, installing e-mail spamming servers or blackmailing victims by encrypting their valuable files.

This thesis presents four novel approaches that all mitigate or prevent this type of imposition of malicious behaviour on otherwise benign code.

Stefan Nürnberger

Mitigating the Imposition of Malicious Behaviour on Code

Mitigating the Imposition of Malicious Behaviour on Code

Stefan Nürnberger

A dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

Universität des Saarlandes

Saarbrücken, 2016

Day of Colloquium	20 July 2017
Dean of the Faculty	Univ.-Prof. Dr. Frank-Olaf Schreyer
Chair of the Committee	Univ.-Prof. Dr. Andreas Zeller
Reporters	
First reviewer	Univ.-Prof. Dr. Michael Backes
Second reviewer	Univ.-Prof. Dr. Christian Rossow
Third reviewer	Univ.-Prof. Dr. Thorsten Holz
Academic Assistant	Dr.-Ing. Sven Bugiel

Deutsche Zusammenfassung

Gutartige Programme, welche sich in schädliche verwandeln lassen, stellen eine größere Bedrohung dar, als Programme, die von vornherein bösartig sind. Während bösartige Programme immerhin die klare Absicht des Diebstahls oder der Manipulation von Daten haben, hat ein gutartiges Programm in aller Regel einen Nutzen für den Anwender. Wenn nun aber ein Programmierfehler dazu führen kann, plötzlich das Verhalten eines Programms zu verändern, bleibt dies von traditionellen Virenscanner völlig ungeachtet, weil diese bloß per se schädliche Programme erkennen. Hinzu kommt, dass Software oft weit verbreitet ist und in identischer Form auf Millionen von Computern Verwendung findet – ein gefundenes Fressen, um Sicherheitslücken millionenfach auszunutzen.

Bereits 1972 zeigten Forscher, dass nicht ordnungsgemäß verarbeitete Eingaben eines Programmes dessen Verhalten beliebig ändern können [P1, S.61]. Programmierfehler, wie beispielsweise das Überschreiten eines Puffers, könnten nachgelagerte Daten überschreiben. Der Morris-Wurm von 1988 [P2] zeigte, dass diese Pufferüberläufe gezielt dazu genutzt werden können das Verhalten eines Programms beliebig zu beeinflussen. Laut MITRE Common Weakness Enumeration (CWE [P3]) ist diese Art des Angriffs auch im Jahr 2015 noch immer eine der weitverbreitetsten. Diese sog. Laufzeit-Angriffe befinden sich auf Platz 2 (*"OS Command Injection"*) und Platz 3 (*"classic buffer overflow"*) der CWE Rangliste. Sie ermöglichen Angreifern sowohl Eingaben zu steuern, Berechnungen zu verändern oder Ausgaben zu fälschen, beispielsweise mit dem Ziel Online-Banking-Transaktion zu ändern, Spam-Email-Server im Hintergrund zu installieren oder Opfer zu erpressen, indem wertvolle Dateien verschlüsselt werden.

Diese Dissertation stellt vier neue Ansätze vor, welche alle auf unterschiedliche Weise bösartige Verhaltensänderungen von eigentlich gutartiger Software verhindern. Da auch die Angriffe während des Schreibens dieser Dissertation verbessert wurden, stellen die hier beschriebenen Lösungskandidaten einen iterativen Prozess dar, der über den zeitlichen Verlauf dieser Dissertation in einem stetigen Katz-und-Maus-Spiel stückchenweise verfeinert wurde.

Abstract

A bigger threat than malicious programs themselves are benign programs that can turn malicious. While malicious programs have the clear intention of stealing or manipulating data, a benign program was developed with good intentions to fulfil everyday purposes such as browsing the Web, reading mail or typing letters. If programming flaws in those benign programs can be exploited to suddenly change their behaviour, traditional virus scanners do not have a chance because there is no ingress of a malicious program. Even worse, most of the computers, smart phones, and tablets share the same software that has been produced in volume, and if exhibiting such a program flaw, malicious behaviour might be imposed on millions of instances out there.

As early as 1972, it was discovered that improperly checked input data processed by a program may change its behaviour on behalf of an attacker [P1, p.61]. Program flaws such as allowing program input to exceed the designated reserved space for that input, have fatal consequences as they overwrite consecutive data or even code. The Morris worm from 1988 [P2] showed that those overwrites can be performed very purposefully as to arbitrarily change the behaviour of a program. As of 2015, this category is still one of the most prevalent types of attacks according to the MITRE Common Weakness Enumeration (CWE [P3]). It ranks top-2 (*"OS Command Injection"*) and top-3 (*"Classic Buffer Overflow"*). It enables attackers to control input, computations and output to achieve their desired goal, such as changing a victim's online banking transaction, installing e-mail spamming servers or blackmailing victims by encrypting their valuable files.

This thesis presents four novel approaches that all mitigate or prevent this type of imposition of malicious behaviour on otherwise benign code. Since attacks have adapted during the writing of this thesis, the counteract techniques presented herein are tailored towards different stages of the still ongoing cat-and-mouse game and each technique resembles the status quo in defences at that time.

Acknowledgements

Even though a Ph.D. thesis is supposed to be a testimony of independent work, the truth of course is that many people have inspired, supported and influenced me. This is why I would like to take the opportunity to thank my supervisors, co-authors, friends and family, who have supported me or otherwise suffered from my lack of time.

I would like to thank Prof. Dr. Michael Backes, my supervisor, who has strongly supported me in my academic career, always gave me insightful comments and we had lots of fruitful discussions. I would also like to thank Prof. Dr. Ahmad-Reza Sadeghi, my former supervisor, who introduced me to the practices of paper writing and selling. I am also grateful for the successful collaboration with Prof. Dr. Thorsten Holz' system security group. The joint work with him and his Ph.D. students resulted in exciting projects and several successful scientific publications. I am honoured to be working in an extremely open and collaborative work environment in which a number of outstanding researchers have supported, encouraged, verified and falsified my research ideas. The enthusiastic and close collaboration with Sven Bugiel, Lucas Davi, Sören Bleikertz, Sebastian Gerling, Dominique Schröder, Thomas Schneider, and Hugo Ideler have resulted in several scientific papers. In particular, I would also like to thank those who have explicitly or implicitly influenced the research direction of this thesis: Matthias Schunter, Philipp von Styp-Rekowsky, Stephan Heuser, and Steffen Schulz.

I also thank my biggest fan who would like to remain anonymous.

Contents

1 Underlying Scientific Papers

This dissertation is based on a subset of my published, peer-reviewed scientific papers. The selected subset consists of four mitigation techniques against run-time attacks that change the behaviour of an otherwise benign program. I contributed to all papers as one of the main authors.

The following list places the underlying scientific publications in time and gives background information and a short summary while the technical depth is explained in their respective chapters.

1. The first approach, presented in chapter 5, is based on **Control Flow Integrity** (CFI) and operates on unmodified binary executables for Apple's iOS platform. The presented approach, called *MoCFI* (**Mo**bile **CFI**), extracts a CFG from the binary executable (the iOS App) and constantly checks the desired control flow of a program against the intended control flow of the current execution. This work was published in IEEE's proceedings of the *Symposium on Network and Distributed Systems Security (NDSS)* in 2012 [M1]. I contributed to the entire MoCFI solution by devising and implementing methods to extract the CFG from iOS executables file and by developing and implementing the CFI checks at run-time, and securing Objective-C message passing with CFI enforcement.

2. Another defence against run-time attacks, presented in chapter 6, is specifically geared towards the shortcomings of ASLR (Address Space Layout Randomisation). This approach is based on load time **Dynamic Binary Rewriting** of executable files and divides legacy binary code into small chunks, which are then randomised to achieve high entropy. This deprives an attacker of the required knowledge about code and data layout. This work was published in the ACM's proceedings of the *Symposium on Information, Computer and Communications Security (AsiaCCS)* in 2013 [M2]. I designed and implement a binary rewriter from scratch that is capable of doing in-place rewriting on process start-up. I have also designed all necessary algorithms, such as the creation of the CFG, representation of instructions in an intermediate language, randomisation algorithms, optimisation algorithms, process loading and executable and shared library parsing. I further extended this binary rewriter for the approach presented in the *Oxymoron* chapter.

3. In that **Oxymoron** chapter (chapter 7), I propose a different approach to achieve fine-grained randomisation. That approach tackles the problem that diversification in processes contradicts the longstanding paradigm of memory sharing amongst processes in order to save space. The solution presented in chapter 7 is the first combination of memory sharing despite having randomised memory layouts and is achieved using **Memory Segmentation**. It was published in the proceedings of the *USENIX Security Symposium* in 2014 [M3]. To achieve the combination of memory sharing and randomisation, I invented Position-and-Layout-Agnostic Code, a derivative of the traditional x86 calling convention that does not incorporate any absolute addresses. To this end, I developed a translation from legacy x86 code to Position-and-Layout-Agnostic Code that is transformed with the help of the aforementioned binary rewriter.

4. An attack to fine-grained memory randomisation with address hiding is JIT-ROP (Just-in-time return-oriented programming). It can revert fine-grained memory randomisation using a memory disclosure vulnerability. In chapter 8, I present a solution that prevents such memory disclosure vulnerabilities from being exploited in order to protect fine-grained memory randomisation. To this end, I modified the **Linux Memory Paging** mechanism to emulate a non-existing hardware primitive **XnR** (eXecute no Read). XnR prevents an attacker from gaining knowledge about code of an unknown or randomised process. This work has also been published in the proceedings of the ACM's *Conference on Computer and Communications Security (CCS)* in 2014 [M4]. I have designed the XnR primitive and developed a software emulation for that hardware feature. I also implemented the XnR solution for the Linux kernel to demonstrate its effectiveness on today's hardware.

I have also published a number of other scientific peer-reviewed papers, which are not subject of this thesis and are listed in the following as a reference and might give some background information about myself. The papers that are part of this thesis are highlighted in gray.

2016

Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. “RamCrypt: Kernel-based Address Space Encryption for User-mode Processes”. *ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS)*. 2016

Johannes Krupp, Dominique Schröder, Mark Simkin, Dario Fiore, Giuseppe Ateniese, and Stefan Nürnberger. “Nearly Optimal Verifiable Data Streaming”. *IACR International Conference on Practice and Theory of Public-Key Cryptography (PKC)*. 2016

2015

Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. “How to Make ASLR Win the Clone Wars: Run-Time Re-Randomization”. *23rd Annual Symposium on Network and Distributed System Security (NDSS 2016)*. 2015

2014

Michael Backes, Rainer W. Gerling, Sebastian Gerling, Stefan Nürnberger, Dominique Schröder, and Mark Simkin. “WebTrust - A Comprehensive Authenticity and Integrity Framework for HTTP”. *International Conference on Applied Cryptography and Network Security (ACNS)*. 2014

Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. “You Can Run but You Can’t Read: Preventing Disclosure Exploits in Executable Code”. *ACM conference on Computer and communications security (CCS)*. 2014

Michael Backes and Stefan Nürnberger. “Oxymoron - Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing”. *USENIX Security Symposium*. 2014

2013

Sören Bleikertz, Sven Bugiel, Hugo Ideler, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “Client-controlled Cryptography-as-a-Service in the Cloud”. *International Conference on Applied Cryptography and Network Security (ACNS)*. 2013

Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “Gadge Me if You Can: Secure and Efficient Ad-Hoc Instruction-Level Randomization for x86 and ARM”. *ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS)*. 2013

2012	Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “XIFER: A Software Diversity Tool Against Code-Reuse Attacks”. *ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3)* (2012)
2011	Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. “AmazonIA: When Elasticity Snaps Back”. *ACM conference on Computer and communications security (CCS)*. 2011
	Sven Bugiel, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. “Twin Clouds: Secure Cloud Computing with Low Latency”. *Communications and Multimedia Security (CMS)*. Springer, 2011
	Sven Bugiel, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. “Twin Clouds: An Architecture for Secure Cloud Computing”. *Workshop on Cryptography and Security in Clouds (CSC)* (2011)
	Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “Poster: Control-flow Integrity for Smartphones”. *ACM conference on Computer and communications security (CCS)*. ACM. 2011
	Stefan Nürnberger Martin Steinebach Sascha Zmudzinski. “Resynchronizing Audio Watermarking after Non-linear Time Stretching”. *Electronic Imaging 2011 - Media Watermarking, Security, and Forensics*. 2011
	Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “CFI Goes Mobile: Control-Flow Integrity for Smartphones”. *International Workshop on Trustworthy Embedded Devices (TrustED)*. 2011
2010	Stefan Nürnberger, Thomas Feller, and Sorin A. Huss. “Ray - A Secure Microkernel Architecture”. *IEEE International Conference on Privacy Security and Trust (PST)*. 2010
2009	Stefan Thiemert, Stefan Nürnberger, Martin Steinebach, and Sascha Zmudzinski. “Security of Robust Audio Hashes”. *IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE. 2009

2 Motivation

Attackers often take advantage of program vulnerabilities to manipulate a remote program or service in their favour. Common attacks target widespread programs such as Adobe's PDF viewer *Adobe Reader*, Microsoft's browser *Internet Explorer* or even entire programming languages such as Java because they are executed in their own run-time, which is written in vulnerable C++ code. The motivation of the attackers is often monetary, realised, e.g. by manipulating web browsers or web servers in order to eavesdrop on sensitive data such as credit card details. From a technical point of view, the culprit is the fact that input processed by a program might influence its behaviour in an arbitrary way chosen by an attacker.

Intriguingly, the execution of arbitrary code that can change during run-time is actually a fundamental feature of computers, which makes them universal computing machines. In fact, this universality is what sets computers apart from other devices and enables these machines to compute every computable function that will ever exist. In other words, computers can solve problems that did not even exist when the computer was built. As a result, detecting a security flaw is not as easily as disallowing the unknown to execute. Quite the opposite, a computer constantly executes a continuous stream of instructions that is fed to it. For an outside observer, programs are always happily computing as they should based on their input – may it be malicious or benign.

Not surprisingly, there is a whole sub-discipline of computer science that aims at preventing unintended behaviour: the field of *correctness*. In the field of correctness, a program that has been designed to do an exactly specified task is formally verified not to do anything else. However, formal verification or automated theorem proofing often requires source code and sometimes is impractical or even impossible for off-the-shelf software or even an entire operating system such as Windows or Linux. Having said that, there exist several approaches that verify operating system components [P4, P5] or even an entire micro kernel [P6]. The general case, proving that a given program is always secure for any possible input is even impossible as this would be equivalent to the halting problem. This is why the goal of this thesis is to rather harden software by converting its binary form into a more secure version – without the need to access source code. This approach is more realistic to have greater impact.

The hardening in this thesis always aims at preventing arbitrary code execution, or at least confining the consequences of code execution vulnerabilities in off-the-shelf programs. Though, the solutions differ in *where* the technique is applied and as a result in *when* it secures a program. Some techniques apply before the vulnerability can be exploited, others after it was exploited but before it can have consequences. These possible points of application correspond to the *dependability chain* [P7]. This chain describes a causality relation in dependence of each other and is depicted in Figure 2.1.

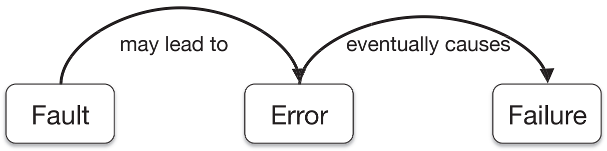


Figure 2.1: The dependability chain of *Fault* → *Error* → *Failure*

Fault - Is the existence of a deviation between intended will and actual design of a program.

Error - If the path exposing the fault gets executed while the program runs, the fault has surfaced as an error. If the faulty part of the program does not get executed, this error cannot occur.

Failure - If the error changed the integrity of the program, it might fail. This does not necessarily mean, that it will crash. But it certainly constitutes undefined behaviour.

For example, a program *fault* may be that an array is not checked for its length when writing elements to it. If the faulty program executes that affected part, data might or might not be written beyond the arrays's boundary. If the data overwrote the array's boundary, it overwrote other data, thereby creating an *error*. This error might later cause a *failure* when the wrong, now overwritten, data or code is used.

A very challenging attribute in the chain of *Fault* → *Error* → *Failure*, is the fact that the *Failure* may occur at any time later after the *Error*. This makes it hard to pinpoint the occurrence of a *Failure* to a specific *Error*. The occurrence of a failure may even depend on the current stack layout, input data, and so forth. Therefore, after an error occurred, the program may show erratic behaviour and it might be impossible to reproduce the exact same behaviour again (a so-called *Heisenbug*).

2.2 | Safe Languages

The most fundamental approach to prevent attacks that change a program's behaviour are situated in the first link of the chain: *Fault*. If a fault can be prevented, the corresponding *Error* and *Failure* will never occur. Memory-safe and type-safe languages or memory-safety dialects [P8]of existing languages are such fundamental approaches. They ensure that classical buffer overflows or pointer manipulations cannot take place and hence programs cannot exhibit unintended control flow changes. Popular examples of such safe programming languages are Java, Ada, and C#.

However, language popularity statistics indicate that unsafe languages, especially C, are still more popular than Java or C# (see Figure 2.2), despite Java being on the market for more than 20 years now.

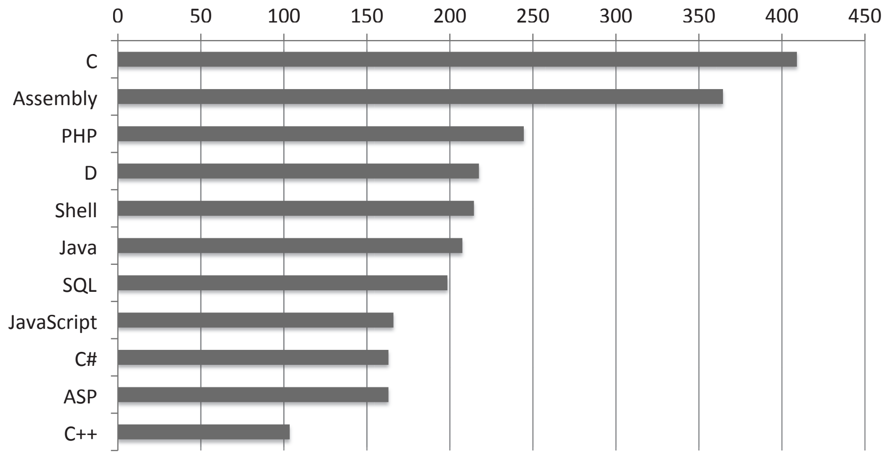


Figure 2.2: Language popularity: Measured by parsing required programming language skills of software developer job offers on Craigslist
source: `http://langpop.com/` (October 2015)

The reasons for that dominance over Java are manyfold: Security and safety is often not a major concern and Java is often associated with performance overhead [L1]. Maybe the choice of programming language and the resistance against certain languages is rooted deep in the inner persuasion about programming language favourites. An additional factor that should not be underestimated is pride: A 10% performance loss for, e.g. in a browser, is equivalent to a setback of a year in terms of performance development [L2].

This continued development of software in old but established languages, foremost C and C++, has led to a whole field in computer security that is devoted to the fact that there is demand for securing legacy code written in legacy languages. Software fault isolation, control flow integrity, security policies, and randomisation all strive for the same goal: depriving programs of their once beloved Turing completeness in order to tame them to entities that have an assessable power. Figure 2.3 categorises those different defences by situating them in the different links of the dependability chain.

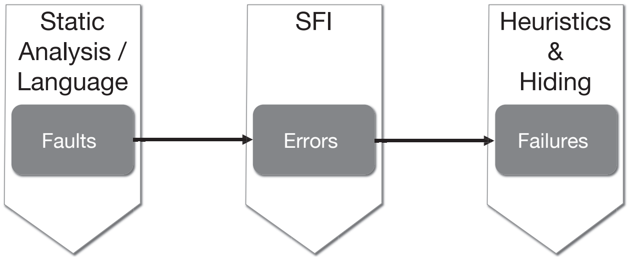


Figure 2.3: Possible anchor points for defences against arbitrary code execution

Safe languages, such as Java, and also safe dialects of C, would be situated in the first link: *"Fault"*. Here, buffer overflow and pointer pivoting bugs are prevented by the design of the language. In the middle link, *"Error"*, approaches can be summarised as *Software Fault Isolation* (SFI): Bugs may still exist in the unsafe language and therefore programs experience *Faults*. However, when a *Fault* appears in the form of an *Error*, it can be detected and program execution can be stopped. For example, a buffer overflow might have already overwritten crucial control data but the consequences can be prevented, e.g. by *Control Flow Integrity* checks, which are alluded to in more detail in the background given in chapter 3. In the link on the right hand side, *"Failure"*, security solutions even tolerate overwriting critical control data (Fault) and jumping to that overwritten addresses (Error), but the consequences are either unlikely or mitigated. A prime example would be *Address Space Layout Randomisation* (ASLR), which does not deprive the adversary of its power to change control flow but rather puts him in a position where he does not know what to do. These concepts are explained in more detail in chapter 3.

While preventing the first link in the dependability chain seems to be the most fundamental approach and the most reasonable from an academic point of view, the industrial use of unsafe C is strikingly prominent (cf. Figure 2.2).

This is why this dissertation focuses on hardening the status quo of unsafe software written in C, C++ and Objective-C. All techniques described herein aim at detecting and preventing certain malicious behaviour. The described solutions are all situated in the middle and right link of the dependability chain, namely methods of SFI, information hiding and heuristics for detection.

3 Background

If an attacker is able to manipulate a program such that it executes code on her behalf, this is equivalent to running a program crafted by the attacker on the victim's machine. This **arbitrary code execution** (a.k.a. **run-time attack**) is particularly severe if it can be triggered remotely. This enables an attacker to compromise millions of machines over the Internet if they suffer from such a vulnerability. Once she controls a single process on the victim's system, she can exploit more vulnerabilities in order to escalate privileges up to system level, effectively taking over control of the operating system. Having achieved this highest level of exploitation, the attacker is able to cover traces pointing to her malicious code and to an infection of the system.

There are many causes for code being susceptible to compromising. In the early days of such exploits, programs were vulnerable to execution of injected code. Meanwhile, almost thirty years after the first incarnation of remote code execution in the wild [P2], this type of attack is still one of the most prevalent ones – however, it has become substantially more sophisticated. The mutual ping-pong between attack and defence has led to a profound refinement of attacks and defences. This evolution culminated in a defence side that uses self-mutating targets through randomisation, which are attacked from the other side using just-in-time compiled malicious code that automatically tailors themselves to unknown processes. To give readers some background on the level of sophistication we have reached thus far, this chapter sheds some light on the history and chronological development of those types of attacks.

3.1 History of Code Injection and Code Reuse

The first written source that mentions security vulnerabilities that can lead to the execution of attacker's code is the *Computer Security Technology Planning Study* by the U.S. Air Force [P1] from 1972. On pages 60 and 61, the study says *"The major weaknesses of contemporary operating systems occurs at the interface between the system and the user. [...] The multiplicity of implementers almost guarantees that one or more important checks will be overlooked. [...] This can be used to inject code into the monitor that will permit the user to seize control of the machine."* One of those *"important checks"* the author refers to, is ensuring that user input data can never be stored at places not intended by the programmer or system design. A very popular example of such missing checks, is checking the destination size when writing to a buffer. If this check is not performed, input data might exceed the designated length of that buffer, thereby overwriting arbitrary consecutive data. This consecutive data might not only contain important control information but also pointers to code. In particular, most processor architectures store return addresses for subroutines on the stack. A deliberate overwriting of a buffer with input data that is too long can thus overwrite the return address, which effectively changes program control flow on the attacker's behalf. A typical stack layout, which demonstrates this vulnerability, is shown in Figure 3.1: the stack was created from right to left. First, the caller put the return pointer on the stack to enable the callee to jump back where execution came from. Then, the prologue of the callee saved the old `EBP`, the base pointer against which local variables are referenced, on the stack. This ensures it can be restored when exiting the sub-route. After that, the sub-routine placed the local variables `buffer` and `i`. If the array `buffer` is overwritten beyond its allocated size, other data gets overwritten, namely the variable `i`, the saved `EBP` and the most notably the return pointer. By overwriting the return pointer, the attacker can change the control flow by deliberately targeting to where the sub-routine returns. The attacker's goal is to execute arbitrary instructions in order to completely control the program. This is fundamentally possible in two different ways. Either by inserting her own instructions and directing control flow to them or by directing control flow to useful existing instructions.

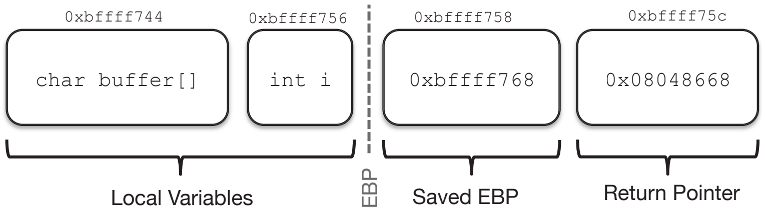


Figure 3.1: A typical stack layout on the common Intel IA-32 architecture: The stored buffer is succeeded by a return pointer, which is read by the processor.

Classic Buffer Overflow. One possibility to exploit a buffer overflow on the stack is to inject code in the form of binary instructions into the buffer that can be controlled by the attacker. The injected data has to be carefully laid out such that it overwrites the return pointer. This attacker-chosen return pointer must point to a valid address, specifically to the beginning of binary instructions that have just been inserted (`0xbfffff744` in Figure 3.1). To this end, the attacker needs to know the memory address of the vulnerable buffer so that she can point execution to the content of that buffer. If the exact address is not known, a typical means is to use a *NOP sled*, a long array of consecutive *No-Op* instructions. With a NOP sled in place, the chosen address does not need to be exact any more but landing somewhere inside the injected data suffices such that execution will slide to the actual useful instructions.

No-eXecute (NX) Processors. The inherent flaw, namely buffer overflows, can be avoided by using higher level languages that support automatic array bounds checking. However, it is impractical to re-write millions of lines of code in legacy software. Hence, an easy fix to thwart the execution of data that has been injected at run-time is to mark memory regions as either executable (X) or writeable (W). This principle ensures that data, which has been written at run-time cannot be executed later. Code that is part of the intended program, however, is not writeable and hence cannot be changed and can be safely executed. An additional advantage is that this mechanism can be implemented in hardware and therefore has no performance overhead. The *Write* xor *eXecute* ($W \oplus X$) hardware support was first introduced in the AMD64 Opteron processors under the name *No eXecute (NX)*. Other nomenclatures for the same technique are Intel's *eXectute Disable (XD)*, or Microsoft's *Data Execution Prevention (DEP)*.

Return-oriented Programming. Because the broad introduction of $W \oplus X$ successfully mitigates code injection attacks, the only possibility left for an attacker is to divert control flow to a useful position inside already existing code. Surprisingly, the reuse of existing code is not limited to the functionality of existing code but can arbitrarily exceed it. Similar to how simple assembler instructions can form complex programs, the re-organisation of seemingly simple and benign assembler instructions can be used to build arbitrary, complex, and malicious new behaviour. To achieve this, the attacker selects individual assembler instructions and executes them in the order of his discretion. This is made possible by jumping the individual instructions instead of the beginning of sub-routines. Because a program easily consists of millions of instructions, the attacker can choose those, which are followed by a `return` instructions, thereby effectively isolating the execution of a single instruction of her will. The subsequent `return` instruction has the side effect of fetching the next return pointer from the stack, which results in execution of the next instruction that the attacker has chosen. These chains of individual, attacker-chosen instructions are identical to writing arbitrary assembler code based on already existing instructions. This method successfully circumvents $W \oplus X$ protection as it does not introduce new code but rather reuses individual, existing instructions. These small code pieces that typically consist of one or a few instructions followed by a `return` are called *gadgets*. Using a stream of gadgets by placing their addresses on the stack is called *return-oriented programming (ROP)*.

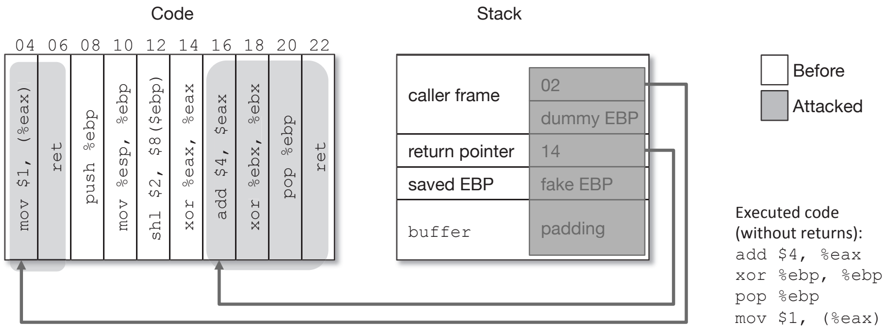


Figure 3.2: ROP attack that selects useful gadgets and chains them together.

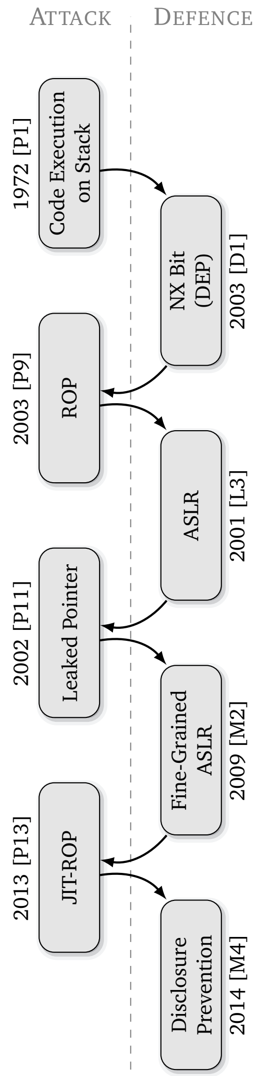
Since $W \oplus X$ does not prevent actual buffer overflows, it is still possible for an attacker to overwrite return pointers such that they point to arbitrary many useful gadgets. This way, it is possible to chain together a multitude of useful assembler instructions – thereby creating arbitrary new behaviour. It is therefore not surprising that it has been shown that ROP is Turing-complete [P9, P10]. A ROP attack example is shown in Figure 3.2: the original stack layout is depicted in white and grows from top to bottom. The attacked layout is depicted in grey and grows from bottom to top. The buffer is overwritten with garbage padding data that is not useful to the attacker but enables her to reach to the *saved EBP* and *return pointer*. By overwriting the saved `EBP` with a fake `EBP`, the attacker accommodates for fact that the sub-routine expects an `EBP` to be saved there. The most important overwritten data starts after that, the return pointer that the attackers wants to manipulate. In the example in Figure 3.2, the attacker overwrite the return pointer with '`14`', which will divert control flow to the address after `14`, which is `16`. Hence, the manipulated stack layout forces execution to continue at address `16`, which is `add $4, %eax`. The execution will continue to execute the following instruction until the `ret` instruction at address `22` is reached. This `ret` instruction pops the next return pointer from the stack, `02` in the example, and executes `mov $1, (%eax)` accordingly. The the return values (`14` and `02`) are not directly following each other as the executed code of the first gadget contains a `pop %ebp` at address `20`, which needs a *dummy EBP* value to pop off the stack.

Address Space Layout Randomisation (ASLR). For these gadgets to work, the attacker needs to place the exact address on the stack. The next move in that mutual ping-pong of attack and defence was therefore to make those needed addresses unpredictable by an attacker. Address Space Layout Randomisation (ASLR, e.g. [L3]) randomises the base address of loaded code, data, and stack segments in memory. Therefore, ASLR in theory makes it infeasible for an attacker to predict the location of gadgets in memory. To make ASLR practical, it only randomises the base address to leave relative offsets inside the program intact. This way, the ASLR-enabled program becomes *position-independent* as it is able to reference all parts of code and data relative to the current execution address. However, this relative addressing can also be used by an attacker who gained knowledge about a single leaked pointer [P11] – may it be a data or a code pointer. From this leaked pointer, all addresses that are of interest to an attacker can be calculated in relation to the leaked pointer.

Additionally, it has been shown that ASLR suffers from low entropy in practice [P12], which originates from various factors such as limited freedom in address assignment and coarse-grained hardware memory addressing. In fact, Shacham et al. [P12] found that the real entropy on a 32 bit memory space is actually 16 bits. These mere 65,536 possibilities can be brute-forced in cases when the victim process re-starts automatically and shows the same randomisation on every start. This seemingly artificial scenario is actually pretty common for web servers such as Apache, because they fork child processes for handling incoming connections. The creation of child processes using the `fork()` system call efficiently starts a new process by copying the parent's address space. As a result, every `fork`ed child exhibits the exact same memory layout. It is actually a feature, not a bug, that a web server is designed to re-start child processes that have crashed. This way, availability is provided.

However, an attacker can exploit the automatic restart of child processes combined with the fact that restarted children will have the same memory layout to brute-force all possible combinations until her attack was successful. Each unsuccessful guess of a valid address will result in a restart thereby narrowing down the possible addresses as the newly created child will exhibit the exact same memory layout.

Despite its known shortcomings, ASLR is actually *state-of-the-art* as of 2015 for defences included in commercial operating systems such as Windows, Mac OS or Linux. In contrast, in the academic world, better solutions have been proposed that, however, have not gained widespread adoption in off-the-shelf operating systems yet.



Fine-Grained Randomisation. Many researchers have improved defences against these remaining attacks, e.g. by finer and finer code randomisation that, in contrast to ASLR, also shuffles the code itself, not just its base address. Despite the fact that such randomisation is a simple idea, its implementation is highly involved and several approaches exist in the literature, ranging from compiler-based solutions [P14, P15, P16] to run-time solutions [P17, M10, M2] that randomise the program either once or even constantly during its lifetime [P18].

Such solutions can either be based on source code transformations, linker modifications, static or dynamic binary translation. While source code transformations [P19] change the source code to modify itself during run-time, linker modifications split small compiler units, such as functions, in separate libraries that can be loaded in a random order [P20]. *Static binary translation* works on already compiled code, reads an executable or shared library file from disk, disassembles it and transforms the instructions according to a pre-defined pattern within the executable file itself. *load time translation* solutions are similar to static translation but apply the translation at load time in order for the processes to benefit from a re-randomisation at each run. This is achieved by rewriting the binary file after it has been loaded but before execution [P21, M10, M2]. In chapter 6, I describes a new method for a load time, fine-grained randomisation scheme.

Just-in-Time ROP. Snow et al. [P13] showed that even fine-grained memory randomisation is not effective if programs suffer from a disclosure vulnerability which in turn allows the assembly of ROP gadgets on-the-fly. They showed that it is possible to traverse valid memory a few or a single valid memory address.

To get a single valid address, one could a buffer over-read to read a valid return address from the stack. Given a memory disclosure vulnerability, bytes stored at that discovered address can then be read. From there, they explore the address space of the vulnerable process step step by following the control flow. This is possible through dynamic disassembly of the memory content that has been discovered so far. After they have found enough ROP gadgets, they compile the payload so that it incorporates the actual current addresses that were recorded on-site. This on-the-fly payload is then identical to classical ROP payload, only the (valid) address have been discovered at run-time.

Control Flow Integrity (CFI). While the approaches presented above tackle the problem by limiting the attacker's capabilities after a control flow vulnerability has been exploited, CFI tackles the root cause—namely prohibiting disallowed control flows. To this end, CFI extracts a control flow graph (CFG) from the program and thereby determines the allowed flows from each vertex in the graph. During run-time, CFI then checks for every possible control flow change if the control flow edge is allowed according to the extracted CFG. The rules for edges and vertices in a CFG are clearly defined: Each basic block (BBL) of code is represented as a vertex. A BBL is defined as a set of instructions that are executed one after another without changing the control flow. The last instruction of a BBL is always a so-called *exit instruction* that changes the control flow (e.g. jumps, branches, function calls). The edges connecting the BLLs, or vertices, represent the possible target of such exit instructions. A typical CFI enforcement is shown in Figure 3.3.

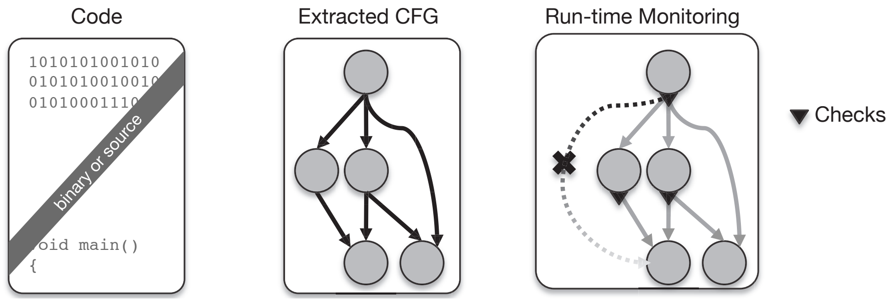


Figure 3.3: CFI extracts a CFG of the program to check whether control flow stays on the pre-determined allowed path during execution.

First, the code is disassembled to discover control flow instructions (left). These instructions are used to create the control flow graph (CFG), which contains the theoretically possible control flow (middle). For each taken branch during run-time, the currently intended control flow is compared to the theoretically possible saved in the CFG, which constitutes the allowed control flow. Any deviation from the CFG results in an exception, thereby precenting the exploitation of control flow deviations. However, the necessary disassembly (first step, left in the figure) of the binary code is not ambiguous, which is a result of the impreciseness of disassembly in general. This ambiguity can stem from aligned data or aligned functions that lead to different disassembly depending on whether code is disassembled fall-through or recursively [P22].

3.3 Terminology

To avoid confusion, the following is a list of technical terms and how they are understood and used within in thesis.

Program. A fixed set of code, pre-initialised and uninitialised data stored in a standardised file format on disk.

Process. A process in contrast is an instantiation of a program that has been loaded into memory by the operating system. Multiple processes of the same program may co-execute simultaneously.

Compile-Time. The point in time before or during a program is compiled from its source code form into a binary executable consisting of processor instructions.

Load-Time. The moment in time when a program is loaded from disk and copied into a fresh address space, waiting to become a new process.

Run-Time. The entire time after load-time while a program is running, until it is terminated.

Instruction The smallest execution unit that can be fed into a processor as gets interpreted as a command to do something.

Exit instruction. An exit instruction is an instruction that can change the control flow such that program execution may continue at a different address. Such instructions change control flow either directly (e.g. `call`), indirectly (e.g. `jmp *%eax`) or conditionally (e.g. `jne`).

Entry instruction. Similarly, an entry instruction is the first instruction to which other exit instructions point.

BBL. A *Basic BLock* (BBL) consists of an entry instruction as first instruction, potentially some other instructions that cannot change control flow are always executed one after another. A BBL ends with an exit instruction. Every program/code can be divided in a set of non-overlapping BBLs.

CFG. (Control Flow Graph) The representation of a program as a graph in which nodes represent BBLs. The edges in turn represent control flow changes, i.e. the exit instructions.

Run-time attack. An attack that dictates a running program which instructions to execute – either by injecting new instructions or by re-using existing instructions.

Leaked pointer. A direct channel (not a side channel) through which information about a certain memory address of the currently running program is revealed to the attacker. This can be through a dangling pointer, a format string vulnerability or a buffer overflow read vulnerability.

Memory Disclosure Vulnerability. Allows an attacker to retrieve the memory content of an arbitrary address chosen by the attacker. Such attackers occur for example when a buffer overflow vulnerability allows an attacker to overwrite the value of an integer pointer, which can also be read by the attacker.

KiB, MiB, GiB, ... This thesis uses the computer science friendly nomenclature of *Kibibytes* (1024 bytes) instead of *Kilobytes* (1000 bytes). It is abbreviated *KiB*. Analogously, 1 MiB is 1024 KiB or $1,048,576$ bytes as opposed to one Megabyte, which is $1,000,000$ bytes. *GiB* stands for *Gibibyte* and so on.

4 Related Work

The field of related works is very broad and there is no *one* way to cluster them according to one or two dimensions. They could be divided into *Attacks* vs. *Defences*. But there is more to it because related work discusses the underlying problems, root causes, and software bugs that lead to arbitrary code execution. Defences, on the other hand, could be divided into approaches that tackle symptoms of malicious code execution and methods that try to solve the root cause instead. While addressing the root cause of unintended code execution is *more fundamental*, the effectiveness and holism of those approaches is usually hard to prove. The other direction, fighting the symptoms, can be summarised as the goal of *hiding information from the attacker*. If this goal were met, an attacker could not exploit vulnerabilities because she could not gain information about memory layout. Some ideas stayed purely academic while others have meanwhile found their way into commercial mainstream operating systems.

Table 4.1 on the following page clusters prior work, concurrent work and derivative work according to different criteria.

4.1 | Software Fault Detection & Isolation

In the broad area of fault detection and isolation, generally software bugs are expected and detection mechanisms are in place to detect deviation from intended behaviour. One prominent example is Google's Native Client (*NaCl* [P46, P47]), a modified compiler that ensures reliable disassembly combined with forced inline reference monitoring. This approach enables the run-time environment to reason about the NaCl program's behaviour because control flow can only take pre-defined and verifiable paths.

More specifically, Control Flow Integrity (CFI) is the approach to prevent a program from leaving its intended control flow. On the one hand, this makes CFI a powerful and fundamental primitive, as it is agnostic to the type of attack and can detect attacks very early, namely when they try to change the intended control flow. On the other hand, CFI cannot use high level semantics to model actual behaviour of a program in order to detect deviations. The high fan-out of vertices in a CFG also makes CFI coarse-grained because it generally does not have contextual information that would allow CFI to constrain control flow to

		SOFTWARE FAULT DETECTION & ISOLATION		INFORMATION HIDING	
		Control Flow Integrity (CFI)	Integrity Checking	Randomisation	Memory Disclosure
WEAKNESS		• Stitching the Gadgets [P23] • Out of Control [P24]	• Smashing the Stack [P25] • Beyond Stack Smashing [P26] • Missing the Point [P27]	• Return-Oriented Programming (ROP) [P9] • Jump-Oriented Programming [P28] • ROP without Returns [P29] • Non-Control-Data Attacks [P30] • ROP is still dangerous [P31] • Size Does Matter [P32] • Counterfeit Object-oriented Programming [P33] • Hacking Blind [P34]	• Memory Disclosure [P35] • JIT-ROP [P13]
DEFENCE	Source Required	• Cryptographically enforced CFI [P36] • G-Free [P37] • HyperSafe (Hypervisor CFI) [P38] • KCoFI (/ˈkɒ-fiː/) [P39] • Modular CFI [P40] • RockJIT (CFI for JIT) [P41] • CFI for data sandboxing [P42]	• StackGuard [P43] • Write Integrity Testing (WIT) [P44] • AddressSanitizer [P45] • Native Client (NaCL)[P46, P47] • Code Pointer Integrity (CPI) [P48] • Fat Pointers [P8] • Low-Fat Pointers [P49] • ROPDefender [P50] • Return Address Defender (RAD) [P51]	• Multi Compiler [P14, P15, P16] • Source to source tranformation [P19]	• Readactor [P52]
	For Binaries	• Original CFI [P53, P54] • XFI [P55] • CCFIR (Compact Control Flow Integrity and Randomisation) [P56] • bin-CFI [P57] • Opaque CFI [P58] • MoCFI [M1]	• Program Shepherding [P59] • StackGhost [P60] Anomaly Detection: • kBouncer [P61] • ROPecker [P62]	• PaX ASLR [L3] • ASLP (Address Space Layout Permutation) [P63] • Cling [P64] • RISE (Randomized Instruction Set Emulator) [P65] • Librando for JIT [P66] • Smashing the Gadgets [P67] • ILR [P17] • STIR [P21] • Gadge me if you can [M2] Binary Rewriting Frameworks: 1. Valgrind [P68] 2. PIN [P69] 3. DynamoRIO [P70]	• Oxymoron [M3] • XnR [M4] • HideM [P71] • Isomeron [P72]

Table 4.1: Clustering of related work. Underlined names represent own work of this thesis.

only *one* allowed outgoing edge at any point in time. For performance reasons, though, even looser notions of CFI have been published [P57, P56].

The foundation for CFI has been laid by Kiriansky et al. in their seminal work on program shepherding [P59], which allows security policies to confine program behaviour. The actual term *Control Flow Integrity* has been coined by Abadi et al., who proposed CFI enforcement based on automatically assigned labels between basic block transitions [P53, P54]. At run-time, for every control flow decision (branch), the list of allowed labels is searched for the intended destination of that control flow. The authors extended their approach to code augmented with meta information about control flow and scope of memory access (XFI [P55]). *Write Integrity Testing* (WIT) goes one step further and determines to which positions code can write data [P44]. The authors use an inter-procedural points-to analysis, which computes the set of objects that can be written by each instruction in the program. Based on this information extracted during static analysis, WIT is then able to detect at run-time whether write-attempts to a certain location do originate from an instruction that is not supposed to write to that particular object.

While the approaches enumerated so far are all applied to executable file level, the *HyperSafe* approach places CFI as the lowest layer in the software stack underneath the unmodified operating system [P38]. Their modifications to the open source hypervisors Xen [L4] and Bitvisor [L5] protect the running operating systems and their user mode applications with control flow checks. It achieves this goal by instrumenting branch instructions with a call to an inline reference monitor that checks the validity of their target.

Since traditional CFI needs a static analysis phase in which all possible control flow targets are determined, it is not suitable for extensible code, such as dynamically loaded libraries or even just-in-time (JIT) compiled code. Niu et al. address things shortcoming with *Modular CFI* [P40], a representation of labels that allows to dynamically add other valid call sites during run-time. The authors use a similar approach to make CFI usable for JIT compiled code in RockJIT [P41]. Criswell et al. [P39] have implemented a version of CFI for use in kernels. Their CFI implementation *'KCoFI'* uses the FreeBSD [L6] kernel and has been partially formally verified.

To address the drawback that prior CFI solutions needed either source code or at least relocation information or debug symbols provided in the binaries, Zhang et al. [P57] have developed *CFI for COTS binaries*. Commercial off-the-shelf (COTS) binaries describe unmodified executable files that are provided *'as is'* by the operating system or distribution. Hence, their solution can be readily applied to all custom software without the need for source code or re-compilation. Zhang et al. also describe a looser enforcement for control flow targets, called *bin-CFI*. By assigning the same labels to all control flow targets, bin-CFI reduces the overhead of CFI checks as they do no longer need to check multiple allowed targets [P57, P56]. This effectively reduces control flow checks to ensure that execution flows to the beginning of *any* function, but not necessarily to the *one* function, which is correct in the current context. The strict notion of CFI gets even more relaxed because corner cases of control flow, such as exception handling and computed code pointers must be dealt with. This is done by including them in the list of allowed control flow targets. The authors argue that this relaxation is not severe, as the number of additional allowed targets does not increase significantly compared to all possible control flows in an unprotected binary. For this purpose, Zhang et al. introduced the notion of the *Average Indirect Target Reduction* (AIR). The AIR shall reflect how many addresses in a program can no longer be targeted on average, due to confined control flow. It is defined as the normalised sum of all ratios of confined control flow vs. program size. Hence, the AIR roughly puts the average control flow restriction for all possible control flows into perspective with unprotected programs. However, while the comparison of AIR values of conservative CFI with individual labels (99.13%) and bin-CFI (98.86%) suggests similar performance, a huge relaxation is actually the case. A value of 99.13% reduction of control flow targets for the original CFI work [P53] means that on average only 0.87% ($100\% - 99.13\%$) of all code can be called. In contrast, bin-CFI allows $100\% - 98.86\% = 1.14\%$ to be called. That is 31% more!

Hence, Göktaş et al. [P24] have shown that this relaxed CFI still allows modified ROP attacks to be mounted despite alleged CFI protection. Their exemplary modified ROP payloads for Internet Explorer are still within the allowed wiggle room of CCFIR and bin-CFI protected binaries. Also, Davi et al. discovered that the relaxed CFI notion as presented by [P57, P56] can be overcome.

Because of these weaknesses, Mashtizadeh et al. [P36] presented *cryptographically enforced CFI*. This approach is based on LLVM's [L7] `clang` compiler [L8] for C and introduces randomly created keys that are used to protect control flow targets using *Message Authentication Codes* (MACs). If an attacker does not know the key (because it is only stored in registers), she cannot create valid MACs for injected code addresses.

4.1.1 | Integrity Checking & Heuristics

The fact that an attacker can execute arbitrary code means that either the code itself can be modified or at least the data controlling the flow of code can be tampered with. This observation is the basis for many approaches that monitor the integrity of data structures during the run-time of a program.

An early solution, proposed by Cowan et al. [P43] in 1998 introduces so-called *canaries* on the stack – just before return addresses. Similar to coal mining, where this term was borrowed from, a canary is was early indicator for an immanent threat: the lack of oxygen. In contrast to mining, a digital canary does not die because of carbon monoxide, but they are placed such that overwriting return addresses will overwrite the canary. Hence, it is a good indicator that the integrity of the program has been violated.

Another way is to save a copy of the return address somewhere on a dedicated stack, so that it is infeasible, or at least unlikely, for the attacker to overwrite both. On return, the address on the dedicated stack is compared to the supposed return address on the ordinary stack. Should the values not match, an alarm is raised. This principle has been employed in StackGhost [P60], using hardware mechanisms so that the operating system can transparently keep a dedicated return copy stack for each running process it protects. Entire software solutions have also been proposed. These solutions create a second, dedicated stack in software and explicitly store each return address additionally on that stack. *Return Address Defender* (RAD) uses a modified compiler to replace the usual `call` and `return` instructions with more sophisticated code snippets that make use of the secondary return stack. Also *ROPdefender* [P50] works similarly but does not need access to source code to transform binary thanks to the binary instrumentation framework Pin [L9], which can add those checks dynamically during run-time.

4.1.2 | Avoiding Buffer Overflows

In order to avoid buffer overflows in the first place, several techniques have been proposed in the literature [P48, P8, P49, P44, P45]. All these solutions tackle the problem from a slightly different angle. The observation of *Write Integrity Testing* (WIT [P44]) was the fact that programs need to either change control flow data (e.g. pointers, C++ vTables, return addresses) or non-control data at some point in order to change the intended program behaviour. Consequently, WIT statically extracts from the source code which parts of the code legitimately write to which pointers, which data, which return addresses and so on. Then, during run-time a monitor enforces that each memory write operation originates from those extracted, legitimate code pieces only. Should data or even control-data be overwritten by a piece of code that would not and should not do that, it is detected and aborted, hence preventing the attack. A little less generic are solutions that protect pointers only [P48, P8, P49]. In these solutions, such as CPI, so-called *Fat Pointers* augment the traditional representation of a pointer, namely a pure integer representing the linear address in memory, by additional attributes such as a valid length of the object it is pointing to. This way, read and write operations can be check to be inside the legitimate boundaries of the object they actually intend to refer to. Should a pointer be pivoted to outside its legitimate range, the access will be detected. However, recent work has shown that some of those techniques might be flawed. In *'Missing the Point'* [P27], the authors show that the needed safe region for CPI can in fact be discovered by an adversary.

4.1.3 | Anomaly Detection

Other approaches in the literature rely on anomaly detection. Examples are *ROPecker* [P62], *Kbouncer* [P61] and *ROPGuard* [L10], which all try to detect ROP payloads by monitoring a process' behaviour for characteristics that differ from normal program execution. For example, a ROP attack usually chains small gadgets (one or a few instructions followed by a `return`) together. Thus, the number of returns per executed instructions is much higher than in usual, compiler-generated program code. This can be detected using the hardware performance counters [D2], such as the *Last Branch Record* (LBR) register. To minimize performance impact, only calls to critical functions (such as syscalls) are monitored and are used as checkpoints for benign behaviour. Recent research results suggest that such approaches can be bypassed by an attacker since she can construct gadget chains that bypass all proposed heuristics [P23, P32, P31].

4.2 | Information Hiding

Many defences are based on the fact that the attacker needs to know valid addresses in a program in order to direct control flow to instructions that benefit the attacker. Hiding those addresses is thus a viable line of defence. This can be achieved by randomisation, i.e. it is infeasible for the attacker to know necessary addresses to mount an attack. An additional hurdle for an attacker is to not to use addresses in plain text that could otherwise be revealed by attacker. *Pointguard* [P73] thus encrypts pointers to make them useless for an attacker who does not know the key.

4.2.1 | Randomisation

The probably most widely used implementation of randomisation is part of all major operating system: *Address Space Layout Randomisation* (ASLR). The implementations between Windows [D3], Mac OS [L11], Linux [L3], Android [L12] and iOS [L13] differ slightly, but are in essence very similar: The load address of the main executable and its shared libraries are randomised in memory. So is the stack and allocation of heap memory. However, the loadable segments of each module stay en bloc. This means that the main executable is loaded at a random address as a whole, i.e. `.text`, `.data` and `.bss` segment keep their relative addresses to each other. This process is repeated for each loaded shared library. Due to the way `fork()` works, the address space is copied when a child process is created, i.e. parent and child share the exact same addresses.

4.2.2 | Fine-Grained Randomisation

A significant downside of ASLR is that it suffers from low entropy [P12]. Additionally, the fact that modules are moved en bloc, means a single leaked pointer reveals all addresses relative to it, i.e. the entire module. Fine-grained randomisation schemes try to address these problems by splitting the code into pieces that are then randomised individually to enhance entropy and to make relative distances unpredictable.

One way to categorise fine-grained memory randomisation solutions is by their implementation: There exist compiler-based solutions, static or load time translations, and dynamic translations. Another dimension is whether they randomise only once, every time the program starts, or even continuously during program execution.

Compiler-Based Solutions. The idea of compiler-based approaches is to randomise the layout of a program and to install differently randomised copies on different computers so that the program layout is not predictable for an adversary.

Cohen et al. [P14] suggested compiling different versions of the same program. In a modern setting this technique can be applied within a smartphone app store to distribute individually randomised software. Similarly, Franz et al. [P15, P16] have suggested the automation of this compiler process such that every customer gets an individual version. This way, an attacker cannot create an exploit for multiple machines since she must know the individual code layout of a single victim instance. Franz et al. suggest that app store providers integrate a multicompiler in the code production process. However, those approaches have several shortcomings: First, app store providers have no access to the app source code. This requires the multicompiler to be deployed on the developer side, who has to deliver possibly millions of app copies to the app store. Second, the proposed scheme requires software update processes to correctly patch app instances that in turn differ from each other.

Code Islands [P20] uses a compiler-based solution to divide a shared library into several fragments. Their approach compiles groups of functions to several shared libraries instead of one shared library containing all the functions. These (potentially thousands of shared library files) are then put in a container whose format is understood by a modified loader which maps the libraries into the particular process. However, their solution needs a modified loader to support the proprietary format. Executables then need to load literally thousands of shared libraries, while each library constitutes a single function.

In contrast, Bhatkar et al. [P19] presented a source code transformer and its implementation for x86/Linux. The main idea is to augment any source code with the capability of self-diversification for each run. In particular, features are added to the source code that allow the program to re-order its functions in memory in order to mitigate code reuse attacks. Their tool can also be applied to shared libraries if the source code is available. However, their solution induces a run-time overhead of 11% and apparently needs access to the source code.

Static Translation. Static translation reads an executable or shared library file from disk, disassembles it and transforms the instructions according to a pre-defined pattern within the executable file itself. Kil et al. [P63] use static translation for their Address Space Layout Permutation (ASLP). ASLP performs function permutation without requiring access to source code. The proposed scheme statically rewrites ELF executables to permute all functions and data objects of an application. The presented scheme is efficient and also supports re-diversification for each run. However, only the functions themselves are permuted, not their content.

Pappas et al. proposed randomising instructions and registers within a basic block to mitigate return-oriented programming attacks [P67]. However, the proposed solution cannot prevent return-into-libc attacks (which have been shown to be Turing-complete [P74]), since all functions remain at their original position.

Load-Time Translation. load time translation solutions are similar to static translation but apply the translation at load time in order for the processes to benefit from a re-randomisation at each run. This can be achieved by several means, such as rewriting the binary file after it has been loaded but before execution [P21, M2]. Such solutions usually suffer from the fact that each execution either needs a translation/rewriting phase each time a process is started or they need a prior static analysis phase [P21].

Dynamic Translation. Dynamic translation leaves the original file untouched and does not apply binary rewriting but the program undergoes a *dynamic translation*, i.e. the instructions are transformed as they are executed. Dynamic translation is very similar to Just-in-Time (JIT) compilation but usually translates from and to the same instruction set architecture. For example, Bruening proposed the DynamoRIO framework in his PhD thesis [P70]. DynamoRIO is able to perform run-time code manipulation. ILR (instruction location randomisation) [P17] randomises the location of each single instruction in the virtual address space. For this, a program needs to be analysed and reassembled during a static analysis phase. This is why ILR induces a significant performance overhead (on average 13%), and suffers from a high space overhead, i.e. the rewriting rules reserve on average 104 MiB for only one benchmark of the SPEC CPU benchmark suite. For direct calls, ILR can only randomise the return address in 58% of the calls, meaning that for a large number of return instructions, ILR needs to do a live translation for un-randomised return addresses to run-time addresses.

Constant Re-Randomisation. To the best of our knowledge, there are only two papers that actually implemented and benchmarked re-randomisation. Curtsinger et al. [P75] have implemented an LLVM compiler modification that injects code, which adds the functionality to re-randomise the address of functions every 500 ms. Their overhead of code, heap and stack (re-)randomisation is 7%.

Giuffrida et al. [P18] changed the Minix [P76] microkernel to re-randomise itself every x seconds. This is achieved by maintaining the intermediate language of the LLVM [L7] compiler for the compiled kernel modules. However, this procedure has a significant run-time overhead of 10% for a randomisation every $x = 5$ seconds or even 50% overhead when applied every second.

Memory Sharing All the related work on fine-grained memory randomisation has in common that they either do not randomise shared libraries, or if they do, the difference introduced in the shared libraries prohibits code sharing. A solution to counteract this is presented in chapter 7 and was published in [M3].

4.2.3 | Memory Disclosure

It has been shown that fine-grained memory randomisation alone does not suffice to protect against code reuse attacks. Snow et al. [P13] showed that given a memory disclosure vulnerability, it is possible to assemble ROP gadgets on-demand without knowing the layout or randomisation of a process. They explore the address space of the vulnerable process step-by-step by following the control flow from an arbitrary start position. After they have discovered enough ROP gadgets, they compile the payload so that it incorporates the actual current addresses that were discovered in the victim's address space. This attack needs a valid code address to begin with in order to avoid accessing invalid memory.

Bittau et al. [P34] use a similar memory disclosure attack in their *blind ROP* attacks to exploit servers with unknown binaries to which the attacker normally does not have access. Their attack allows to scan a remote process for valuable gadgets, transfer the code contents back to the attacker over the network where she can then mount a traditional ROP attack. Blind ROP consist of three stages: First, a stack reading attack bypasses stack canaries and finds return addresses. Second, return addresses on the stack are altered and the server's behaviour is observed as to infer gadget positions. They end this stage once they found enough gadgets to perform a `write()`-syscall to transmit the executable memory over the network. Lastly, they scan the dumped binary for gadgets to launch a common ROP attack.

4.2.4 | COOP

Counterfeit Object-oriented Programming (COOP [P33]) is a sophisticated type of attack that does not chain simple return address one after another, like ROP. Instead, a COOP attack creates fake C++ objects on the stack that resemble existing C++ objects of the victim program. COOP is based on the observation that C++ objects are implemented such that code for each method of a class exists only once but each instance's fields and virtual functions are kept as separate structs in memory. Hence, diverting object instance pointers to counterfeit objects created by the attacker lets a class' methods operate on counterfeit objects. Using typical C++ implementation patterns, execution can chain together several C++ objects to achieve turing completeness.

4.2.5 | Hybrid Approaches

The authors of Mohan et al. [P58] propose a crossing between randomisation approaches, typically used to hide information from the attacker, and CFI approaches that aim at preventing control flow deviations in the first place. The authors' observation is that classical CFI can be subverted given complete process memory disclosure because the set of allowed branch targets (labels) is implicitly stored in the CFI checks in code. They overcome these shortcomings by externalising those CFI checks to hidden memory. Each control flow check is merely a check whether the current target is within the memory area containing all allowed targets for that particular control flow. The memory area is minimised by clustering possible targets close together. The positions of those clusters are then randomised in memory so that for each run the allowed memory area is at a different location and cannot be guessed.

5 MoCFI

Control flow integrity (CFI) is a fundamental concept to inhibit control flow attacks, such as code injection or ROP, at run-time. Control flow integrity was first introduced in 2002 by Kiriansky et al. who enforced CFI in their secure program interpreter (*"Program Shepherding"* [P59]). Later, in 2005 Abadi et al. used a similar concept to instrument code such that control flow instructions check their destination [P53].

In this chapter, the design and implementation of a CFI enforcement framework specifically for smartphone platforms is presented. In particular, the *MoCFI* (Mobile CFI) framework focuses on the ARM architecture, which is the most prominent hardware platform for smartphones. It is implemented to work with Apple's iOS platform for which numerous exploits (so-called *jailbreaks*) exist. The implementation of CFI on ARM is often more involved than on desktop PCs due to several subtle architectural differences that highly influence and often significantly complicate a CFI solution.

1. The program counter is a general-purpose register
2. The processor may switch the instruction set (between *THUMB* and *ARM*) at run-time
3. There are no dedicated return instructions
4. Control flow instructions may load several registers as a side-effect.

The MoCFI framework performs CFI on-the-fly during run-time without requiring the application's source code. Although MoCFI can be deployed to any ARM based smartphone, the implementation was exemplarily done for Apple's iPhone because of three challenging issues:

1. The iPhone platform is a popular target of control flow attacks due to its use of the Objective-C programming language. In contrast, Android is not as prone to control flow attacks because applications are mainly written in the type- safe Java programming language.
2. iOS is closed source meaning that we can neither change the operating system nor can we access the application's source code.
3. applications are encrypted and signed by default.

Despite this prototype being developed for iOS, MoCFI and its principles are also applicable to other ARM-based devices such as Android phones and tablets. The performance as well the average overhead for typical applications and worst-case scenarios was measured. The evaluation shows that MoCFI is efficient and can successfully prohibit a control flow attack.

5.1 | The Contribution to Science and My Part in it

MoCFI is the first CFI framework for iOS devices. It operates on binaries and can be enabled on a per-application basis. This overcomes the problem that for a vast majority of smartphone apps no source code is available.

The MoCFI approach presented herein first recovers the control flow graph (CFG) of an iOS application provided in binary format. Based on this GFG, the control flow verification is done at run-time for all instructions that can change control flow. The prototypical MoCFI implementation is based on in-memory patching of code, which is done by an injected library at load time. This technique makes it compatible with ASLR, static code signing, and encryption. The MoCFI urlibrary, which must be loaded alongside to-be-protected processes can be installed using a jailbreak.

I contributed to the entire MoCFI solution by designing and implementing methods to extract the CFG from a decrypted iOS executable file and by designing and implementing the run-time check mechanism that compares the actual control flow with the allowed control flow based on the statically extracted CFG. Further, I contributed to securing Objective-C message passing with CFI enforcement. Message passing is a characteristic feature of Objective-C, the language in which iOS and its apps are programmed, and allows methods of inherited other objects to be invoked.

5.2 | Primer on ARM and iOS

Since the processor architecture ARM and the operating system iOS influence the design decisions of MoCFI, their peculiarities are briefly described in this section.

5.2.1 | The ARM Architecture

The ARM architecture is very popular with computation- and graphics-intense, low power devices. The ARM architecture is a RISC architecture developed by

ARM Holdings and licensed to chip manufacturers such as Samsung, TSMC or Qualcomm.

This work focuses the very popular 32 bit processor type, because they were state-of-art when this research was conducted in 2012. Since then, 64 bit versions are used since the introduction of the iPhone 5S in September 2013. In the 32 bit processor, each instruction is exactly 32 bits wide and must also be aligned to a multiple of 32 bit memory address. It features sixteen 32 bit general-purpose registers `r0` to `r15`. Some of these registers have a special meaning, as shown in Table 5.1

`r13`	Stack pointer (`sp`)
`r14`	Link register (`lr`)
`r15`	Program counter (`pc`)

Table 5.1: Special Meaning of ARM Registers.

In contrast to Intel x86, machine instructions are allowed to directly operate on the program counter `pc` (called `EIP` on x86).

To reduce the size of instructions, ARM has introduced the 16 bit wide *THUMB* instruction set. Processors supporting *THUMB* mode can actually switch between *ARM* and *THUMB* instructions while executing code. To use the correct instruction set during execution, function calls and returns implicitly encode whether they are targeting *ARM* or *THUMB* code [D4]. In ARM assembly, a function is called by either a `BL` instruction (Branch with Link) or `BLX` instruction (Branch with Link and eXchange). `BLX` can also take a register as an argument (indirect call). Typical for function calls, `BL` and `BLX` both save the return address, however, not on the stack. Instead, the return address is stored in the link register `lr`. At the end of a function, execution returns by setting `pc` $\leftarrow$ `lr`. For nested function calls, the value of `lr` is usually pushed on the stack when the called function is entered.

5.2.2 | Apple's iOS Operating System

iOS is the name of Apple's mobile operating system that has been unveiled in 2007 with the first iPhone smartphone. Since then, it has become their default operating system for mobile Apple devices such as iPhones, iPads, and iPods. The $W \oplus X$ data execution prevention (cf. chapter 3) has been enabled by default since iOS 2.0. Additionally, since iOS 4.3, ASLR is enabled by default to randomise the address space of each started process. To reduce the attack surface for malicious software, iOS is designed to only run applications (Apps) that are digitally signed by Apple. An interesting security feature of iOS is dynamic *Code Signing Enforcement* (CSE) at run-time to prevent the injection of new code [P77]. By default, stored programs are encrypted and are only decrypted before execution. This encryption and signature would prevent the traditional CFI approach [P53, P54] as it performs changes on the executable file.

5.3 | The Design of MoCFI

From a high-level point of view, MoCFI operates in two phases:

Preprocessing Phase. In this first phase, which only has to be done once per application, all necessary information is extracted and packed with the original App for later use. First, the encrypted binary executable file gets decrypted so that it can be disassembled. The assembler representation is then used to extract the CFG and also saves the position of control changes (outgoing edges in the CFG) to be used with run-time enforcement later.

Run-Time Enforcement Phase. This phase takes place every time the App is executed. First, during load time, the executable is patched in memory to incorporate the CFI checks. During run-time, the App is monitored by means of an Inlined Reference Monitor (IRM [24]), which checks for each control flow change whether it is allowed according to the statically extracted CFG.

The interaction between the two phases is depicted in Figure 5.1.

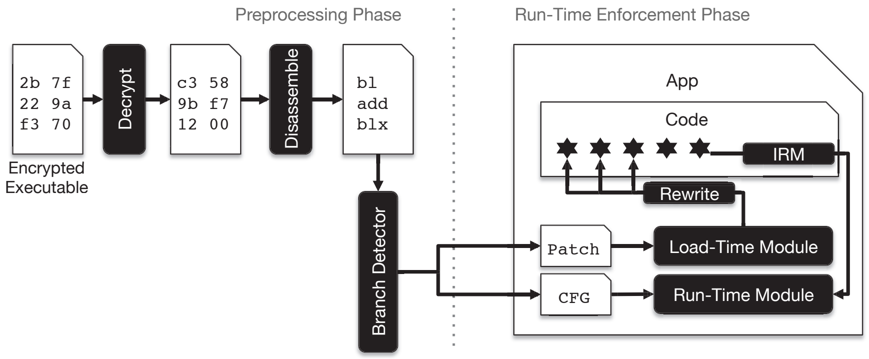


Figure 5.1: Overview showing the two phases of MoCFI.

In more detail, the individual steps that make MoCFI work as follows:

Decrypt. To protect iOS Apps from piracy, they are encrypted by default. In order to operate on the unprotected binary executable file, it must be decrypted first. The unencrypted file content (code and data) is obtained by dumping the running process of an App once, based on the method presented in *PiOS* [P78].

Disassemble. Then, the binary machine instructions stored in the code segment needs to be disassembled to be able to operate on an instruction basis. A proven and reliable disassembler is the *IDA Pro* disassembler by Hex Rays [L14]. IDA allows the use of scripts written in C or Python to operate on the reconstructed instructions and their control flow.

Branch Detector. I created an IDA script that is able to automatically extract the CFG of a given decrypted iOS App. For vertices of the CFG, indirect branch instructions, indirect jumps, calls, and function returns are considered. Their target addresses are variable at run-time and must therefore be protected. Moreover, direct function calls with hard-coded targets are also included to verify the return address when the callee returns. Including direct jumps in the CFG, however, is unnecessary as the target address for these instructions are hard-coded and hence cannot be manipulated by an attacker.

Patch File. Alongside the CFG, *patch* information is created that contains the instruction's relative offset into the code segment, it's length, type and precalculated target(s). This patch file is later used at load time to patch the loaded code with calls into the MoCFI run-time that do the actual CFI checks. The executable file itself cannot be patched in-place by means of static binary translation, because it is signed and encrypted and even if the encryption key were known, the static modifications would invalidate the signature. Using separate CFG and patch files leaves the original App unmodified and thus Apple's CSE can still be used to protect the App. The patch file and the CFG are saved inside the App's package and are not included in the App's signature since this cannot be changed.

Run-Time Enforcement Phase. To achieve the goal of performing CFI checks at *run-time*, a MoCFI protected App is modified in memory during *load time*. To this end, the MoCFI library has to be injected into an App's address space (see details in section 5.4). The MoCFI library performs in-memory binary rewriting after it has been loaded but before execution starts.

Dynamic Binary Rewriting. During load time, the MoCFI library uses the *patch* file to quickly find and overwrite the control flow instructions that correspond to edges in the CFG. These patches redirect control flow into the MoCFI run-time part for each edge in the CFG. We achieve this by replacing all control flow instructions with a single instruction: the so-called *dispatcher instruction*. The dispatcher instruction redirects the control flow to a code section where the CFI checks reside, namely to the *run-time Module* of our MoCFI shared library.

Inline Reference Monitor (IRM). This inter-positioned call instead of the original call is known as *Inline Reference Monitor* (IRM) [P1]. It adds the possibility to have a centralised monitor that is consulted for each edge transition in the CFG. If the intended target is allowed, control will be transferred to the correct and valid vertex of the CFG without the App noticing. Otherwise, an exception will be thrown and the App is terminated immediately. For the IRM to determine if a target is valid, it checks the type of control flow and its intended target against a valid target or a list of valid targets. While the target address of an indirect jump or call can be validated against a statically extracted list of outgoing edges, the validation of function returns requires special handling. Returns are dynamically determined by the value of the `lr` register and cannot be predicted ahead of time. Therefore, MoCFI uses the concept of a shadow stack, which stores the only valid return target whenever a function is called [P51].

5.4 | Implementation

MoCFI was implemented for the then-new iOS version 4.3.2 and is made up of three different parts:

- **gdb script** for decrypting any encrypted application.
- **IDA Pro IDC Script** for generating the CFG and patch files.
- **iOS shared library** to be attached to each MoCFI-protected process.

The *Pre-Processing Phase* is implemented as IDA Pro 6.0 IDC script (C language) and generates the CFG for the application as well as the needed patch file. The *Run-time Phase* library that is loaded in each process is written in `C++` using Apple's *Xcode* 4.0 for iOS. The IDA IDC script consists of 842 lines of code while the MoCFI library consists of 1,430 lines of code.

5.4.1 | Pre-Processing Phase

Decrypt. Before the executable file can be loaded into the IDA disassembler, it needs to be decrypted first. Apple's Mac OS X and iOS operating system share the same Darwin kernel and hence use the same binary file format for executable files and for shared libraries. Their *Mach-O* file format supports the description of different sections (e.g. code and data) within the file but also stores a variety of meta information alongside. Specifically, the Mach-O file stores cryptographic signatures in the `LC_SIGNATURE_INFO` command [P78]. When an application is started that carries this information, the iOS loader verifies the stored signature by recalculating the signature and comparing it against the information stored in this section. If they do not match, the application is not loaded and terminated. Moreover, Mach-O binaries also support the so-called `LC_ENCYPTION_INFO` command, which specifies whether an executable is stored in encrypted form. If this is the case, the loader retrieves the decryption key from the system's secure key chain and places the decrypted file contents in memory so that they can be executed. MoCFI uses this fact to retrieve the cleartext executable file contents by dumping the App's memory contents. This dumping process has been automated using the *gdb* debugger, which loads a MoCFI gdb script, starts a new App and attaches to the newly created process. The script then matches the loaded cleartext sections in memory with the to-be-loaded sections stores in the Mach-O file and generates a new Mach-O file with the encrypted sections being replaced by their cleartext equivalents. This Mach-O file can be read by the IDA Disassembler natively.

Disassemble. IDA Pro uses a rather sophisticated disassembler whose recovery rate of instructions is improved by guiding disassembly recursively through the code as portions of the code as discovered. This method is superior to fall-through disassembly, which cannot recover code and data very reliably. IDA also automatically and reliably detects whether *ARM* or *THUMB* instructions are used based on how they are called. The MoCFI IDC script loaded into IDA identifies all relevant branch instructions and saves their type, relative offset, and their length in the patch file. These branch instructions can be divided in two categories:

(1) Instructions that encode a hard-coded control flow destination as an immediate value in the instruction itself.
Example: `bl 0x407ac`

(2) instructions that take the target address from the value of a register.
Example: `bx lr`

Category (1) branches are trivial to handle on iOS because the $W \oplus X$ environment does not allow code to be changed. Hence, as an optimisation step we do not consider category (1) branches in the CFG. Category (2) branches on the other hand, are more involved to resolve as the possible register value needs to be determined statically by means of program slicing [P79]. Program slicing back-tracks the instructions preceding the branch instructions in question with respect to whether and how they modify the register value the branch instruction uses. This way, the value of the register, and hence the the branch target, can be determined. Also more complex cases, e.g. `LDR pc,[r2,r3,LSL #2]`) can be solved by back-tracking the used registers. The `LDR` instruction loads a register value from another register or from a complex computation of other registers. In this example, the program counter `pc` is loaded from $pc \leftarrow \texttt{r2} + \texttt{r3} \cdot 2^2$. This effectively changes control flow to the calculated address. If both `r2` and `r3` can be back-tracked, the correct value for `pc` can be calculated.

Indirect Branches and Heuristics. Sometimes, it is not possible to back-track the register to a single value, for instance when the register is loaded from memory. In this case, MoCFI uses heuristics to narrow down the possible control flow destinations. Therefore, the reconstructed CFG is always a superset of the actual CFG of the program because it cannot be accurately reconstructed in all cases. Please note, that this is a general limitation of CFG re-construction methods and not specific to the MoCFI approach. The heuristics MoCFI employs to reconstruct register values are based on typical patterns that the LLVM iOS App compiler incorporates in the executable file.

```
0x1000: MOV    r2, 0x2000
0x1004: ADD.W  r2, r2, r3, LSL#2
0x1008: MOV    pc, r2
0x2000: B.W    0x3000
0x2004: B.W    0x3100
0x2008: B.W    0x3200
```

Listing 5.1: ARM Indirect Jump Using a *Jump Table*

Listing 5.1 is a common compiler-generated pattern to optimise switch statements. Depending on the value of `r3`, the possible control flow targets (`pc`) are `0x3000`, `0x3100` or `0x3200`. The MoCFI analysis can recover these possibilities because the compiler pattern is detected and all possible targets are extracted. Patterns that have not been trained (e.g. handwritten assembler code) cannot be detected. In those cases, the control flow is constrained to the understanding of possible control flows. For instance, indirect calls must target the beginning of functions. Even though it is still possible to call arbitrary functions, the target control flow cannot land inside a function body. Similarly, indirect jumps (BX) are only used to jump inside the scope of a function and hence can be confined to the current function's boundaries. Even though there is no technical need for these restrictions, the C, C++ and Objective-C languages restrict control flow to the scope of one function, except for function calls. Hence, MoCFI assumes that the scope of indirect jumps is bounded by the enclosing function.

Objective-C Peculiarities. Traditional CFG generation techniques also need to be extended for iOS applications due to a peculiarity of Objective-C. Internally, any method call of an Objective-C object is resolved to a call to the generic message handling function `objc_msgSend()`. The name of the actual method (called *selector*) is given as a parameter. Consequently, MoCFI tracks these parameters in the CFG generation phase and includes them in the CFG. Otherwise, an attacker could modify the method parameters of `objc_msgSend()`, thereby diverting the control flow to an invalid method. The parameter extraction of `objc_msgSend()` was loosely based on the work presented in PiOS [P78].

5.4.2 | Run-time Phase

Most UNIX-based operating systems support the injection of libraries by providing the environment variable `LD_PRELOAD` that is checked by the OS loader when initialising a 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 [D5]. By letting this variable point to the MoCFI library, it gets loaded when a new process is created. The variable is inherited to child processes, which is handy in iOS because it allows MoCFI to be applied to all Apps started by a parent process. In iOS, the home screen that shows all installed Apps is a process called *Spring Board*. Forcing all Apps started by Spring Board to be protected by MoCFI is simply a matter of setting `DYLD_INSERT_LIBRARIES` for the Spring Board process once. Because of the way `DYLD_INSERT_LIBRARIES` works, MoCFI is initialised *before* any other dependency of the program is loaded but *after* the signature of the App has been verified.

Load-Time Binary Rewriting. When control is transferred to the MoCFI library for the first time, no other code of the App has executed yet. This is ideal for patching the loaded code in memory such that it incorporates the necessary calls to the IRM, which performs the actual CFI checks.

For the patching to take place, MoCFI first locates the correct patch file inside the App bundle. Then, it rewrites the loaded code in memory according to the information stored in the patch file. Since iOS enforces a $W \oplus X$ memory protection policy, the code segment is actually not writable and also cannot be turned writable again by using the `mprotect` system call. Instead, MoCFI creates a copy of the code segment using the `mmap` system call. Then, all relevant instructions are patched using the meta information stored in the patch file. As a last step, the patched code is `mmap()`'ed to the original position of the code, thereby overwriting the write-protected, unmodified code. Please note, that the presence of `mmap` does not give an attacker the opportunity to subvert MoCFI by overwriting code. To use `mmap`, an attacked would already need arbitrary code execution, which is prohibited by MoCFI.

Dispatcher Instructions. As described earlier, MoCFI's binary rewriter overwrites the relevant control flow instructions with *Dispatcher Instructions*. The different dispatcher instructions accommodate for the fact that they replace different control flow instructions, but they all transfer control to the MoCFI library (see Figure 5.2). They are used as bridges between the application that MoCFI protects and the MoCFI library itself.

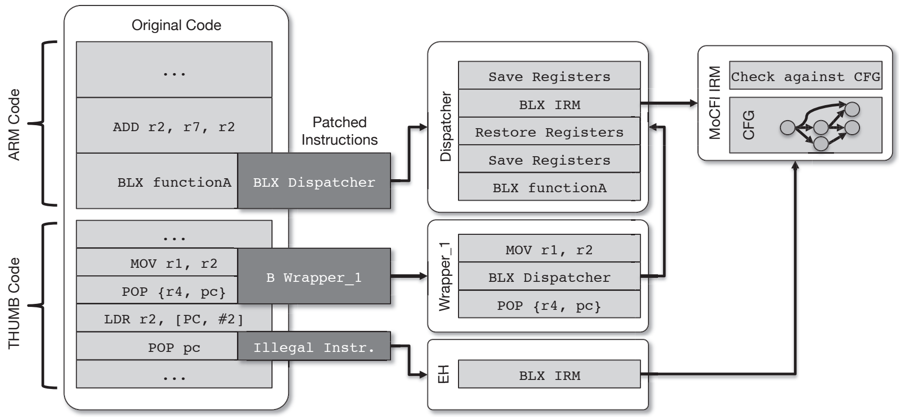


Figure 5.2: The *Dispatcher Instructions* overwrite original control flow instructions to build a bridge to the MoCFI Inline Reference Monitor (IRM)

As shown in Figure 5.2, MoCFI uses either a **Static Dispatcher**, a number of **Wrappers** or an **Exception Handler (EH)** to call the actual MoCFI IRM, which performs the CFI checks.

Static dispatcher. This dispatcher is part of the MoCFI library and only exists once. It is the simplest way to call the IRM. The dispatcher saves the registers that could be modified by the IRM and calls the IRM. The IRM then checks if the intended target is reachable from the current vertex in the CFG via single outgoing edge. If yes, this constitutes a legitimate control flow and the IRM simply returns. Otherwise, the program terminates itself. After returning from the IRM, the dispatcher restores the registers and transfers control to the actual intended target as if the check did not happen. The dispatcher is slightly different for calls to external libraries.

Wrapper. The wrapper is needed when the dispatcher instruction is larger than the instruction it replaces or when the to-be-replaced instruction has side-effects. The first case may happen for *THUMB* code as the needed `B/BL/BLX` instruction to call the dispatcher is always 32-bits wide, while the instruction it replaces might only be 16 bits wide. This necessarily results in overwriting two 16 bit *THUMB* instructions, which need to be preserved in order to remain the original semantics of the program. To this end, MoCFI has allocated an individual Wrapper for each of those cases during its load time patching. The wrapper stores both instructions that have been overwritten and also calls the Static Dispatcher. In the example shown in Figure 5.2, the instruction `MOV r1, r2` gets overwritten by the call to `Wrapper_1`. Instead, `Wrapper_1` then calls the missing `MOV` instruction, followed by a call to the dispatcher, which then checks the actual control flow instruction for a valid target. If the IRM and the Static Dispatcher return successfully, the actual copied control flow instruction is executed inside `Wrapper_1`. This is necessary on the ARM platform, as control flow instructions may have side-effects, such as loading registers. In the example in Figure 5.2, the patched control flow instruction `POP {r4, pc}` loads both, the program counter `pc` and register `r4`, from the stack.

Exception Handler. An exception handler (EH) is required, if it is not possible to overwrite two consecutive 16 bit instructions. For example, if the preceding instruction uses the program counter `pc` or is itself a branch. In the example shown in Figure 5.2, the preceding instruction `LDR R2,[PC,#2]` references the program counter `pc`. Hence, we cannot simply overwrite that instruction and execute it inside the Wrapper, as this would change the value of `pc`. Instead of complicated corrections of the `pc` value, we opted for replacing only the control flow instruction with a 16 bit illegal instruction. This way, iOS transfers control to the exception handler for illegal instructions, which the MoCFI load time Module has registered before. However, this simple method is deliberately avoided when possible, as it introduces a significantly higher overhead than the Wrapper approach. The EH then calls the IRM function directly because there is no need to restore registers as the iOS exception handler takes care of that when returning.

The MoCFI IRM. In order to check intended control flow against legitimate control flow stored in the CFG, the correct vertex must be retrieved from the CFG. MoCFI realises a quick look-up by storing the CFG as vertex-edge pairs that are sorted by their memory address. Hence, the CFG can be indexed by the `pc` to retrieve the correct vertex. A flow chart of the different cases that the IRM handles internally is depicted in Figure 5.3. The different branch types (indirect jumps, indirect calls, direct calls and Objective C msgSend) are handled differently and hence have dedicated validation routines.

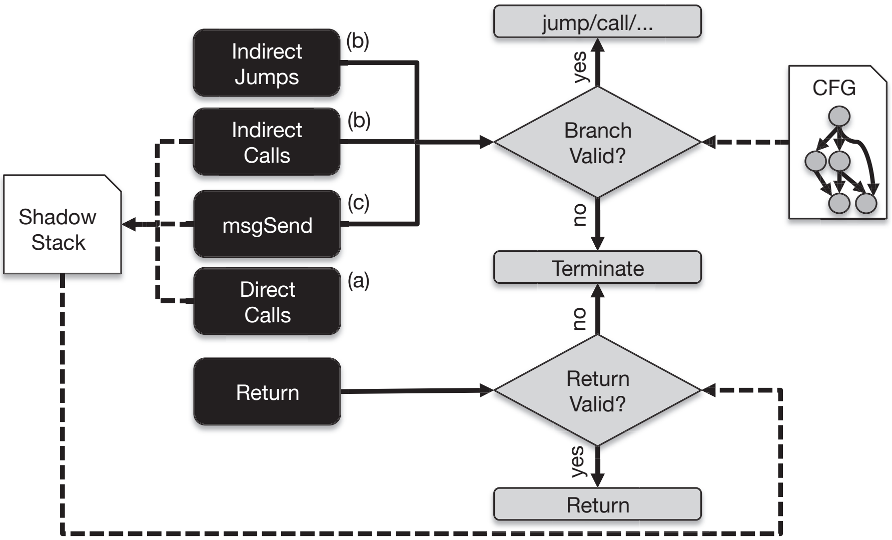


Figure 5.3: The MoCFI Inline Reference Monitor (IRM), which performs the actual checks of whether control flow is still on the pre-determined legitimate path.

Function Calls and Returns (a). To prevent return-oriented attacks, MoCFI uses the shadow stack principle that has been introduced by StackGhost [P60]. Whenever the program invokes a subroutine (through a direct, indirect, or dispatcher call), MoCFI saves the return address on a dedicated shadow stack. Upon function return, MoCFI compares the return address the program intends to use to the address stored on the separate shadow stack. Since function calls (through `BL` or `BLX`) automatically store the return address in the `lr` register, MoCFI simply pushes `lr` onto the shadow stack. One such stack is maintained per application thread.

Indirect Jumps and Calls (b). The possible jump targets for indirect jumps and calls have either been calculated during the *Pre-Processing Phase* or have been confined to the scope of the current function. In the first case, the pre-calculated values are compared to the intended control flow of the instruction that MoCFI intercepted. In the most complex and versatile form, the instruction is of the form `LDR pc,[rX,rY,LSL#Z]` which loads `pc` according to the given register values: $\texttt{pc} \leftarrow \texttt{rX} + \texttt{rY} \cdot 2^{Z}$. Consequently, MoCFI checks the current value of the registers according to the above equation and matches them against all outgoing edges of the corresponding CFG vertex. The instruction itself does not need to be disassembled to determine which registers are used. Instead, this information is saved as meta info attached a vertex in the CFG. For simpler indirect jumps such as `MOV pc,rX` and indirect calls (`BLX rX`), MoCFI only checks the content of `rX`.

Objective C `msgSend()` Calls (c). Dispatcher calls via the `objc_msgSend()` function work similar to indirect calls. However, instead of a register, they use the function's name (*'selector'*) and an instance of a class to refer to a function's implementation. This information is used by `objc_msgSend()` to find the correct memory address at run-time and then calls the appropriate address. To protect those Objective C calls as well, MoCFI checks the supplied parameters *selector* and *class instance*. The *selector* is a pointer to a string and is compared to the correct string that has been extracted by the *Pre-Processing Phase*. To avoid a string compare for each function call, MoCFI uses the same technique as the `objc_msgSend()` function: a cache. The use of a cache is safe since all possible strings are stored in a read-only memory area. Therefore, it is sufficient to compare the string once and save its memory address as the only comparison in the cache.

5.5 | Security Evaluation

MoCFI adheres to the goal of detecting deviations from the control flow at run-time from the known-good control flow. Since iOS enforces $W \oplus X$, a memory page cannot be writable and executable at the same time. Hence, it suffices to check branch targets that depend on the value of a variable since those can be changed during run-time. These include indirect branches and returns from function calls, as they use an address popped from a potentially tampered stack. As we check each such instruction, an adversary cannot subvert the control flow without MoCFI noticing. However, not all valid, known-good targets can be calculated in advance during the static analysis phase. If this is not the case, heuristics confine the control flow to the scope of the current function. This somewhat loosens the security guarantees in case of indirect branches. Fortunately, for the majority of tested applications, indirect branches are used in conjunction with jump tables (see section 5.4), and can be resolved during the static analysis phase.

Since MoCFI performs binary rewriting after the iOS loader has verified the application signature, it is compatible with application signing. On the other hand, the load time modifications of MoCFI are not compatible with the iOS CSE (Code Signing Enforcement) run-time model (see section 5.2). CSE prohibits any code generation at run-time 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, the attacker 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. 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.

For an IRM to be secure, (i) it must mediate all events relevant to the security policy being enforced, (ii) its integrity must be protected from subversion by applications, and (iii) its presence must be transparent to applications [P80]. (i) is ensured by patching the code to redirect to the IRM for all control flows present in the CFG. (ii) is ensured because the App cannot subvert the involvement of the IRM without patching its own code or adding code during run-time. Both can be prevented by disabling the appropriate system calls after load time. (iii) is guaranteed since the involvement of the IRM has no side effects if the control flow target was determined to be legitimate.

5.6 Performance Evaluation

In order to evaluate the performance of MoCFI, the iOS benchmark tool *Gensystek Lite2* [L15] was used. Additionally, micro benchmarks for control flow transitions and algorithms that heavily use control flow transfers were tested.

One such algorithm is *quicksort*, as it recursively calls itself. MoCFI is only applied to the main application code and not to the loaded libraries. However, the benchmark tools we apply perform most part of the computation within the application.

A bar graph of the Gensystek benchmarks is shown in Figure 5.4. The number underneath each benchmark name represents the slowdown factor of that particular benchmark. Surprisingly, the FPU/ALU, PI calculation, and the RAM memory read-/write) benchmarks experience the highest overhead (3.85x and 5x, respectively). The overhead (slowdown greater than factor 1) for the remaining benchmarks is very low and ranges between 1% and 21%.

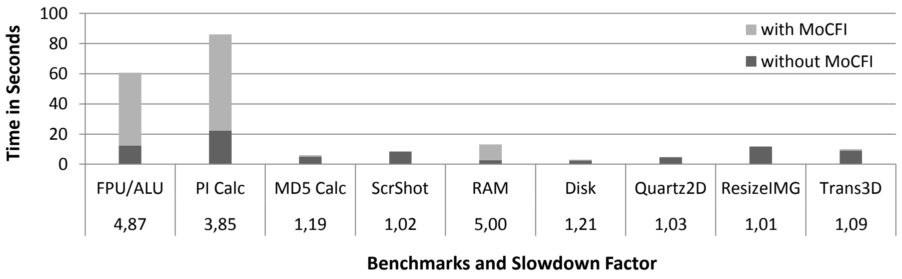


Figure 5.4: Gensystek Benchmark Result with/without MoCFI

In order to approximate an upper boundary for the MoCFI performance impact, the quicksort algorithm was used. The quicksort implementation makes use of recursion and continuously calls a compare function which consists of only 4 instructions and one return. Therefore, MoCFI frequently performs a control-flow check in this worst-case scenario. The resulting worst-case slowdown factor is 12x.

In order to evaluate the overhead of an instruction that has been replaced by a CFI check, the execution time of three typical instructions and their replacement by MoCFI has been measured. For the exemplary case of Function Calls and Returns, the actual function bodies (α) are subtracted from the time measurement between call and return (β). The actual execution time is therefore $\beta - \alpha$. This calculation of the overhead per replaced instruction is depicted in Figure 5.5.

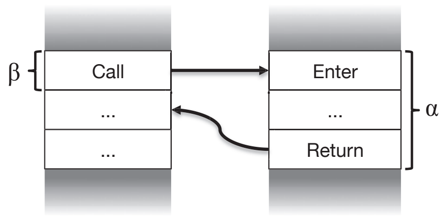


Figure 5.5: Function overhead calculation.

When running with MoCFI, the measurements (α' and β', respectively) are set in relation to the measurement without MoCFI. Hence, the instruction slowdown factor φ for one function replaced by MoCFI is:

$$\varphi = \frac{\beta' - \alpha'}{\beta - \alpha}$$

For our tests, all the measurements were conducted 10,000 times and averaged. However, in a typical program, instructions that have to be checked by MoCFI are surrounded by other instructions. For n instructions in between, MoCFI only has to be called every $(n + 1)$-th instruction. The total slowdown ψ is therefore:

$$\psi(n) = \frac{n + \varphi}{n + 1}$$

The overhead $(\psi(n)-1)$ as a function of n (instructions between MoCFI checks) is plotted in Figure 5.6.

5.7 | MoCFI Conclusion and Limitations

The MoCFI proof-of-concept demonstrates that a protection technique hat heavily relies on the understanding of program and control flow internals is still possible for binary applications. MoCFI protects an existing binary at the last link of the chain Fault $\rightarrow$ Error $\rightarrow$ Failure by monitoring the symptoms of an exploited vulnerability. Deviations from expected and allowed behaviour are then caught and the hijacked application is stopped before the changed behaviour can cause any harm.

However, the current MoCFI implementation does not detect attacks exploiting the exception handler. An adversary can overwrite pointers to an exception handler and then deliberately cause an exception (e.g. by corrupting a pointer before it is dereferenced). This is possible because GCC pushes these pointers

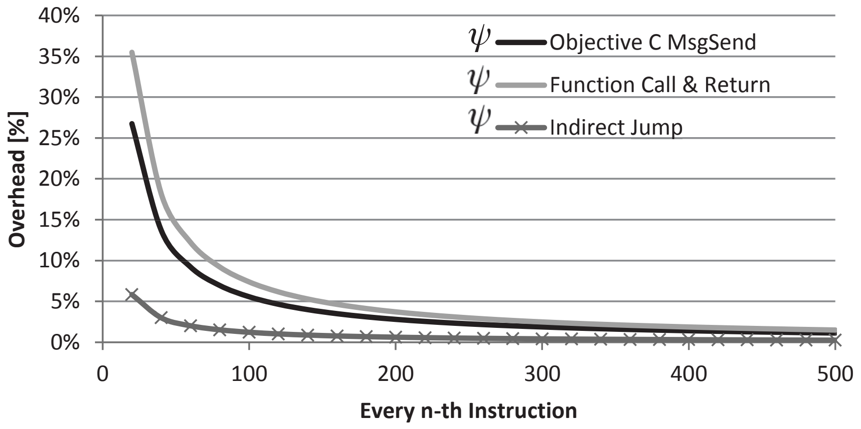


Figure 5.6: Overhead as function of n instructions between MoCFI calls.

on the stack on demand. Similar problems have already been solved on other platforms, such as on Windows [D6].

MoCFI currently does not protect shared libraries, which an adversary may exploit to launch a control flow attack. However, extending MoCFI accordingly is straightforward, because no new conceptual work is need. Moreover, shared libraries were no allowed in iOS Apps as this would introduce the possibility to load code, which is discouraged by the *Apple App Store Review Guidelines* [D7]. However, with the release of iOS version 8.0 Apple has allowed App developers to use so-called *Frameworks* in their Apps, which essentially is a packaged shared library. Consequently, MoCFI detects when execution is redirected to a library and disables the return address check for functions that are directly invoked by the shared library.

An orthogonal solution to control flow integrity is to protect the root cause of code reuse and not just its symptoms. For an attack to be successful, an attacker must know useful address in the to-be-exploited program. A method of hindering an attacker to learn useful addresses is to randomise a running program in small pieces. Such fine-grained randomisation is presented in the next chapter.

6 Fine-Grained Randomisation

The idea of *Address Space Layout Randomisation* (ASLR) is to deprive the attacker of the necessary knowledge about gadget addresses. However, as already described in the background chapter 3, ASLR moves the entire code segment en bloc and as a result the relative locations of objects and functions do not change. If the adversary additionally knows which object or function has been leaked, he knows the address of that object/function. A single leaked address is hence enough for an attacker to revert the layout for an entire ASLR process. *A single* leaked address from the code segment is enough to calculate the address of *every* instruction inside a process. This effectively enables an attacker to infer all other objects or functions relative to the leaked address because the relative distances between functions stay exactly the same. This puts an attacker back in the position prior to ASLR where he could choose gadgets at his discretion, ultimately defeating ASLR.

The traditional ASLR as implemented in modern, off-the-shelf operating systems has several **D**rawbacks.

D1: Only entire modules (i.e. the main executable and each library) are randomised. The modules themselves are treated as atomic units.

D2: The achieved entropy is much lower than the theoretically possible 32 bit on a 32-bit system.

While **D1** is a conceptual flaw, the limited entropy **D2** results from several legacy limitations that the operating system has to adhere to when loading a module at a *'random'* address. The following is an in-depth look at both drawbacks.

D1: For executables or shared libraries to benefit from ASLR under Linux, they must be compiled using position-independent code (PIC) and linked such that the resulting ELF file type is `SYM` (shared object). Interestingly, the load address will be fixed to zero. Consequently, code, data, and other loadable segments have a load address relative to zero, or in other words, their load address is identical to the offset of that section in the ELF file.

```
$ readelf -h /usr/sbin/sshd
ELF Header:
Class:        ELF32
Type:         DYN (Shared object file)
Program Headers:
 Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 PHDR          0x000034 0x00000034 0x00000034 0x00120 0x00120 R E 0x4
 INTERP        0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
     [Requesting program interpreter: /lib/ld-linux.so.2]
 LOAD          0x000000 0x00000000 0x00000000 0xc7dbc 0xc7dbc R E 0x1000
 LOAD          0x0c859c 0x000c859c 0x000c859c 0x02130 0x07d50 RW  0x1000
 DYNAMIC       0x0c9750 0x000c9750 0x000c9750 0x00158 0x00158 RW  0x4
 NOTE          0x000168 0x00000168 0x00000168 0x00044 0x00044 R   0x4
 GNU_EH_FRAME  0x0b3f50 0x000b3f50 0x000b3f50 0x02714 0x02714 R   0x4
 GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
 GNU_RELRO     0x0c859c 0x000c859c 0x000c859c 0x01a64 0x01a64 R   0x1
```

Listing 6.1: PIC segments of the SSH daemon

The kernel's ASLR implementation then adds a random value to that load address, thereby shifting the entire module (shared object) in memory. This results in the fact that relative distance between instructions inside the code segment or relative distance from code to data always stay the same – regardless of the random load address. This fact is actually used in PIC to address data: the absolute address of data to be referenced is calculated by adding the offset (known at compile time) to the absolute address of current execution: the program counter (`EIP`). An attacker can also exploit this fact when she knows a correct run-time address of a particular object. Suppose a program suffers from a vulnerability that leaks a pointer. Then, an attacker can add the known offset to a useful ROP gadget to calculate its absolute address. This is crucial information for mounting a ROP or return-into-libC attack by exploiting yet another vulnerability in the program. A leaked pointer occurs easily in a C program if array boundaries are not checked properly.

```
int sizes[10];
int * size_ptr = sizes;

void retrieve_size(unsigned int index)
{
    return sizes[index];
}
```

Listing 6.2: Exemplary program vulnerable to pointer leaks

An attacker can exploit this code to deliberately read beyond the end of the array, thereby retrieving values of objects stored subsequent to the array. In the example above, a supplied parameter of `index = 10` would read the subsequently stored global variable size_ptr. This pointer holds the address of the array sizes, which is stored in the loadable segment .bss. After the attacker knows the address of sizes, she can calculate the distance to a useful ROP gadget by inspecting her local copy of the vulnerable executable. The distance between size_ptr and the desired gadget is the same for her copy and any other copy. Adding that distance to the leaked pointer results in the absolute address of the desired ROP gadget in the (possible remote) target process.

D2: The possible *'random'* load addresses are reduced by the granularity of the page addressing system (multiple of 4 KiB pages) and by the fact how mapping large memory areas work. In fact, in the latest Linux kernel 4.1, for an mmap()'ed address, only 8 bits (bits 12 through 20) are randomised for 32-bit processes. Similarly, only 28 bits (bits 12 through 40) are randomised for 64 bit processes.

```
unsigned long arch_mmap_rnd(void)
{
    unsigned long rnd;

     //  8 bits of randomness in 32bit mmaps, 20 address space bits
     // 28 bits of randomness in 64bit mmaps, 40 address space bits
    if (mmap_is_ia32())
        rnd = (unsigned long)get_random_int() % (1<<8);
    else
        rnd = (unsigned long)get_random_int() % (1<<28);

    return rnd << PAGE_SHIFT;
}
```

Listing 6.3: Linux Kernel 4.1 (/arch/x86/mm/mmap.c)

In this chapter, a much higher level of granularity in randomisation is achieved by also randomising contents of loadable sections, especially of code. This is achieved by the means of a binary rewriter that was written specifically to address the aforementioned drawbacks **D1** and **D2**. The binary rewriter changes the code such that the distance between any two instructions in the code changes. This renders information about relative distances to each other useless. Consequently, an attacker cannot predict the address of a particular instruction given a leaked address of another instruction.

The binary rewriter presented in this chapter was implemented in C++ and supports the Linux ELF executable format. It's core rewriting techniques are agnostic of the processor architecture, however the differences between instruction sets and how processors address memory need an architecture specific layer on top. A layer that supports Intel's common x86 32 bit architecture was implemented. A second architecture, the widespread ARM architecture is supported, however not fully, e.g. no *THUMB* code.

6.1 The Contribution to Science and My Part in it

In traditional ASLR, the offset between all loadable sections (`.text`, `.data` and `.bss`) stays the same so that position-independent code can reference data relative to the load address of code. Instead, the binary rewriter presented in this thesis is designed to transform binary executables at load time by randomising the base addresses of all loadable sections. Furthermore and most importantly, it randomises the code (`.text` section) to a high degree – if necessary down to instruction level granularity. This makes predictions concerning relative distances of instructions to each other infeasible, thereby preventing pointer-leak-based attacks. More precisely, the binary rewriter presented herein made the following contributions when the paper was published:

1. Code segments are randomised on a fine-grained (e.g. instruction) level,
2. Randomisation does not conflict with code signing, i.e. no static binary rewriting is used resulting in invalid signatures due to changed executable files,
3. Instead, randomisation is performed at load time of the program such that each process start exhibits a different memory layout.

In order to avoid a prior offline static analysis phase, the binary rewriter must perform all necessary rewriting duties while the process is being loaded. Consequently, this process must be very quick in order not to introduce any noticeable delay at process start. To this end, I designed the rewriter in three very performant phases:

1. **Phase a**: Selective disassembly that only disassembles instructions that would change when stored at a different address.
2. **Phase b**: Reference discovery to create an internal reference graph representation that describe which instruction depends on whose other memory address.
3. **Phase c**: Assembly of instructions which are connected by the reference graph while the irrelevant instructions are treated as binary data.

The necessary randomisation is then performed between phase a and phase b. Any changes made to the program are automatically reflected in phase c because the reference graph keeps track of which instructions need to be adjusted for their new address.

Besides load time performance, run-time performance is also an important factor that I took into consideration. An instruction granularity randomisation has a high performance overhead as every single instruction must be connected with an explicit control flow instruction to keep the original semantics of the program. To optimise performance, a trade-off between the desired entropy and run-time performance is dynamically calculated for each program. The resulting granularity of randomisation is chosen such that the entire program still features a high entropy. The desired entropy, and therefore the resulting granularity, is adjustable by the user.

Existing binary rewriting approaches are not tailored to do in-memory rewriting and are hence not suitable for a fast fragmentation and randomisation. On the other hand, for the required goal of splitting code into pieces and shuffling it, only a subset of the features of a full-blown binary rewriter are required. This is why I chose to design and implement a binary rewriter from scratch that is capable of doing in-place rewriting on process start-up. Its implementation is detailed in section 6.4. It is not based on any prior software, not even the x86 disassembly routines. I have also designed all necessary algorithms, such as the creation of the CFG, representation of instructions in an intermediate language, randomisation algorithms, optimisation algorithms, process loading and executable and shared library parsing. The binary rewriter is also the basis for the *Oxymoron* rewriter presented in chapter 7.

Benchmark results obtained using SPEC CPU2006 (see section 6.7) demonstrate that the run-time performance overhead induced by the randomisation is a mere 1.2%. Another additional overhead is the start-up time, as process loading goes through an additional step of on-the-fly binary rewriting. This rewriting achieves a throughput of 5000 KiB/s, i.e. a 10 KiB program takes only 2 ms longer to load.

The prototypical implementation presented in this thesis relies on relocation information to guide the partial disassembly. Otherwise, instructions that reference data or other code (i.e. pointers) cannot be recovered reliably. Note that Smithson et al. [P81] have presented a patented binary rewriting solution, which supposedly does not require relocation information.

6.2 | Design

The randomisation of all code (main executable and loaded libraries) takes place at load time, before execution of the process starts, but after all the necessary code has been loaded into the address space by the linker (phase (a) in Figure 6.1). In order to randomise the individual code segments and to intermix them (all library code and executable code), all references in the code are extracted in phase (a). Then, in phase (b) the code is cut into arbitrarily small pieces, which are then shuffled. In the last phase (c), all code pieces are transferred back into x86 machine code and each loadable segment (e.g. `.text`, `.data` and `.bss`) is loaded at a random address.

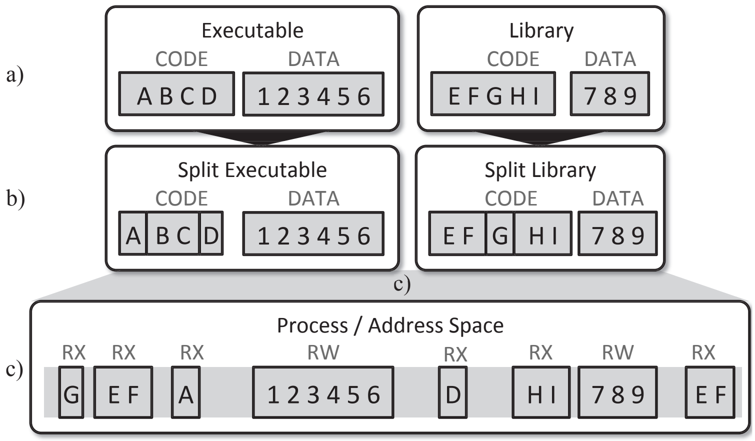


Figure 6.1: High-level process of the randomisation

6.2.1 Phase (a) – Intermediate Representation

These individual phases are quite challenging and call for various corner cases to be taken into account. The most prominent challenge is the fact that code cannot simply be re-arranged without breaking its semantics since all addresses are hard-coded and therefore control flow and data access would point into nonsense. Identifying all references and keeping track of references is hence the main duty of the binary rewriter. The foundation for automatic reference bookkeeping is done in phase (a). This is achieved by transforming binary code back into a representation that uses symbols instead of hard-coded addresses. This higher-level abstraction, or intermediate representation, allows for instructions to be inserted while keeping the semantics of the underlying code intact. Using a higher level intermediate representation makes code abstract enough as to work with it without destroying its semantics.

The following is an example of disassembled x86 binary code and how it is transformed to intermediate representation.

```
0x8048000: e8 00 00 00 00     call   0x8048005
0x8048005: 5b                 pop    %ebx
0x8048006: 81 c3 27 70 03 00  add    $0x37027,%ebx
0x804800c: 8b 03              mov    (%ebx),%eax
```

Listing 6.4: Disassembly of the input program

When this piece of code is moved in memory, the referred absolute addresses (`0x8048005`) will no longer be correct. The same applies to relative addressing of data (in this case `eip+0x37027`), since, e.g. inserting instructions changes the distance between code and data. On the one hand, destroying relative distances is the intention of fine-grained randomisation, on the other hand, it makes re-arranging code very difficult. Hence, a symbolic representation is created that abstracts any relative or absolute addresses. As a basis, the unmodified (assumed to correct) program is taken and all references (relative or absolute) are transformed into symbolic labels from which the reference graph can later be created.

```
loc_8048000: e8 00 00 00 00     call   loc_8048005
loc_8048005: 5b                 pop    %ebx
loc_8048006: 81 c3 27 70 03 00  add    $(GOT+0x27-loc_8048005),%ebx
loc_804800c: 8b 03              mov    (%ebx),%eax
```

Listing 6.5: Intermediate representation using symbols

The above form of code is similar enough to x86 to be efficient in terms of disassembling and re-assembling, yet it is address-agnostic and can be re-arranged in memory.

The most challenging part of this transformation into symbolic intermediate representation is to identify **all** references in the code. If a reference is overlooked, the semantics of the program would change, presumably leading to a crash. Fortunately, compiler-generated patterns (such as the above *position-independent code*) can be detected reliably. However, not all cases of memory references can be detected with 100% accuracy. Ambiguous cases can be, for instance, the addition of two seemingly random numbers whose sum is used as a pointer. To reliably detect those corner cases, the binary rewriter additionally uses relocation information as auxiliary information to guide the disassembly processes even in those cases. Especially indirect jumps are hard to detect, even though not impossible. The typical use of an indirect jump is a compiler optimisation of a `switch/case` statement. Often, compilers generate a single indirect jump that uses the argument of `switch`, instead of several `if-then-else` blocks to handle all cases.

The following example code uses a `switch` statement and illustrates how the GCC compiler uses indirect jumps as optimisations. The attached relocation information is then useful to identify the references in the indirect jump.

```
int res = 0;
switch(i)
{
  case 1: func1(); break;
  case 2: func2(); break;
  default: funcdefault();
}
```

Listing 6.6: Indirect jump generated from C source

Without any optimisations (i.e. `-O0` GCC switch), the generated code contains two `if`s and an `else` case to simulate the three cases in the switch:

```
8048ec7:       83 f8 01                cmp    $0x1,%eax
8048eca:       74 0c                   je     8048ed8
8048ecc:       83 f8 02                cmp    $0x2,%eax
8048ecf:       75 0e                   jne    8048edf
8048ed1:       e8 cd ff ff ff          call   8048ea3 <func2>
8048ed6:       eb 0c                   jmp    8048ee4
8048ed8:       e8 b7 ff ff ff          call   8048e94 <func1>
8048edd:       eb 05                   jmp    8048ee4
8048edf:       e8 ce ff ff ff          call   8048eb2 <funcdefault>
8048ee4:       c3                      ret
```

Listing 6.7: Generated non-optimised x86 assembler

When optimising with `-O2`

```
8048eeb:    ff 24 85 28 ed 0b 08    jmp    *0x80bed28(,%eax,4)
8048ef2:    e8 9d ff ff ff          call   8048e94 <func1>
8048ef7:    eb 1a                   jmp    8048f05
8048ef9:    e8 9e ff ff ff          call   8048ea3 <func2>
8048efe:    eb 0c                   jmp    8048f05
8048f00:    e8 bd ff ff ff          call   8048ed0 <funcdefault>
8048f05:    c3                      ret
...
80bed2c: 8048ef2
80bed30: 8048ef9
80bed34: 8048f00
```

Listing 6.8: Generated optimised x86 assembler

the GCC compiler has inserted an indirect jump into a jump table (`jmp *0x80bed28(,%eax,4)`), which redirects to the landing pads right after that instruction. The jump table at `0x80bed2c` simply contains word-sized addresses pointing to the landing pads at `0x8048ef2` and so on. The relocation information provided inside the executable file is helpful for determining the size of the jump table because each entry in the jump table is annotated with relocation information.

```
Offset     Info     Type       Dest. Section     Dest. Addr.
080bed2c  00000601 R_386_32   .text             8048ef2
080bed30  00000601 R_386_32   .text             8048ef9
080bed34  00000601 R_386_32   .text             8048f00
```

Listing 6.9: Relocation Information stored inside the compiled executable file

Even though this implementation relies on relocation information to accurately determine all memory references, literature suggests that disassembly is possible without any relocation information [P81]. Code that is typically loaded at different addresses, such as Linux and Windows kernel modules/device drivers or Windows DLLs typically include such relocation information. However, in the default configuration, it might not cover all memory references since the linker assumes the code and data sections will stay exactly as they were at compile-time. Because the rewriter is deliberately tearing apart the code, more relocation information might be necessary to reliably discover all memory references. To help this, all code was compiled using `-Wl,-emit-relocs` compiler/linker options. This option includes relocation information to *every* instruction that reads, writes or jumps to memory.

The rewriter details are described in section 6.4.

6.2.2 Phase (b) – Randomisation

The intermediate representation can be used to efficiently insert instructions into the code segment. Inserting instructions with no side effect changes distances between instructions as they are then further apart. The set of instructions that can be inserted without any side effect are basically synonyms of `nop` instructions:

```
0:  90                   nop
1:  8d 00                lea    (%eax),%eax
3:  66 8d 36             lea    (%esi),%si
6:  67 66 8d 34          lea    (%si),%si
a:  0a 05 00 00 00 00    or     0x0,%al
10: 8d b6 00 00 00 00    lea    0x0(%esi),%esi
```

Listing 6.10: Instructions with no Side-Effects (nops)

Then, the code is broken apart into chunks with the intend to change their order. To achieve the desired randomisation, the code is first logically divided into chunks by drawing boundaries between two subsequent instructions. Before those chunks can be re-ordered in memory, their implicit control-flow needs to be transformed into explicit control flow. Otherwise, ripping apart two subsequent code chunks results in a semantically invalid program. To avoid this, the implicit control flow, i.e. the fall-through of one instruction to the next, is made explicit by inserting a connection between the boundary of two chunks. Such a connection is an unconditional `jump` that connects the last instruction of one chunk to the first instruction of the consecutive chunk. When either of the chunks is moved away, the connection remains intact. This is due to the use of symbols rather than real addresses. As the symbols do not change, a code chunk can be moved in memory without impairing the semantics of the program. In Figure 6.2, the monolithic example program program consisting of basic blocks A through H has been split in chunks of different size whose order was changed. The original program's semantics do still hold however, as the control flow between the chunks has been adjusted to accommodate for their changed position in memory.

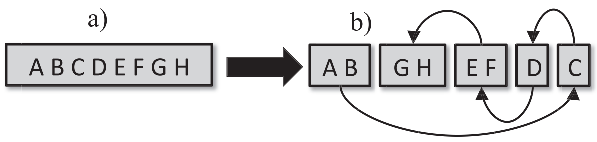


Figure 6.2: Splitting of code into several interconnected pieces.

Another challenge is the fact that splitting code in too many pieces imposes a lot of pressure on the processor's instruction cache (as can be seen in the performance evaluation in section 6.7) since the locality of code has been destroyed. To counteract this, the position and the number of cuts are restrained:

Positions: Whenever possible, existing control transfer instructions (e.g. `jump`, `call`) in the original program are leveraged as a splitting boundary. This has two advantages: when the two chunks surrounding a split are moved away, no additional control flow instruction needs to be inserted because the existing one can be used to transfer control to the new position. Second, the run-time will be almost identical because no instruction has been introduced by only the control flow target will change. An analysis of the `coreutils` and `SPEC` benchmark binaries has shown that on average 15.5% of all instructions are control flow instructions that can be used to split the code into chunks. This means that for a very small 1000-instruction-binary already 156 code pieces are available to be shuffled. This entropy of $156!$ is already higher than the ultimate ASLR solution that would provide an entropy of 2^{48} on a 64-bit system.

Number of Cuts: The possible number of permutations of n chunks is $n!$. Since this number gets large very quickly for rather small n, it is possible to limit the number of chunks n while still achieving a large entropy. The maximum entropy for a 32-bit user mode process is 31 bits: 2 GB kernel space, 2 GB user space. Therefore, a reasonable limit for $n!$ is 2^{31}, even though the usable user mode address space is actually less than 31 bits due to technical limitations. The entropy of 13 permutations ($13! = 6,227,020,800$) is already larger than 2^{31}. Therefore, it actually makes no sense to split more than 12 times[1]. This yields $log_2 13! \approx 32.54$ bits of entropy for a 32-bit system. Analogously, for a 64-bit system with 48 bit canonical address user space, 17 chunks yield sufficient entropy.

6.2.3 | Phase (c) – Re-Assembly

The last phase (c) takes care of reflecting all changes done on the intermediate representation in the output binary code. This code is emitted into the address space of the loaded process, since rewriting is done in-place in the newly created process. In chapter 7, the rewriter is used to create new executable files instead.

The code reassembly is achieved by stitching together binary strings of bytecode that represent instructions. Instructions that do not reference any memory do not change their binary representation when moved to a different position. Hence, they can be copied and pasted from the original input. For the instructions that do change because they encode an address, a similar technique to relocation information is used. The original bytecode of the instruction is copied and pasted to the correct destination address covering the entire length of the instruction. However, the bits of the bytecode of the instruction that encode either a relative or absolute address are patched using the reference graph that was created in phase (a). The relocation information types (absolute, relative, multiple of 2, and so forth) are covered by the architecture-specific relocations anyway so that all possible ways of encoding an address in an instruction are covered. This way, re-assembly is fast.

All three phases (a) through (c) with their internal steps are depicted as an overall process in Figure 6.3: The application gets disassembled and the reference graph is created. All nodes in the graph ($A, B, \ldots, E$ in Figure 6.3) lie flat in the program memory until they are chunked, additional instructions are inserted and the chunks are then permuted. This results in a reference graph

[1] 12 times splitting makes 13 pieces

with changed order of its nodes as depicted in the bottom of Figure 6.3. Some nodes have been split ($A \rightarrow \{A1, A2\}$), others needed to be changed to insert new control flow instructions (B', C', D', E').

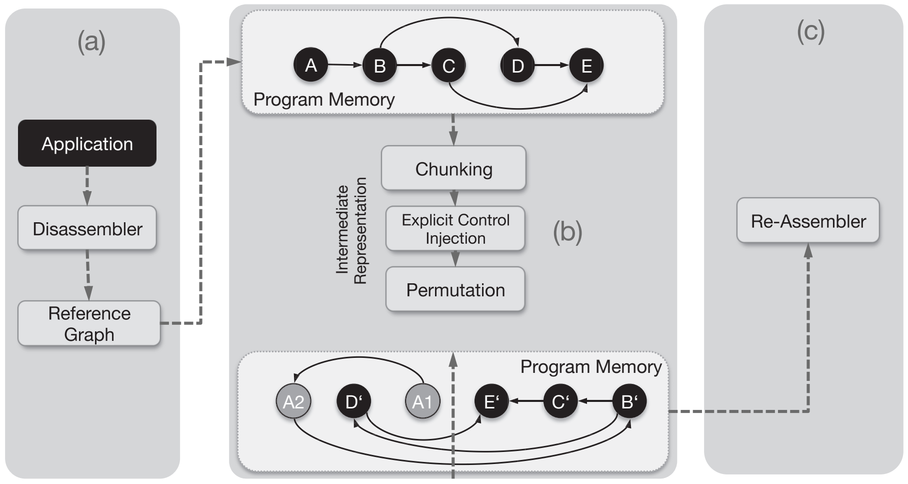


Figure 6.3: Re-Organisation of control flow in memory.

6.3 | Challenges

Function Returns. Instructions that unconditionally transfer control to another point in the program (e.g. `jmp 0x1234`) can simply be rewritten to their new address in memory to which the original target has been moved. As described above, this is why they are used as a *'natural'* split boundary for chunks. However, when code is split after an instruction with implicit fall-through control flow, the insertion of an explicit control flow transfer is required. All non-control-flow instructions have such implicit control flow, i.e. the execution continues at the subsequent instruction. If code is split after such an instruction then an explicit `jump` needs to be inserted when code following the split is moved away.

Some control flow instructions feature both: an explicit control flow transfer to another position in memory and an implicit fall-through. Such examples are conditional branches. They might branch to a different position in the code or execution might simply fall through in case the branch was *not* taken. `call` instructions are similar in that respect as they transfer control to another position in memory but control flow will continue at the next instruction after returning from the call. In either case, the subsequent instruction to which the control flow would implicitly fall through can potentially be moved away because it is now being part of different code piece that has been shuffled away.

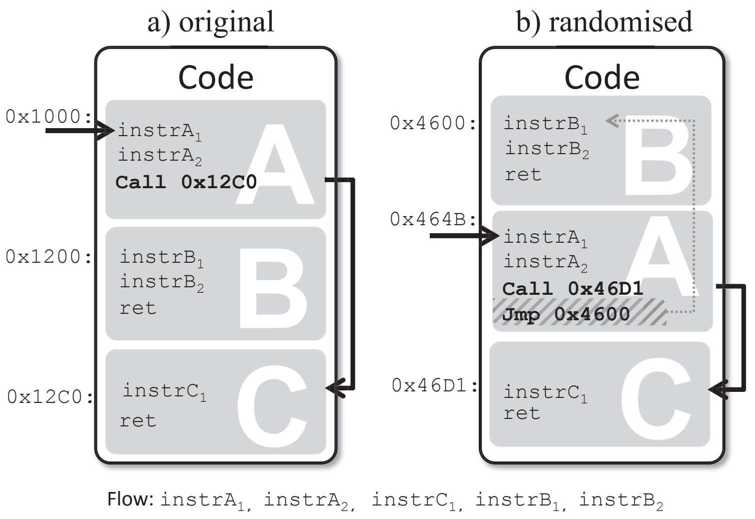


Figure 6.4: Implicit fall-through control flow needs an explicit control flow in order to sustain the original sequence of instructions.

Figure 6.4 shows an implicit control flow after a `call` instruction. In the original code (a), execution returns from the call to `0x12C0` and will continue after the `call` instruction by falling through to code piece B (`instrB`$_1$`, instrB`$_2$) located at address `0x1200`. If no explicit control flow is inserted in the randomised version, it would also return to the position right after the `call` where it left off. However, control flow would fall through to code piece C – which is wrong. This is why the unconditional jump has been inserted in the randomised (b) version in order to connect the control flow with the original code piece that now resides at `0x4600`.

Position-Independent Code (PIC). The implementation of ASLR required executables and libraries to be able to work when loaded at different addresses. The encoding of fixed, absolute addresses like done in traditional executables contradicts ASLR. To this end, position-independent code was used. Instead of absolute addresses, code and data are referenced as an offset to the current execution position. 32 bit and 64 bit x86 code both reference code in a relative fashion anyway. The only exception are indirect calls (e.g. `call *%eax`) that are usually used for function pointers or C++ vtables. In 64 bit mode, the current address of execution, the instruction pointer RIP, can directly be accessed and be used for addressing. Data can then be referred to relative to RIP like so:

```
b3f9: 48 8b 0d e0 da 25 00    mov    0x25dae0(%rip),%rcx
b400: 0f b6 01                movzbl (%rcx),%eax
```

Listing 6.11: Position-Independent Code (PIC) taken from ssh executable (64 bit)

In contrast, 32 bit x86 code cannot access EIP directly and has to go through a little workaround to find out its current address of execution.

```
a2c2: 53                    push   %ebx
a2c3: e8 a8 fa ff ff        call   9d70
a2c8: 81 c3 30 07 0a 00     add    $0xa0730,%ebx
a2cd: 8b 83 78 07 00 00     mov    0x778(%ebx),%eax
...
9d70: 8b 1c 24              mov    (%esp),%ebx
9d73: c3                    ret
```

Listing 6.12: Position-Independent Code (PIC) taken from ssh executable (32 bit)

In the example above, the code gets its own address by moving the return pointer into %ebx. Thus, after calling 9d70, the register %ebx contains a2c8 – the current address of execution. Data is then accessed the same way as in 64 bit mode – relative to this address.

For function calls into another library or global variables of shared libraries, it gets more complex. Linux uses a *Global Offset Table* (GOT) to solve the problem. This adds an additional level of indirection in a separate piece of memory that is filled by the OS linker at start-up. The address of the GOT itself can in turn be extracted from the executable's symbol table. The program or library then always indexes this GOT instead of calling the function or accessing the global variable directly. Figure 6.5 a) shows a typical usage of the GOT: the address of the GOT is calculated in relation to the current address of execution, then an index inside the GOT is dereferenced to get the actual address.

In both cases, relative access to data and relative access to the GOT, the offset from the current address of execution to the data or GOT in question needs to be adjusted when randomising code chunks. Otherwise, the relative addresses would not match the intended targets anymore. Because the relative distances are not part of relocation information, access in that relative fashion can easily be detected. To overcome this, the rewriter uses a code pattern detection technique to detect PIC.

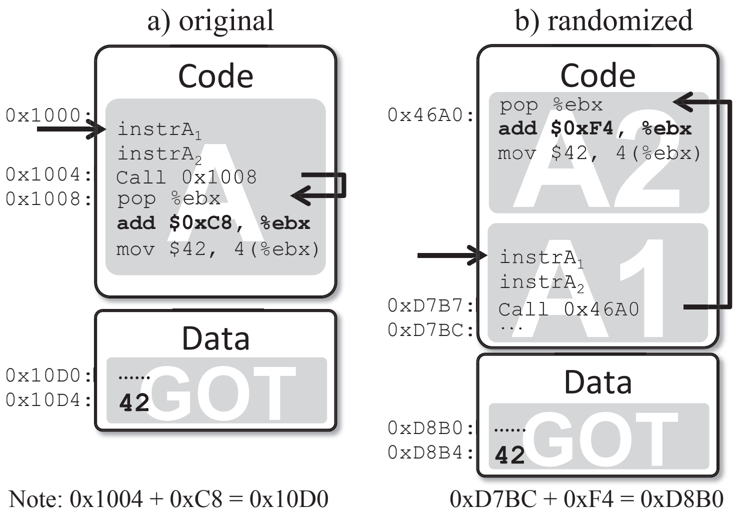


Figure 6.5: Position-independent code (PIC) needs to be adjusted as it assumes relative relations inside the code stay intact.

Figure 6.5 depicts a common write operation to data residing in the GOT. In this example, the compiler-generated code to determine the current address uses a `call` instruction that targets the next instruction (`0x1004`). The next instruction then pops `%ebx`, which contains the return address of the aforementioned `call` off the stack. The `%ebx` register then holds the absolute address in memory of where the call should return. In other words, the `call` and `pop %ebx` move the current instruction pointer into `%ebx`. In 64 bit mode the instruction pointer `%rip` can be directly accessed and moved into `%rbx`, however in 32 bit mode there is no such instruction as `mov %eip, %ebx`. Even though the `call` instruction has no corresponding `ret` instruction, the `pop` mimics the removal of the return address from the stack.

The bold `add` instruction in Figure 6.5 is the instruction in question which encodes the offset into the GOT relative to `%eip`. In the randomised version in Figure 6.5 b), the `add` instruction is inserted into the reference graph in order to benefit from the automatic reference adjustment of the rewriter. The result is that the encoded offset is adjusted (`0xC8` to `0xF4`) such that it still references the GOT. The detection of PIC code works as follows: First, call targets are analysed for immediate `pop` instructions. Under normal function call conventions, this would not make any sense as the callee cannot make any assumptions about the previous items on the stack. The detected `pop` instruction reveals into which register the return address is loaded. This register is then tracked in arithmetic operations to detect which offset is added to or subtracted from it. When the original offset plus its calculated own position equals the GOT, a valid PIC case has been found and the offset can be extracted from the `add` or `sub` instruction. During testing, no cases were found in which the extracted offset + `%eip` did *not* point to the GOT.

Interestingly, the semantics of the `pop %ebx` instruction have slightly changed in the randomised version. Before, the instruction popped its *own absolute address* off the stack as if it were copying `%eip` into `%ebx`. However, as it resides at a different position later, this is no longer the case. Since the 32 bit PIC uses the diversion of the stack, this does not make any difference. However, a future 64 bit version of the rewriter can no longer simply `mov %rip, %rbx` because it resides at a different address.

C++ Exceptions. Handling C++ exceptions is not easy because the involved stack unwind mechanism that tries to find the `catch` block at run-time relies on pre-calculated addresses.

When the *GCC* compiler compiles C++ exceptions, a so-called *unwind frame* is created for every function in a special section called `.eh_frame`. It contains information about how to restore the stack and registers when returning to the point inside the code that actually `catch`es the exception. To find the correct unwind frame inside `.eh_frame`, the list of all frames is searched for the `%eip`. This is possible because all potential addresses where an exception might be thrown are known at compile-time. Hence, the offending `%eip` must be in the list. This list is stored in ascending order so it can be efficiently searched using binary search. Each entry provides information about which function catches that particular exception and how to unwind the stack to continue execution at that function. The stack unwind information is stored as a stack machine bytecode that is interpreted at run-time when the exception is thrown. This *GCC*-specific machine language incorporates relative addresses in order to calculate the beginning of a function given only `%eip`. The following is an example of a typical unwind frame that has been decoded as GCC machine language:

```
0000dea8 00000014 0000deac FDE cie=00000000 pc=080bd620..080bd64b
  DW_CFA_advance_loc: 3 to 080bd623
  DW_CFA_def_cfa_offset: 32
  DW_CFA_advance_loc: 39 to 080bd64a
  DW_CFA_def_cfa_offset: 4
  DW_CFA_nop
```

Listing 6.13: Unwind Frame retrieved using `readelf -u`

However, after randomisation this information is no longer valid. Therefore, the `.eh_frame` is rewritten according to the changed layout.

Intermixed Code and Data. Compilers often optimize code by aligning functions in memory so that they start at the beginning of a cache line. The inevitable gap before aligned functions is sometimes filled with data in the middle of code. A typical fall-through disassembly[2] is thus not possible, as it would interpret alignment zeros (garbage) or intentional data as instructions. Unfortunately, this could lead to cases where the disassembly is out of step with the actual instructions.

To prevent this, code must not be disassembled in a linear fashion but should rather realign itself with respect to indicators for the start of an instruction. Such indicators are targets of control flow, function calls, conditional and unconditional branches. This information is used to realign the disassembly process based on discovered control flow targets. Sanity checks ensure all references from and to code and data stay in their respective segment and target only the beginning of an instruction. Still, even self-aligning disassembly is provably complete as corner cases might exist. However, during testing, no corner cases were detected, i.e. no illegal opcodes and no discovered references into the middle of an instruction or outside of code or data were discovered. Additionally, the rewritten files showed original behaviour and did not crash. This is a good indicator that the disassembled code actually makes sense.

Interesting Compiler Quirks. The goal of the binary rewriter is to be compatible with all, or at least most, of the programs it might ever encounter. This means the disassembly and rewriting process must be able to cope with weird code that compilers might generate. To this end, the binary rewriter was designed to perform a multitude of sanity checks on the disassembled code in order to self-detect if recovered code and code-flows seem implausible. Some of those plausibility warnings were actually compiler optimisations that the

[2]The process of disassembling instructions from the beginning of the code in a linear fashion to its end.

rewriter stumbled upon because they are seemingly irrational at first glance. The following lists a few interesting discoveries about gcc's optimisation.

`call 0x00000000`: One of those examples is an instruction found in every program compiled with the GCC C and C++ compilers: `call 0x00000000`. It occurs multiple times in each program and only a few of those calls are placeholders for relocation that eventually get replaced by meaningful addresses. The majority stays zero and should any of them ever be executed would definitely result in the process being killed by a `SIGSEGV`. All calls were located inside the `glibc` library that is linked into every executable. Their pattern is always similar: Each call to zero is preceded by a conditional jump, like so:

```
...
80481bd:    85 c0                   test   %eax,%eax
80481bf:    74 05                   je     80481c6 <_init+0x1e>
80481c1:    e8 3a 7e fb f7          call   0
80481c6:    83 c4 08                add    $0x8,%esp
...
```

Listing 6.14: Call to zero found in every program

Since the default installations of GCC libraries do not have debugging information attached, it is not possible to pinpoint the source of those peculiar calls. After re-compiling GCC from its source with debugging information enabled, it was possible to pin-point the exact source code files that are responsible for those deliberate null-pointers. Almost all occurrences are function pointers that are being called inside a loop. The following is taken from one occurrence, namely the call to C++ destructors linked into every[3] C and C++ program.

```
func_ptr f;
while (dtor_idx < max_idx)
  {
    f = __DTOR_LIST__[++dtor_idx];
    f ();
  }
```

Listing 6.15: Culprit found in GCC's crtstuff.c start-up code

The optimisation level of the GCC compiler however un-rolls that loop and splits it into cases where `f` equals zero and `f` is something else than zero. Since `__DTOR_LIST[...]` is filled at compile time with pointers to destructors, it will never actually be zero and this code will never be called.

[3]GCC also links C++ constructor/destructor code by default for compatibility to mixed C/C++ code.

The rewriter, however, needs to be made aware of those peculiar calls to zero. Otherwise, the reference graph cannot work because it would try to connect this call instruction to a non-existent instruction located at address zero. However, leaving this instruction untouched would lead to incorrect code, as call targets are encoded as relative position in both, x86 and ARM architectures. Consequently, simply moving `call 0` to a different address would result in a call to an arbitrary address. Therefore, those `call 0` instructions are treated separately and will be corrected to point to zero again in the final executable. Since they are actually never called (i.e. unreachable code) it would not matter to treat them specially, but for the sake of semantical correctness, those calls shall still point to zero in the final code.

`nop, nop, nop, ret`: Another peculiar case worth mentioning is that some code snippets were detected as implausible, because several `nop` instructions occur before the return (`ret`) of a function.

```
08048e50 <smallfunc>:
8048e50: 8b 44 24 04           mov    0x4(%esp),%eax
8048e54: 83 c0 01              add    $0x1,%eax
8048e57: a3 9c af 0e 08        mov    %eax,0x80eaf9c
8048e5c: 90                    nop
8048e5d: 90                    nop
8048e5e: c3                    ret
```

Listing 6.16: Short functions compiled for Atom CPUs exhibit several nops before returning

At first sight, this does not make any sense. At second sight though, this instruction pattern only occurs when compiling for the x86 architecture for Intel Atom CPUs. A closer look in the Intel Developer's Manual for Atom CPUs reveals that this particular CPU does not feature out-of-order execution, i.e. all instructions are always executed in the order they occur in the code. However, it features a pipelined architecture and because of that the return address, which is pushed on the stack by a `call` instruction is not yet available for `ret` if the executed function body consists of only a few instructions. The CPU has to stall execution until the address becomes available. So how come code for old x86 CPU that did not feature out-of-order execution as well (e.g. the 486) did not exhibit these `nop`s? The difference lies in the Atom's Hyperthreading feature, which allows the pseudo-parallel execution of instructions. While one core executes `nop` instructions, the other core can fully use the CPU's execution unit. In contrast, a `ret` instruction stalls the entire CPU, i.e. both execution units. In summary, executing a couple of `nop` instructions on one core allows at least the other core to continue while waiting for the return address to become available.

6.4 Rewriter Internals

The binary rewriter is at the core of the fine-grained randomisation idea. Consequently, its inner workings are explained in detail in this section. The rewriter understands the default Linux executable and shared library file format ELF [D8]. The rewriter itself works as a stand-alone program and as a library. In the first configuration, it is designed to work as static binary rewriter, i.e. it can emit an ELF binary output file. In the library configuration, it is designed to be injected into a process in order to perform in-place transformations, for example at process start-up. The code is entirely written in C++ and consists of $8,010$ source lines of code (SLOC).

The precise internal workflow of the binary rewriter is shown based on the following minimalistic running example:

```
0x8048000: e8 00 00 00 00      call   8048005
0x8048005: 5b                  pop    %ebx
```

Listing 6.17: Running Example

This example is a typical code snipped compiled into position-independent programs by the GCC compiler. This routine returns the memory address of the instruction that called it and is typically called `i686.get_pc_thunk.bx` in the symbol table. The first instruction calls the next instruction thereby pushing the return address on the stack, which is then popped off the stack by the second instruction. This second instruction (`pop %ebx`) is independent of its load address in memory and therefore does not need to be transformed into the intermediate representation.

In the following, the main steps (see Figure 6.6) involved in the rewriting process are explained:

1. *Loading* the executable.
2. *Disassembling* the bytecode on-the-fly.
3. Building a *Reference Graph* of the executable.
4. Applying *Transformation*.
5. Writing the executable back to memory (*Fixation*) so that it can start executing.

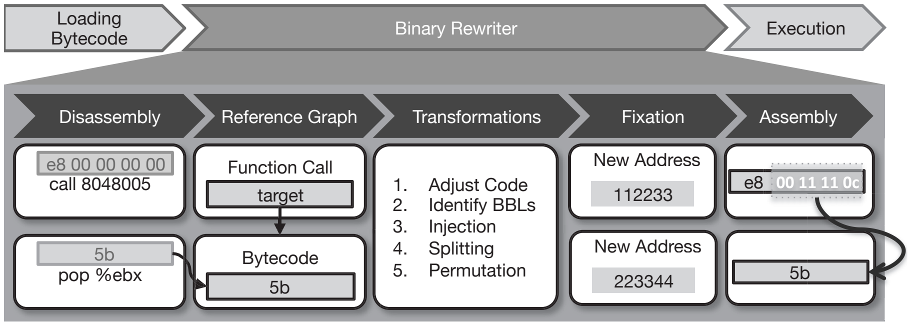


Figure 6.6: Processing Steps of the Rewriter

6.4.1 Executable Loading

The default Linux file format for executable files and shared libraries is the Executable and Linking Format (ELF [D8]). ELF executables and shared libraries are composed of different loadable segments. These segments are usually divided into chunks of code with different memory protections, e.g. read-only, writable or executable. Inside those loadable segments, ELF files feature sub-segments, so-called *sections*. These sections are not technically necessary to create a running program, but they help organise the executable. Typically inside the executable segment, code is located in the `.text` section. Analogously, writable data resides in the section `.data`, read-only data inside `.rodata`, uninitialised data inside `.bss` and exception handling information is stored in `.eh_frame`. Typically, `.rodata`, `.eh_frame` reside in the read-only loadable segment. However, some linkers put those sections in the executable segment together with the ELF header[4].

In order to rewrite the code stored in ELF files, the rewriter can either read them from disk or it can be attached to a process before it is started. The first way treats the ELF executable as an ordinary file, the second method uses the `LD_PRELOAD` environment variable, which forces foreign shared libraries to be loaded into an address space before the program starts. `LD_PRELOAD`was originally intended to fix bugs of legacy software by allowing the override the symbol resolution of shared libraries. It can also be exploited to force arbitrary code to be run in any process. Therefore, it is a viable solution to forcefully inject the rewriter as a shared library (`librewrite.so`) into a process that shall be protected by fine-grained randomisation. This procedure makes `librewrite.so`

[4]This clearly is a security risk as it unnecessarily increases the number of gadgets due to unintended instructions found in read-only data such as the ELF header.

part of the address space as if it were referenced as a shared libraries by the to-be-protected program. However, no function calls to `librewrite.so` exist, as COTS programs are unaware of the rewriter. Consequently, its code would never execute. Furthermore, the rewriter is designed to run *before* any code starts executing in order to have a change to randomise the address space. To achieve this desired goal, all shared libraries, including `librewrite.so`, can make use of a special loadable section called `.init`. All code residing in `.init` is guaranteed to be executed before the main entry point of the executable. The rewriter uses this technique to ensure rewriter code is executed before any of the main program code. Of course, other shared libraries might also have an `.init` section that executes early code. However, the operating system cannot guarantee the order of execution for several `.init` sections.

When execution starts in the `.init` section of `librewrite.so`, it is mostly identical to the static binary rewriting version that can operate on ELF files stored on disk. The only difference is that code and data sections are either loaded from disk or accessed in-place in memory, respectively.

Because the rewriter needs relocation information, it is read from the main executable file and its shared libraries. However, relocation information provides no information about *local* data/code references within a segment. As already briefly explained in section 6.2, this information is obtained by static analysis when transforming x86 instructions into the intermediate representation.

6.4.2 | Instruction Representation

In order to work fast and efficiently with instructions, a representation was chosen that is

1. independent of the current address of execution
2. allows direct manipulation of immediate instruction values such as memory addresses and memory offsets.

To address the above requirements, each native instruction is represented as a C++ class called `Instruction`. This class features access to the actual bytecode of an instruction by means of `char*` pointers that point to the actual position in the address space. Only an additional field is required, the `length` of an instruction, as that may vary for x86. Derived classes, such as `CallInstruction`, use a pointer of type `Instruction*` to reference other instructions. This way, the resulting set of `Instruction`s that represents the original code is free of any addresses. If code needs to be written back to memory, all `Instruction` classes can directly set the call target by manipulating specific bytecode bytes.

This design decision has the advantage that unimportant instructions do not have to be resolved to their precise type but can be stored as the generic `DontCareInstruction`, which merely stores their bytecode representation. This applies to a majority of instructions as they do not reference memory addresses and hence their byte representation is not affected by the randomisation.

Instruction Format. The rewriter must understand the native bytecode of x86 in order to know each instruction's length. Otherwise the beginning of the next instruction could not be found. Both ARM and x86 instructions are composed of a mandatory opcode, which defines the type of instruction (e.g. `push`, `call` etc.) and optionally other encoded information. Unfortunately there is no straight-forward way of decoding an x86 instruction. However, there is a generic concept depicted in Figure 6.7.

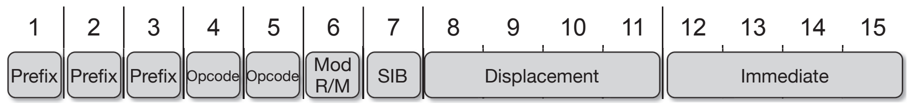


Figure 6.7: Layout of the longest x86 instruction possible (15 bytes)

The Main Opcode (byte 4 in Figure 6.7) is mandatory and is usually the first byte so that the processor knows how many bytes can follow. However, it can be preceded by up to three prefixes, each represented by a byte.

Prefixes have a unique value, that is not identical to any opcode so that they can be identified. These prefixes usually affect the word operation width (16 or 32 bits), can make an instruction atomic (`LOCK` between multi processors) or override a segment selector (`%gs:0x4`). In some cases, an additional opcode byte (byte 5 in Figure 6.7) is needed to accommodate the need for more than 256 opcodes. Some opcodes need an additional `ModR/M` byte. Depending on the opcode, this byte is either used for `Mod`-ifiers, it names a register as operand or it indicates the use of `Mem`-ory.

Mod is used to further diversify or **mod**ify an opcode. It has a length of two bits. It is therefore similar to two more opcode bits.

R/M is a three-bit value, which is stored two times (6 bits total) in the `ModR/M` byte. Each value determines whether the source and/or destination is a **R**egister or **M**emory. In the latter case, the `SIB` (Scale-Index-Base) byte can further specify the kind of memory access. The absolute address of memory is specified in the `Displacement` section of the opcode.

The SIB byte stands for *Scale-Index-Base* and describes how the actual memory address is calculated manner. As a mnemonic, it is used as follows: `addl (%ebx,%ecx,0x4),%eax`. The mathematical result of this instruction is $eax \leftarrow eax + ebx + 4 \cdot ecx4$. *Scale* refers to a multiplier to use (1, 2, 4 or 8), *Index* names the register to use and *Base* is used as an offset. An immediate value might even be encoded as an additional offset (s.u.).

Displacement holds either an 8-bit, 16-bit or 32-bit memory address, hence is either 1, 2 or 4 bytes long.

The Immediate part may contain any 8-, 16- or 32-bit operand, e.g. `addl eax, $1234`. The number is then directly encoded in the instruction itself. The immediate value can also be used in addition to an SIB or Displacement. For example `subl 0x10(%ebx,%ecx,0x4),%eax` (note the `0x10` in contrast to the example above). This results in $eax \leftarrow eax + ebx + 4 \cdot ecx + 0x10$.

Unfortunately, x86 has no bit in the opcode that indicates whether SIB or an immediate value are encoded. To overcome this, the rewriter uses a look-up table that it indexed by the opcode and returns whether SIB or immediate are used. The decoding of SIB and ModR/M then reveals how many bytes of Displacement or Immediate follow. This boils down the processes of disassembly to a mere look-up and decoding of SIB/ModR/M. This feature increase performance in two ways:

1. No sophisticated disassembling process is required that actually resolves the mnemonic and all its operands,
2. Opcodes that do not or cannot encode any memory references are immediately set to `DontCareInstructions` and all that remains is their mere bytecode representation.

Experiments have shown that this *'fast disassembly'* is $\approx 10\times$ faster than using `objdump`'s internal `libdisasm`.

6.4.3 | Reference Graph

All instructions that reference data or code are candidates, which potentially need to be rewritten later and act as input to build the *Reference Graph* that provides information as to which other part of code or data is referred to by an instruction. This is enabled by keeping an additional layer of references above the original instructions. Those references resemble a directed graph with edges pointing from `Instruction` objects to other `Instructions`. If data is referenced, it works in the same way as data is stored in a single, huge

DontCareInstruction, which simply contains all the data bytes. Instructions that manipulate the stack are not part of the graph as the stack is always addressed in relation to %esp. Hence, the stack (1) cannot be considered by the rewriter, which is static analysis, and (2) it does not need to be considered as the original semantics are kept in tact and hence the stack layout during run-time is not affected. The internal representation and especially the reference graph is generic enough as to work with several processor architectures. Currently, it supports a majority of the 32 bit x86 ISA and a subset of ARM.

Following the running example from above (call 0x8048005), the underlying bytecode is e8 00 00 00 00. In x86, calls and jumps are always expressed relative to EIP. When executing an instruction, the EIP internally already points to the next instruction. Because the call is 5 bytes long, EIP points to 0x8048005. Consequently, zero needs to be added to point to 0x8048005. This is exactly what the bytecode does: e8 is the opcode for call and 00 00 00 00 is 32 bit relative address to call.

In the Reference Graph representation, the target (Instruction*) is therefore connected to the instruction decoded at 0x8048005, which is of type DontCareInstruction and simply has the bytecode 5b.

In order to be able to reassemble the call instruction in the last step of the rewriter, the Instruction object must also save where inside the instruction the target is encoded. In case of the call instruction, this is at offset 1 (skipping the opcode byte) and is a 32 bit address stored relative to the instruction's own address with an offset of 5 (the instruction's length). In C/C++ terminology, this technique is a union as arbitrary bytes of memory inside an instruction's bytecode are interpreted as 8-, 16- or 32-bit (un-)signed integer numbers. How to encode and decode those unions is stored as so-called *FastDecode* information for each Instruction object (see Table 6.1). Particularly, *FastDecode* information includes whether it is signed or unsigned, a bit mask, a bit shift and a summand. This coding is generic enough to enable the rewriter to later write back addresses to an instruction without understanding the instruction itself, and is faster than assembling an instruction from scratch. When considering the running example, the attached *FastDecode* information would store the values depicted in Table 6.1 for the ARM beq instruction.

The *Bit Mask* and *Bit Shift* fields are always the same for x86 but are required for ARM. ARM encodes code references as 24 bit numbers, hence uses a *Bit Mask* of 0x00ffffff. And because ARM instructions are always 4 bytes in length, the relative distances would be multiples of 4. Therefore, ARM encodes the relative distance to the code target shifted right by 2 bits.

Info	Value
Opcode	`e8`
Signedness	Signed
Bit Mask	`0xffffffff`
Bit Shift	0
Summand	5

Table 6.1: *FastDecode* information codes how to write back a part of an instruction in an assembler-agnostic way

6.4.4 Transformations

After the entire code has been read and is represented by `Instruction` objects, the desired fine-grained randomisation can take place on the Reference Graph without caring about memory addresses.

The rewriter supports grouping instructions to *Instruction Sequences* that are later used to move code pieces together in memory. ELF sections (`.init, .fini, .text`) are treated as 'natural' *Instruction Sequences*. Every sequence can be (recursively) separated in two sequences. This is done by inserting a `jump` as the last instruction in the preceding sequence. This is necessary to keep the program's semantics as alluded to in section 6.2. Each *Instruction Sequence* also supports the insertion of new, arbitrary instructions at any position inside the *Instruction Sequence*. Inserting instructions in the *Instruction Sequence* is used as a building block for the necessary explicit connection of severed code pieces using `jumps` but also for the insertion of the aforementioned `nop` instructions. The randomisation itself therefore is an *Instruction Sequences* permutation.

6.4.5 Fixation & Assembly

Lastly, the code pieces are transformed back to x86 or ARM instructions either on-the-fly inside the address space of the process (`librewrite.so`) or they are written to an ELF file on disk (static binary translation).

For this final step to work, all *Instruction Sequences* need a randomly chosen memory address. This is the address at which they are loaded in memory. This implicitly assigns each `Instruction` stored inside an *Instruction Sequence* a unique address. The code and data is then written back by writing the instruction's bytecode at the calculated instruction's address. Of course, the bytecode is adjusted with the help of the stored *FastDecode*. The necessary addresses that need to be encoded into an instruction are taken from assigned address of the instruction they reference.

Because an *Instruction Sequence* does not necessarily fit inside a memory page, a few bytes are potentially wasted. As the aligned instructions on ARM allow exactly 1024 instructions to take place in a memory page, up to 1023 instructions (or 4092 bytes) are wasted in the worst case. For x86, in the worst case 4095 bytes are wasted.

6.5 | Debugging

Debugging a randomised process is rather difficult in comparison to the original process for two reasons:

1. The debugger expects the instructions in memory to be in the same order as the program that is stored on disk.
2. Two subsequent executions of a program result in completely different memory layouts, which makes it harder for humans to understand memory-related faults in the program.

The first issue occurs because the debug symbols are rendered obsolete due to the re-shuffling in memory. To counteract this, the rewriter can emit debug symbols with up-to-date addresses to disk – even when loaded as a shared library. Currently, debugging information is written in the common DWARF [D9] file format which can be read e.g. by the `gdb` debugger. The debugger can then step through the code, inspect variables etc. as if the program were unmodified.

The second issue means that addresses change between every program run. Therefore, finding the new addresses for the same object is labor-intensive. We avoided this issue by adding a debug flag to `librewrite.so` that indicates that the same random seed should be used for every execution of the program, resulting in the same memory layout for each executing with that flag enabled. The consequence is that every run of a particular program ends up in exactly the same address space layout with every single instruction being at the exact same address across multiple runs. For all intends and purposes, this is against common sense of randomising a process in the first place. However, it greatly helps debugging a randomised process because all the variables reside at the very same address across different process runs or even reboots.

6.6 | Security Evaluation

The main goal of the rewriter is to change all relative offsets with the intend to nullify all assumptions about code positions when a single pointer is leaked. By dividing the code in several chunks and changing their order, all relative distance to each other have changed. Only distances within a single code chunk are kept as they were. Hence, the smaller the chunks, the more unlikely it is for an attacker to find exactly the necessary gadgets in the surroundings of the leaked pointer. Because the code is randomised at start-up, an attacker has no knowledge about the order in memory even though she might have access to the binary file itself. Since the attacker has no knowledge about the order of the chunks, she must assume that the needed gadget address is within the same chunk as the leak. If it were in another chunk, no assumptions about the distance to it can be made. This stresses that the size of a chunk corresponds to the security of this solution.

Let us assume a process' address space contains p memory pages of code, which present the c chunks into which the code has been broken. If the chunks are equal in size and each memory page is typically 4 kiB, then each chunk is of size $s = 2^{12} \cdot \frac{p}{c}$ bytes. Now the attacker's desired gadget may either reside in the chunk where a leak occurred:

$$Pr[Adv_{samechunk}] = \frac{1}{c}$$

or the attacker has to guess the address of a chunk, which is

$$Pr[Adv_{guess}] = 2^{-20}$$

because there are 2^{20} (2^{32-12}) possible memory pages in the address space where a particular page could be. Because the rewriter also inserts `NOP` instructions, the actual addresses inside a code chunk change. In fact, they are slightly increased as an inserted instruction can increase the memory address of following instructions but not decrease them. The farther the gadget address is from the beginning of the original code start, the higher it is moved up because of inserted `NOP`s. This means the insertion must be done with a random spacing in between the `NOP`s because otherwise their distances become predictable.

6.6.1 | Practical Security Evaluation

In order to test the effectiveness, two experiments were performed: (1) Calculating the *gadget elimination*. A comparison of found gadgets before and after the randomisation. (2) **Mitigation of an exploit** targeted to a vulnerable program.

Gadget Elimination. The 12 benchmark programs from the SPEC CPU2006 suite (see Table 6.2) were used to scan for ROP gadgets using the program *ROPgadget* [L16]. After applying the rewriter in static mode, ROPgadget was run again to check whether and how many gadgets have stayed at the original position.

Benchmark	ROP gadgets	Remaining gadgets
400.perlbench	67	0
401.bzip2	51	0
403.gcc	194	0
429.mcf	45	0
445.gobmk	105	0
456.hmmer	58	0
458.sjeng	57	0
462.libquantum	45	0
464.h264ref	79	0
471.omnetpp	168	0
473.astar	91	0
483.xalancbmk	460	0

Table 6.2: Overview of the SPEC CPU2006 integer benchmark suite.

Exploit. The following toy example was used for the purpose of illustrating how an exploit is defeated.

```
#include <stdio.h>
char welcome_msg[20];
float VERSION = 0.1f;

void main (int argc, char* argv[])
{
  char buf[8];

  strcpy(welcome_msg, "Welcome␣to␣");
  strcat(welcome_msg, argv[0]);
  strcat(welcome_msg, "␣version␣%f\n")
  printf(argv[0], VERSION);

  scanf("%s\n", buf);
}
```

Listing 6.18: Example Exploit Used to Test Fine-Grained Randomisation

The above code first prints `"Welcome to program version 1.1"`, where `program` is the actual name of the executable file. The executable file's name is gathered from the supplied `argv` parameter of the `main` function. Here lies the first defect: The program's name is used unescaped as an argument to `printf`. If an attacker renames the program to `'%p'`, it will actually print the address of the next argument, which is the global variable `VERSION`. In fact, it would print an address residing the `.data` section of the executable. If traditional ASLR is enabled, an attacker could learn the address of `VERSION`, thereby learning all addresses of all gadgets in the program.

The next vulnerability is the fact that the supplied user argument is written to the array of chars `buf`, which can only hold 7 characters plus the terminating zero character. This could easily be exploited by writing a saved `EBP` at the ninth position and a fake return address at the thirteenth position.

To mount the attack, ROPgadget was used to find gadgets in the executable. The address of those gadgets at run-time are simply offset by the address, which was leaked in line 11 as a printf-statement. This exploit has a success probability of 100% if the executable can be renamed to '`%p`'.

After applying the fine-grained randomisation of the rewriter, however, the program has been started the exploit failed. This is attributed to the fact that all relative distances to the gadgets founds earlier have been modified and do no longer work.

6.7 Performance Evaluation

In order to measure the design goal of performance, run-time and memory overheads of the binary rewriter and the entire fine-grained randomisation solution was evaluated. The run-time overhead was calculated using the industry standard *SPEC CPU2006* benchmark by comparing the performance of unmodified benchmark binaries to that of randomised binaries. Micro benchmarks were also used to measure the actual run-time of individual important control flow instructions.

Run-Time Overhead on Intel x86. All benchmarks were performed on an Intel Core i7-2600 CPU running at 3.4 GHz with 8 GB of DDR3-SDRAM. All benchmarks were compiled using gcc-4.5.3 and the *uClibc* C library. All measurements include a *complete* randomisation of the entire address space including the executable and all shared libraries. Two different randomisation configurations were compared:

- **All BBLs:** Every found control flow instruction is used to split code into pieces.
- **52 Bits:** The number of chunks is chosen such that the total entropy is greater than the theoretical entropy of a 64-bit address space (48 bits usable). The closest permutation with > 48 bits was 52 bits.

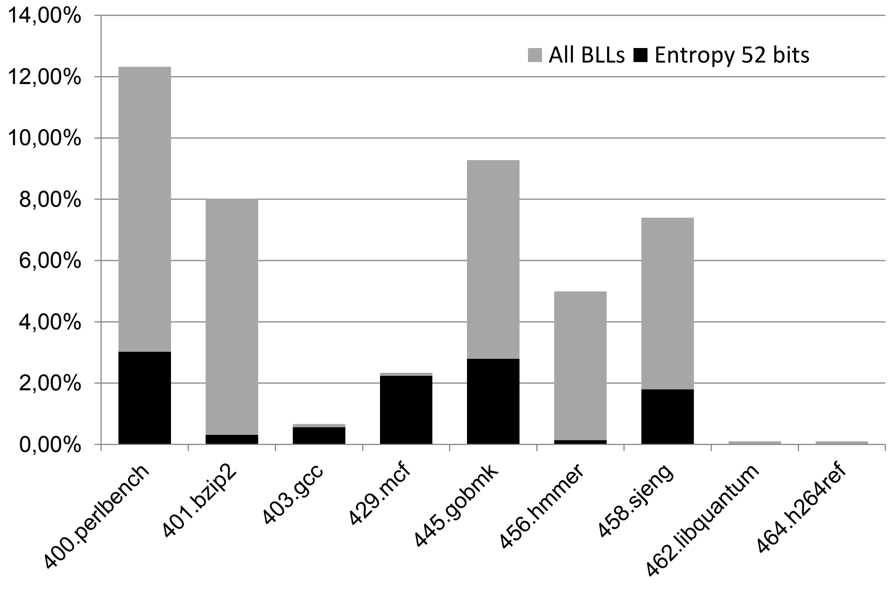


Figure 6.8: Run-time measurements with SPEC CPU2006

As can be seen from Figure 6.8, the entropy of 52 bits achieved a low average overhead of only 5%.

Explicit Jump Penalty. From a processor's perspective, fine-grained randomisation counteracts the locality an optimising compiler tries to achieve. Normally, compilers try to organise code such that so-called *hot* code paths are close together so that they fit in a single cache line of the processor. This improves performance because code does not have to be fetched from RAM as often when it is already inside the cache.

Before code chunks can be shuffled and moved around in memory, they need to be connected using explicit jumps, i.e. `jmp` instructions. To measure the impact of the explicit jumps alone, randomisation was turned off and only `jump` instructions were inserted. The number of total chunks did not matter, but only their size. The order of the chunks was left untouched in order not to disturb locality. The total run-time of all instructions of `bzip2` with a test input file was measured. The tests were repeated 1,000 times. Figure 6.9 shows the resulting overhead in dependence of the number of instructions between a forceful jump. The graph is to be read as follows: if a `jump` is inserted as every 11th instruction, there are 10 original ('productive') instructions in between – the resulting overhead is about 1.0%.

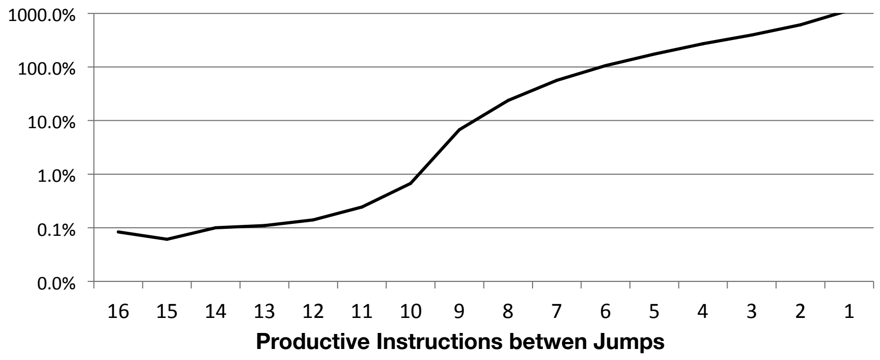


Figure 6.9: Microbenchmarks for Explicit Call Overhead

The insertion of `jmp` instructions effectively decreases the number of original program instructions that are executed per cache line. Additionally, the processor has to reach out to the main memory more often because the number of instructions increased for the same work unit.

6.7.1 Rewriting Time

Based on the SPEC CPU2006 benchmark programs, the time the rewriter requires to rewrite a single program was evaluated. The rewriting process is not exactly linear, but on average achieves between 250,000 and almost 1 million instructions per semcond. An overview of the timings of several programs is given in Table 6.3.

The number of instructions per benchmark reflect the total number of instructions from the executable file itself plus its dependent libraries.

6.7.2 Memory Overhead

During run-time, the `librewrite.so` is loaded once into the address space of a process. The code size of `librewrite.so` that is mapped into an address

Benchmark	Total # of Instructions	Rewriting Time (s)	Throughput Instr/sec
483.xalancbmk	1,111,779	4.02	276k
403.gcc	942,244	2.39	394k
471.omnetpp	238,978	0.656	364k
400.perlbench	322,084	0.684	471k
445.gobmk	226,661	0.724	313k
464.h264ref	170,942	0.284	602k
456.hmmer	54,582	0.096	569k
458.sjeng	40,438	0.088	460k
473.astar	32,502	0.056	580k
401.bzip2	28,087	0.040	702k
462.libquantum	15,788	0.016	987k
429.mcf	12,268	0.020	613k

Table 6.3: Rewriting time for benchmark executables. The total # of instructions include the all shared libraries.

space is ≈ 90 KiB. The overhead due to the inserted instruction varies. On average, it increases the code by $\approx 5\%$.

6.8 Fine-Grained Randomisation Conclusion

Fine-grained randomisation is an effective way of defeating return-oriented programming, especially in the light of leaked pointers. The chosen method of binary rewriting and in-place rewriting allows the mitigation to take place on binary programs for which no source code is available. Furthermore, in-place transformation is compatible with application signatures, which are commonplace on desktop and mobile operating systems. Even this prototypical implementation showed that a binary rewriting approach is a viable path to go because of its low performance overhead. A binary rewriting approach, however, has its limitation when it comes to corner cases of assembler code and is challenged by ambiguities between code and data. For a bullet-proof end user product, more time needs to be invested in coping with corner cases.

An open problem at the time of designing fine-grained randomisation was the fact that the popular copy-on-write (COW) feature depends on the fact that the address space is contiguous. COW is a Linux feature that saves memory by mapping the largely identical portions of processes in each address space while they are only backed by a single instance of physical memory. This is smart as most of the programs use identical libraries – hence the name *shared* *library*. Naturally, such a sharing is only possible if the content in each process is *identical*. With fine-grained randomisation, this is no longer the case. A problem that defeats the benefits of memory sharing, or COW in particular. This is addressed with the solution presented in the next chapter, which combines seemingly contradicting goals: sharing of different content. Hence the name *'Oxymoron'*.

7 Oxymoron

Oxymoron /ˌɒk.sɪˈmɔː.rɒn/ (noun)

Greek. ὀξύμωρον ("sharp dull"). A figure of speech that combines contradictory terms.

This chapter presents the sweet spot of effective code reuse mitigation and code sharing savings: A secure fine-grained memory randomisation technique on a per-process level that does not interfere with code sharing.

Previous fine-grained memory randomisation solutions by definition rendered memory sharing techniques impossible. Because there is intentionally no identical code in any two processes, no code can be shared amongst processes. Normally, sharing mimics the existence of the same library in different processes by mapping the library's memory pages into different address spaces. This sharing is a fundamental concept of modern operating system design and minimises overall memory consumption. A dysfunctional code sharing, however, is not just a step backward in operating system design, but also increases the memory footprint of the entire system, likely on the order of Gigabytes, as elaborated in section 7.2.

7.1 | The Contribution to Science and My Part in it

Oxymoron cuts executables and libraries into the smallest sharable piece: a memory page. Those individual pages are then shared amongst processes. Each shared page appears at a different, random address in each process. For this technique to work, code can no longer refer to other code or data using addresses as they are different in each process. Instead, Oxymoron transforms existing code into *Position-and-Layout-Agnostic CodE* (PALACE). PALACE code uses no instructions that reference other code or data directly, but instead the instructions use a layer of indirection referred to by an index. This index uniquely identifies a target and hence remains identical when targets are randomised in memory. Consequently, the memory in which those instructions are stored does not change, thereby making it available to be shared with other processes. The unique indices are organised in a translation table. The x86 processor's segmentation feature is used to hide the actual addressed behind an index.

Oxymoron works by changing existing code to Position-and-Layout-Agnostic CodE. To this end, I invented PALACE and a translation from legacy binary code to PALACE code. Therefore, I adapted the binary rewriter presented in chapter 6 to support PALACE. For this purpose, I extended its functionality to output static binaries instead of in-place memory transformation. The resulting binaries and shared libraries do not incorporate any direct memory references anymore. This way, the executables and libraries only have to be transformed once and continue to run on an unmodified Linux.

To demonstrate the effectiveness and efficiency, SPEC 2006 benchmarks were run and yielded a very low run-time overhead of only 2.7%. By re-enabling code sharing, Oxymoron is the first memory randomisation technique that reduces the total system memory overhead back to levels it was before fine-grained memory randomisation.

7.2 | Primer on Memory Sharing

The goal of fine-grained randomisation is to feature a different memory layout for every process – even identical programs. Chunking code in pieces and shuffling the puzzle pieces throughout the entire address space yields a very high entropy. However, sharing those puzzle pieces with other processes is only possible if the content of each piece is identical in each process. With traditional code, the content of those piece necessarily changes when their order in memory is rearranged, because references in code need to be adjusted accordingly.

Code references other code or other data using either absolute memory addresses, e.g. `call 0x804bd32`, or relative addresses, e.g. `call +42`. For absolute addresses it is obvious that different randomisations necessarily lead to different code and data addresses. As a result, the encoding of instructions that hold such addresses changes as well, thereby forfeiting the sharing with other processes.

For example, a call to function `foo()` is only possible if the entry point of `foo` is known. In x86 assembly this is written as

```
call 0x804bd32
```

Listing 7.1: Example Absolute Call to Function

Relative addresses, in turn, make code independent of its load address in memory. Making code independent of its base address can be achieved using *position-independent code* (PIC). Here, function `foo()` is called relative to the current instruction, e.g.

```
call +90
```

Listing 7.2: Example Relative Call to Function

However, x86 only uses relative encoding in their `call` and `jump` instructions. Thus, the absolute call from the example above is actually encoded as the relative distance to that function. In any case, relative and absolute distances change when code pieces are moved in memory. Those pieces cannot be shared across processes as they feature different absolute/relative addresses encoded in the instructions. Consequently, fine-grained memory randomisation impedes common code sharing, which is a fundamental concept of *all* modern OSes.

Severity. Modern operating systems make use of code sharing because most of the running programs use the same libraries (C library, threading library etc.). As a result, different address spaces have identical code loaded – even for different programs. For multiple instances of the same program, all code and read-only data is shared and hence only occupies space in RAM once – regardless of how many instances are loaded simultaneously. Only the corresponding data and stack is private to each process.

A simple experiment was conducted to measure how much RAM is actually saved due to code sharing and therefore how much is at stake. The experiment used Ubuntu 13.10 x86 operating system on a machine with 4 GiB of RAM. After booting to an idle desktop, the 234 running processes consumed a total 679 MiB for code, data, heap and stack. The memory mappings in `/proc/<PID>/maps` revealed that most of the processes used the same set of shared libraries. Two libraries were even loaded in every single processes: the standard C library (`libc.2.17.so`) and the Linux linker itself (`ld-2.17.so`).

The sharing of code is realised by mapping the physical RAM portion that holds the actual code into many other address spaces instead of duplicating it. Memory page mappings in each process obtained from `/proc/<PID>/maps` revealed that `libc.2.17.so` was shared between all of the 234 processes. All mapped portions of `libc` sum up to 207,028 KiB while only 885 KiB of real memory are consumed. This is a savings of 206 MiB for `libc` alone.

Figure 7.1 illustrates the top ten savings by library. In total, sharing instead of duplicating saved 1,388 MiB of RAM on the idle Ubuntu desktop. When additionally starting the Firefox browser, the memory consumption was increased from 679 MiB to 817 MiB. The total amount of savings by sharing summed up to 1,435 MiB of RAM, which is an additional savings of 47 MiB.

File	Occurrences	Average Mapped Size	Saving
`/lib/i386-linux-gnu/libc-2.17.so`	234	860 KiB	196.6 MiB
`/usr/lib/i386-linux-gnu/libicudata.so.48.1.1`	20	8482 KiB	165.7 MiB
`/usr/lib/i386-linux-gnu/libgtk-3.so.0.800.4`	44	2531 KiB	108.8 MiB
`/usr/lib/i386-linux-gnu/libgio-2.0.so.0.3800.0`	130	726 KiB	92.2 MiB
`/lib/i386-linux-gnu/libglib-2.0.so.0.3800.0`	136	514 KiB	68.3 MiB
`/usr/lib/i386-linux-gnu/libstdc++.so.6.0.18`	106	446 KiB	46.1 MiB
`/usr/lib/i386-linux-gnu/libX11.so.6.3.0`	64	600 KiB	37.5 MiB
`/usr/lib/i386-linux-gnu/libLLVM-3.3.so.1`	4	8747 KiB	34.2 MiB
`/usr/lib/i386-linux-gnu/libcairo.so.2.11200.16`	48	562 KiB	26.3 MiB
`/usr/lib/i386-linux-gnu/libxml2.so.2.9.1`	32	665 KiB	20.8 MiB
`/usr/lib/i386-linux-gnu/libgobject-2.0.so.0.3800.0`	130	161 KiB	20.4 MiB
`/lib/i386-linux-gnu/libpcre.so.3.13.1`	156	127 KiB	19.4 MiB
`/usr/lib/i386-linux-gnu/libicui18n.so.48.1.1`	20	889 KiB	17.4 MiB
`/lib/i386-linux-gnu/libm-2.17.so`	132	131 KiB	16.9 MiB
`/lib/i386-linux-gnu/libdbus-1.so.3.7.4`	114	147 KiB	16.3 MiB
`/usr/lib/i386-linux-gnu/libpixman-1.so.0.30.2`	50	331 KiB	16.2 MiB
`/usr/lib/i386-linux-gnu/libfreetype.so.6.10.1`	50	310 KiB	15.1 MiB
`/lib/i386-linux-gnu/ld-2.17.so`	234	66 KiB	15.0 MiB
`/usr/lib/i386-linux-gnu/libicuuc.so.48.1.1`	20	669 KiB	13.1 MiB
`/lib/i386-linux-gnu/libgcrypt.so.11.7.0`	52	255 KiB	12.9 MiB
`/usr/lib/i386-linux-gnu/libgdk-3.so.0.800.4`	44	285 KiB	12.3 MiB
Other Libraries	(584)	$\approx$ 603 KiB	352.2 MiB

Table 7.1: Top 20 libraries in all processes of the idle Ubuntu desktop, grouped by the same library (*File*), how often it was loaded (*Occurrences*) and the resulting savings.

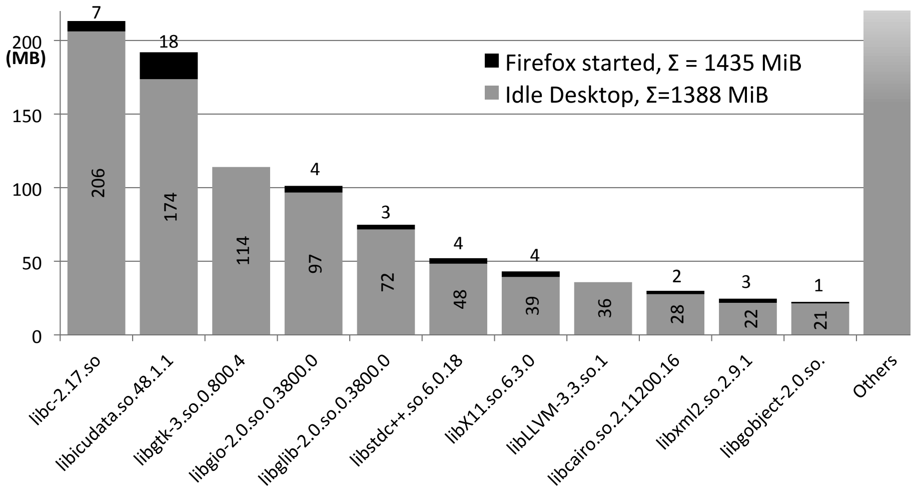


Figure 7.1: Savings due to sharing of libraries. Idle desktop saves 1388 MiB, with Firefox 47 MiB more are saved.

7.3 | Threat Model

The attacker model assumes a Linux operating system that runs a user mode process, which contains a memory corruption vulnerability. The attacker's goal is to exploit this vulnerability in order to divert the control flow and execute arbitrary code on her behalf. To this end, the attacker knows the process' binary executable and can precompute potential gadget chains in advance. The attacker can control the input of all communication channels to the process, especially including file content, network traffic, and user input. However, the attacker has not gained prior access to the operating system's kernel and the program's binary is not modified. Apart from that, the computational power of the attacker is not limited.

7.4 | High-Level Design of Oxymoron

To benefit from the best of both worlds – fine-grained memory randomisation and code sharing – the challenge is to create a form of code that does not incorporate absolute or relative addresses, as we have already shown that both addressing schemes by definition suffer from being dependent on their randomisation. An additional layer of indirection that translates unique labels to current randomised addresses allows the byte representation of code to remain the same, which enables code page sharing. However, this approach is difficult to realise as the following four requirements are crucial for its success:

1. The size of the translation table shall be small in order not offset the memory that is saved by sharing,
2. A layer of indirection must be efficient in order to be practical,
3. The translation must be inaccessible for adversaries,
4. The solution shall run on a commodity, unmodified Linux OS.

Overall Procedure. Oxymoron prevents code reuse attacks by shuffling every instruction of a program to a completely different position in memory so that no instruction stays at a known address, thereby making it infeasible for an adversary to guess or brute-force addresses. Oxymoron features a code representation that does not incorporate absolute or relative addresses. In this way, several processes can map shared libraries and executables that are split in pieces in their address space, but at random addresses.

Oxymoron uses a three-step procedure (see Figure 7.2):

A) **Code Transformation**: The executable E is transformed to Position-and-Layout-Agnostic CodE (PALACE). The result is a PALACE-code executable P_E. The same applies to shared libraries, which can be treated like executables.

B) **Splitting**: The P_E code is then split into the smallest possible piece that can be shared among processes: a memory page. The code of P_E now consists of code pieces $P_E = p_1|p_2|\dots|p_n$.

C) **Randomisation**: At program load time, the pieces $p_1|p_2|\dots|p_n$ are shuffled by the ASLR part of the operating system loader. In memory, their order is completely random and the pieces may have gaps of arbitrary size between them.

The first two steps only have to be done once, while the third step is performed at load time of the executable P_E.

7.4.1 | Code Transformation

To enable layout-agnostic code, all references to code and data are replaced with a unique label. Such a unique label is an assigned index into a translation table. This *Randomisation-agnostic Translation Table* (RaTTle) in turn refers to the actual target (see Figure 7.3). This is the key to code sharing amongst

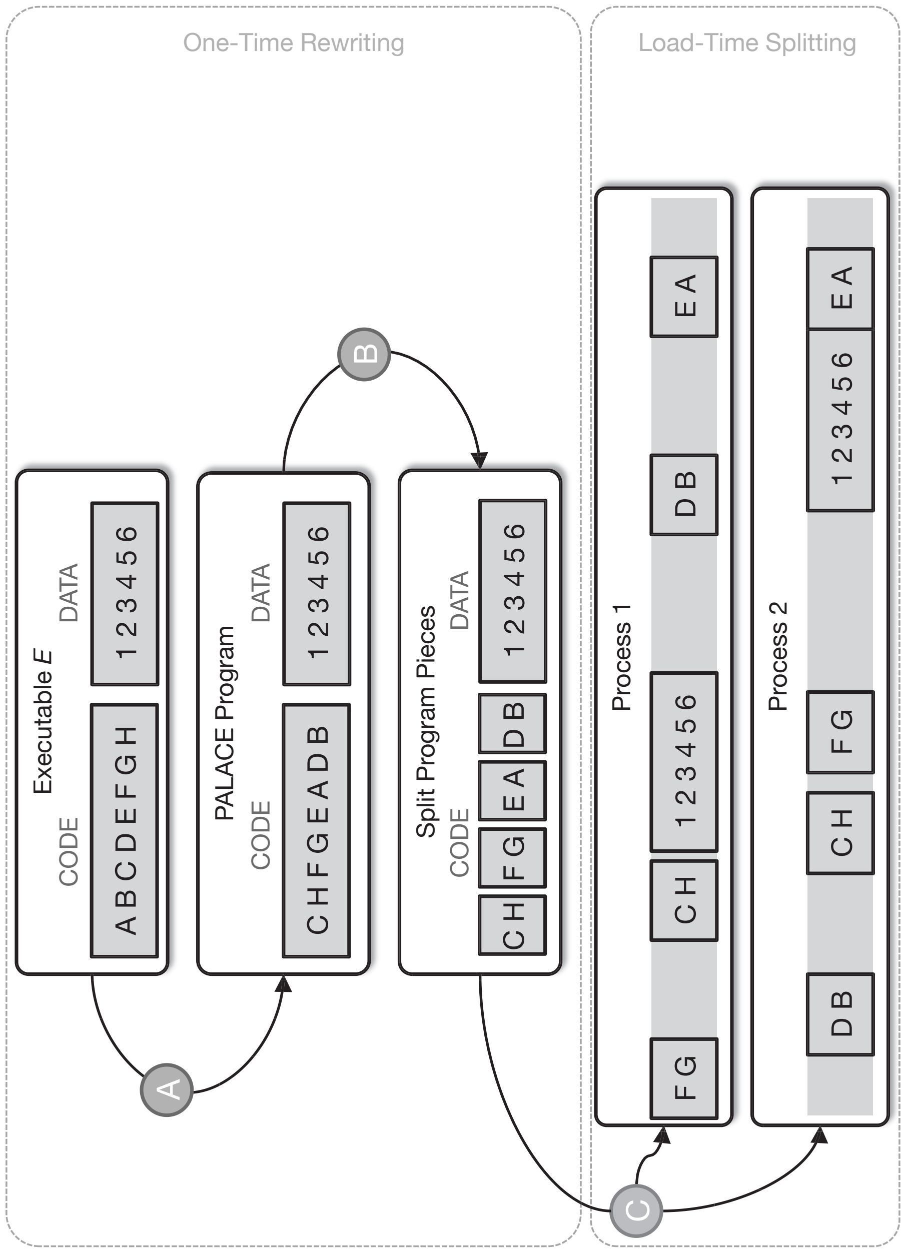


Figure 7.2: The program is transformed and split once (A and B), then randomised at every process start-up (C).

processes, since the byte representation of the PALACE code does not change in the next step, when it is split and individual pieces are shuffled in memory.

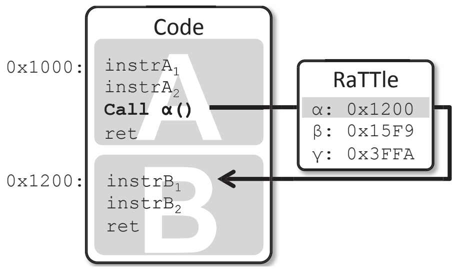


Figure 7.3: Control-flow is redirected through the RaTTle rather than jumping to addresses directly.

7.4.2 | Splitting

Splitting ensures that the resulting pieces can be mapped into different processes at different addresses. As PALACE code references every target through a unique label in the RaTTle, it can be split without the need for traditional relocation, which would rewrite addresses that need to be changed.

The PALACE code is split into page-sized pieces. If those pieces are later shuffled, it must be assured that the original semantics of the program are kept intact. This requirement is identical to the fine-grained randomisation explained in section 6.2. This is essential when control flows from the end of one piece to the piece that was adjacent to it in the original program code. Thus, explicit control flow instructions are inserted at the end of code pieces that might be moved away in a later stage of randomisation. These explicit links only need to be inserted as the last instruction of a piece to ensure that control indeed flows to the intended successor (see Figure 7.4). After the links have been inserted, the code pieces can be randomised in memory without violating the original program semantics.

7.4.3 | Randomisation

Modern OS loaders for shared libraries already support Address Space Layout Randomisation (ASLR), i.e. they load the code, data, and stack segments at random base addresses. Oxymoron leverages this fact by putting every memory page in its own loadable segment of the executable file or of the shared library. As the page-sized code pieces are already transformed to PALACE code, no

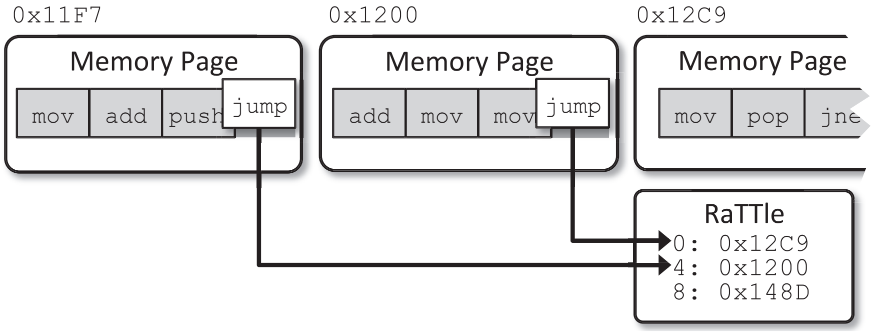


Figure 7.4: Filling a page with instructions and linking them with explicit control flow transfers.

relocation of addresses is needed. An ASLR-enabled conventional loader can blindly load all pieces at random addresses. Consequently, each process can have its own permutation of the randomisation. Only the RaTTle needs to be kept up to date with a per-process randomisation.

7.4.4 Addressing the RaTTle

At first glance, it might seem we have only shifted the problem of addressing functions in code to securely addressing the RaTTle. However, the RaTTle can be accessed by control flow instructions but not be read, e.g. by adversaries. To understand *how* the RaTTle works, it is worthwhile to elaborate on existing approaches – and why they are all not suitable. As already alluded to by Shacham et al. [P12], the following techniques have drawbacks:

Fixed. Storing the RaTTle at a fixed address in memory allows for its address to be hard-coded in the instructions themselves. Unfortunately, a hard-coded address restricts the table to a fixed position. This fact can be exploited by an attacker.

GOT. Accessing the GOT is realized by using relative addresses, which forfeits sharing as discussed earlier. Moreover, several attacks are known that dereference the GOT [L17].

Register. A dynamic address that is randomly chosen for every process could be stored in a dedicated machine register. However, this register would need to be sacrificed and every original use of that register must then be simulated with other registers or the stack. Moreover, a leakage vulnerability could reveal the address of the RaTTle.

Oxymoron Approach. The RaTTle does not suffer from the aforementioned drawbacks. Oxymoron uses the x86 feature of memory *segmentation* to address and at the same time hide the RaTTle from adversaries. X86's segmentation is no longer used today because it has been superseded by memory paging. Memory paging, also called virtual memory, allows a fine-grained mapping of memory on a per-process basis and is much more versatile than segmentation. However, segmentation is still available in modern processors and in combination with paging allows for the security we need for the RaTTle. Additionally, as segmentation is a hardware feature and we can use it to implement the translation table, it is very efficient. Segmentation allows the memory to be divided in user-defined segments that may overlap (see Figure 7.5).

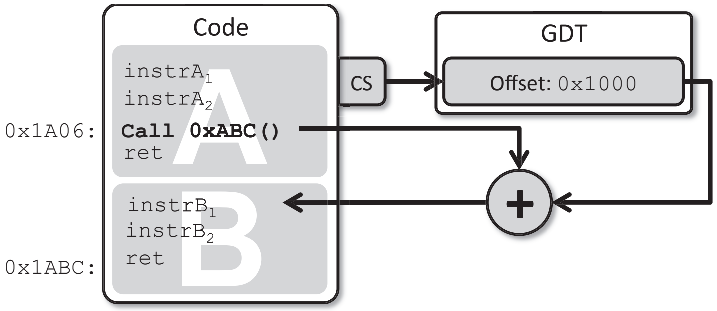


Figure 7.5: Code using segments as offsets for addresses.

Segmentation is realised in the processor by adding a user-defined offset to all addresses the processor uses. The offset itself is defined by choosing from a list of segments via a *Segment Selector*. Each segment in turn has its own *base address* and *limit*, i.e. the start and length of that segment. The list of those so-called *Segment Descriptors* is kept in the *Global Descriptor Table* (GDT, see Figure 7.5). A *Segment Selectors* must then point to exactly one segment entry in that GDT. Segment selection is done using dedicated segment selector registers such as `CS` (Code Segment), `DS` (Data Segment), `SS` (Stack Segment) and three general purpose segment selectors `ES`, `FS` and `GS`.

Position-and-Layout-Agnostic CodE (PALACE). By default, `CS`, `DS` and `SS` all point to the entire virtual address space (e.g. 0 to 4 GiB). Oxymoron does not change that. However, it is possible to select a different segment for a single instruction. In this way, a single instruction may use an addressing that is relative to the RaTTle, thereby indexing the RaTTle to change control flow or to access data. For example, `call *%fs:0x4` dereferences the double-word stored at `%fs:4` and calls the function stored at that double-word. If we let the segment selector FS point to the randomly chosen address of the RaTTle, we effectively index the RaTTle by an offset of 4 (see Figure 7.6).

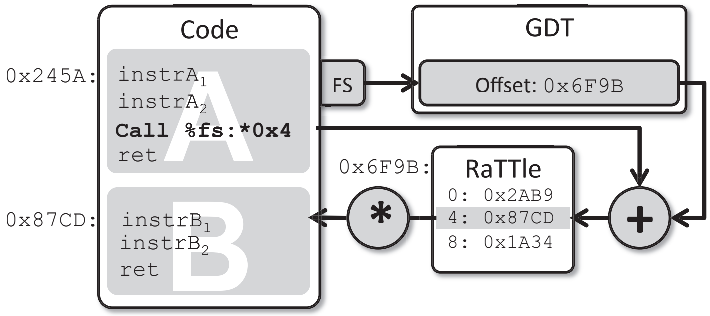


Figure 7.6: The RaTTle in Action: Indexed through the GDT and dereferenced using an indirect call; all in *one instruction*.

PALACE code substitutes each branch and jump instruction with an %fs segment override and a unique index. When not using the FS segment override, code does not have access to the RaTTle because it uses a different segment. The address of a segment, and hence of the RaTTle, cannot be read from user space because the local and global descriptor tables point to kernel space memory which is inaccessible from user space. This makes the address of the RaTTle inaccessible. As a segment selector for the RaTTle, we chose the general purpose segment selector register FS, as already used in the example above. This register is usually unused. Some exotic applications may use it though, e.g. the Windows emulator Wine that uses segmentation for its 16-bit Windows emulation.

Note: the GS register is already used in a similar fashion. Linux stores data that needs to be globally accessed GS-relative, e.g. thread-local storage or the system-specific syscall implementation (as shown in Listing 7.3):

```
int main(int argc, char* argv[])
{
        __asm__(
                "movl␣$20,␣%eax\n"
                "call␣*%gs:0x10\n"
                "movl␣%eax,␣pid\n"
        );
        printf("pid␣is␣%d\n", pid);
        return 0;
}
```

Listing 7.3: A Linux syscall to getpid() using a GS-relative call

Efficient Data Access. Data can be accessed in a similar way, but through the Global Offset Table (GOT). The GOT is used in position-independent code such as libraries anyway. To this end, the typical calculation of the GOT's address must be substituted with an indirection through the RaTTle. Further access is done through the GOT as in traditional position-independent code. This is explained in more detail in subsection 7.5.5.

Populating the RaTTle. The RaTTle is the only part of the code that needs rewriting at load time. The RaTTle is empty in the ELF executable file on disk and its memory gets initialised by the loader with the help of relocation information. This relocation information points to the actual symbols that each RaTTle index refers to. The Linux loader automatically takes the relocation information to rewrite the RaTTle at program load [D8].

7.5 | Design Details

With the ingredients described earlier, we can put together Oxymoron, a mitigation against code reuse attacks that is efficient, lightweight, and shares code and data between processes.

7.5.1 | Design Decisions

There are several ways to implement PALACE. A PALACE executable can be produced by a compiler, or it can be transformed from a traditional executable using static or load time translation.

Compiler Support. The same way contemporary compilers support PIC, they can be augmented to emit PALACE code. Based on the principles of PALACE code, the compiler needs to generate PALACE code and put it in subsequent memory-page-sized chunks. It is then ready to be loaded by a traditional loader that permutes the chunks prior to execution of the code.

Static Translation. If the source is not available, an existing executable can be transformed to PALACE by means of static translation [P82, P83]. Static translation reads an executable or shared library file from disk, disassembles it, transforms the instructions, and writes a modified executable file back to disk. For PALACE code, static translation keeps most of the instructions untouched while only replacing code and data references with the appropriate indirection through the RaTTle. However, static translation needs to disassemble compiled instructions which cannot be done reliably in all cases [P81].

load time Translation. load time Translation can be regarded as a static translation that happens automatically at every load time, after the executable or library has been read from disk into memory but before it starts execution. Its advantage is that a process can be randomised at every startup. In our scenario, however, we do not need load time translation as we can achieve a randomisation at load time with specially crafted PALACE chunks and small ASLR pieces in the executable file.

Decision. Oxymoron can be implemented by a compiler that simply emits PALACE code instead of traditional code. However, I chose to build a legacy-compatible solution that uses static translation and can be built on an existing fine-grained memory randomisation framework, which already uses static translation. Here, the rewriter from chapter 6 comes in handy.

In theory, a static translation approach may seem fragile because it needs a perfect disassembly. However, static translation can be tuned to reliably disassemble code generated by a particular compiler with known and carefully chosen parameters. Besides, in this thesis, the translation from traditional x86 code to PALACE code is used as a comprehensible running example that compares traditional x86 code to PALACE code side-by-side.

PALACE Code and the Linux Loader. In both cases, compiler and static translation, the generated PALACE code of the executables and libraries can be read by a commodity Linux. The Linux OS loader will detect the executable as being ASLR-enabled and will randomise its base address. Unfortunately the commodity loader does not randomise the program segments individually but keeps their relative distances. For traditional position-independent code that was necessary so that code in the `.text` section can still reference objects in the `.data` section by their relative distance to the current instruction pointer. However, for PALACE this limitation is not required. In contrast, a more fine-grained randomisation is desirable that allows to place program segments individually in memory. This can be achieved by requesting a special linker in the program header, which randomises the segments individually. This achieves the best of both worlds: a randomisation for every process start and legacy compatibility as we do not need source code to transform executables.

7.5.2 Setting up the RaTTle

During the creation of PALACE code, the RaTTle needs to be filled with unique indices that can be assigned to call sites and data references. Later, during load of the rewritten program, the RaTTle needs to be loaded at a random address and then populated with all references in the executable. Before the RaTTle can be used, it has to be set up as follows:

1. During PALACE code generation, assign every reference in code a unique number that will act as an index into the RaTTle,

2. During program loading, fill the RaTTle with the actual, current, random addresses of the original targets, and
3. Set up segmentation so that a free segment selector points to the RaTTle such that it can be indexed.

In step 1, the absolute addresses of the original program are saved in a hash set. Then, every address is assigned an ascending index. This ensures that the table does not grow unnecessarily large. Since the final, random addresses are unknown before the process is started, the RaTTle cannot be filled until start-up of the process. Because the operating system loader shall not be modified, the RaTTle must be populated using only traditional features of the loader. Such a feature is relocation. Relocation information tells the loader which objects in the executable file or in the library must be overwritten with current addresses at load time. Therefore, for each RaTTle index, relocation information is added to the final executable/library file. This ensures that the loader rewrites each index so that it points to the corresponding position of code or data that this index represents. As a result, the randomised addresses of the code pieces are automatically written into the RaTTle by the operating system loader during start-up.

7.5.3 Setting up Segmentation

In order to find the RaTTle in memory, the x86 segmentation must be set up so that a pre-defined segment points to the beginning of the RaTTle. Setting up segmentation requires the Global Descriptor Table (GDT) to be changed. Changing the GDT can only be performed in kernel mode. Since the goal is to avoid operating system modifications in order to stay legacy compatible, this is not an option. Luckily, the x86 architecture additionally supports a so-called *Local Descriptor Table* (LDT [D10]). The LDT can be switched for every address space, so that Linux emulates a per-process LDT. This is a perfect feature for enabling Oxymoron on a per-process basis.

GDT and LDT. The actual value that a segment selector must hold is not merely an index to the GDT/LDT, but is defined by the architecture set as follows:

Bits 15 - 3	Bit 2	Bit 1 - 0
Index into the table (0-4095)	0=GDT, 1=LDT	Privilege Level

Table 7.2: The GDT and LDT structure of the x86 architecture

As user mode is in Ring 3, bits 0 and 1 must be set to $\texttt{11}_{bin}$. Because the LDT is used, bit 2 must be set. Finally, the index is prepended, so index “0” yields

111_{bin} (7 in decimal), index “1” yields 1111_{bin} (15 in decimal) and so on. Ultimately, the selector is set up using a simple mov with the correct number: `mov $7, %fs`.

The setup of the LDT and the segment selector that points into the LDT is done in initialisation code. For this purpose, the ELF executable format supports a designated section, called `.init`, that stores initialisation code. Code in this section is guaranteed to be executed before any other code. Oxymoron uses this init code to set up the RaTTle. The code figures out the address at which the RaTTle has been randomly loaded by the loader and sets up the LDT accordingly. After the initialisation code has run, the segment selector `FS` points to the random address of the RaTTle. The PALACE code can now work as intended.

7.5.4 | Control Flow and Data

The following are examples of instructions and instruction sequences that need to be transformed by the static translation. In the following examples ’4’ is used as an index in the RaTTle as an exemplary identifier of a fictional target.

Code. Control flow branches or function calls that target another memory page need to be replaced with an indirection through the RaTTle. The simplest case is a direct `call` or an unconditional `jmp` to a different place in code:

Address	Before	After
8050512:	`call 0x8050c08`	`call %fs:4`
RaTTle:		[0] .... [4] `0x8050c08`

Only branches that reference code outside of the current memory page must go through the RaTTle. Code and data access within one memory page may be encoded position-relative (e.g. `call +90`).

If the to-be-replaced instruction is an indirect jump, the translation is slightly larger due to the fact that x86 does not support two levels of indirection. It is either possible to use the RaTTle to get the address of the second indirection and then dereference that using an indirect jump or to use a trampoline. The following example translation uses a trampoline as it is slightly faster:

Address	Before	After
8050512:	`jmp *0x80a00012`	`jmp %fs:4`
80a00012:	`8050c08`	`8050c08`
RaTTle:		[0] .... [4] `jmp *80a00012`

A slightly more complicated case is a conditional jump because there is no equivalent conditional indirect jump. Hence, the solution emulates a conditional indirect jump and is more involved:

Address	Before	After
8050512:	`cmp %eax, %ebx`	`cmp %eax, %ebx`
8050514:	`jne 0x8050590`	`jne 0x8050518`
8050516:		`jmp 0x805051a`
8050518:		`jmp *%fs:4`
RaTTle:		[0] .... [4] `0x8050590`

An *indirect jump*, such as `jmp *%eax` does not need to be replaced at all. However, the used register (in this example `%eax`) must point to the correct randomised position in memory. This is either ensured by the compiler that generated PALACE code or by the translation from traditional code to PALACE code. In either case, the `%eax` register must be loaded with a code address. Optionally, jump tables or C++ vTables modify the register prior to transferring control to the destination address.

Usually, the register used for an indirect jump is first populated with a fixed address, e.g. `mov $0x8402dbc, %eax`. In the case of PALACE, this step needs an indirection to conceal the actual address and to make the address exchangeable by the RaTTle. In PALACE code this register loading looks like this: `mov %fs:$0x4, %eax`. This copies an address stored as an entry in the RaTTle to the register `%eax`. Then, some mathematical operations can be performed, such as adding the offset into C++ vTables and finally the indirect jump is performed as in traditional x86: `jmp *%eax`.

Data Access. Accessing data through the RaTTle is done in exactly the same way. An indirect memory operation is used to read or write data from or to an address stored in the RaTTle. The instruction `mov %fs:$0x4, %ebx` is used to read the first entry (4 bytes) of the RaTTle into register `%ebx` and vice versa the operation `mov %ebx, %fs:$0x8` copies the the content of register `%ebx` to the second entry of the RaTTle.

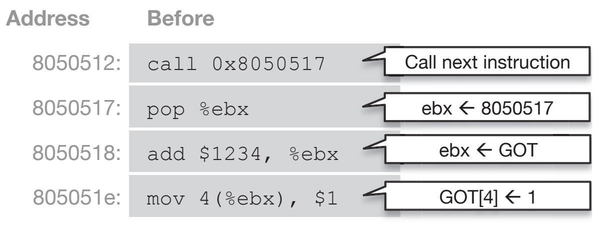
7.5.5 Inter-Library Calls and Data

Control flow and access to data is not restricted to one library or executable. Naturally, these code elements frequently use each other's functions and data. Some operating systems, like Windows, use relocation information to directly patch the control flow so that it points into a library after it has been loaded. Linux, on the other hand, uses the procedure linkage table (PLT) to link calls to libraries with the advantage of lazy loading. In the beginning, PLT entries do not point to the actual procedure inside a library because it has not been loaded yet. Instead, they point to code that loads the library and then rewrites the PLT to link the call to the actual target procedure. This way, when a program calls `printf@plt`, the stub stored in the printf entry of the PLT first loads the C library and then replaces the call with a call to the actual address of `printf` inside the loaded C library.

Similarly, Oxymoron uses an indirection through the RaTTle for every library call or access to global library data because this approach conceals the actual address of the loaded library and has only minor performance impact.

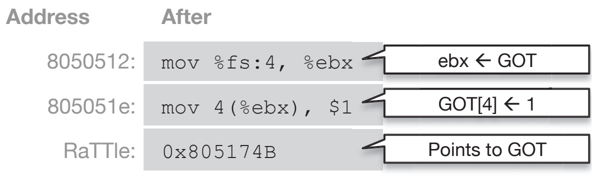
Inter-Library Data. Libraries can export data to be used by the executable main process or other shared libraries. Since it is known a priori which data is accessed in another library, each reference gets a place-holder in the GOT which can be accessed as described above. When the appropriate library is loaded by the loader, it automatically updates the GOT based on the relocation info pointing to this entry in the GOT.

The following is an example of typical position-independent code that uses a GOT to access data: The code is first calling the next instruction, thereby pushing its own address as a return address to the stack. Then, this address is popped off the stack to get the absolute address of the currently running code. The address of the GOT is calculated by adding a known offset.



```
Address    Before
8050512:   call 0x8050517
8050517:   pop %ebx
8050518:   add $1234, %ebx
805051e:   mov 4(%ebx), $1
```

When transforming this piece of code into PALACE, only the calculation of the GOT needs to be substituted. In this case, the three former instructions get compressed to a single instruction with segment override.



Interestingly, this is a faster method of accessing the GOT than the currently used PC-relative addressing for x86-32. However, this calling convention for the GOT has changed in 64 bit mode (x86-64) Linux. The call to the next instruction is not needed as x86-64 can use the instruction pointer `RIP` (formerly `EIP`) as a general purpose register. Hence, adding an offset to `RIP` directly became possible.

Inter-Library Calls. Inter-library calls are calls from one loaded library to another or from the main executable to a library. In theory, these calls are no different from a call within the same library or executable because the RaTTle can simply point to code in another library. However, in practice, this would require the RaTTle to reflect all possible combinations of loaded libraries. Therefore, a unique RaTTle is used for every loaded library. An inter-library call acts as a trampoline that changes the segment selector `FS` to point to the corresponding RaTTle of another library prior to jumping into that library (see in Figure 7.7).

Please note the missing "*" in the `call %fs:8` of Figure 7.7, which means the RaTTle is not dereferenced rather than used as a trampoline. This trampoline then lets `FS` point to the index of the other library's RaTTle without the need to know the exact address. Suppose the function that we want to call is stored at index 0 in RaTTle$_2$, but RaTTle$_1$ is currently active. The code in Figure 7.7 first sets `FS` to point to RaTTle$_2$. RaTTle$_2$ is the second selector in the LDT. Hence, the trampoline code in RaTTle$_1$ assigns $\texttt{10111}_{bin} = 23$ to `FS`, which corresponds to a segment selector of "2" (see Table 7.2).

The trampoline code then jumps to index `0`, which now corresponds to currently active RaTTle$_2$. Because the trampoline uses a `call` instruction to finally call into the other library, control flow returns to the trampoline where `FS` is restored to its former value.

Legacy Libraries. In order to work with legacy libraries that do not support a RaTTle, the traditional approach using a GOT and PLT is used. The address of the GOT and PLT, however, is resolved using the RaTTle, of course. For this

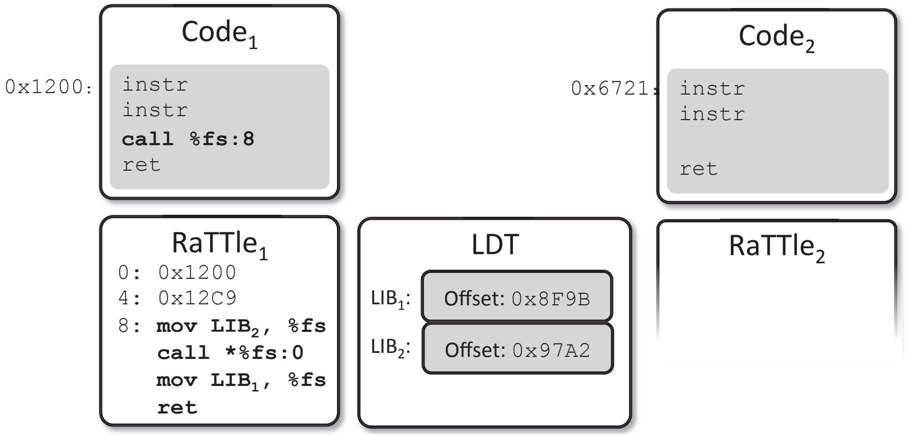


Figure 7.7: Inter-Library Calls: Because the indices overlap, a new RaTTle needs to be set up before those calls.

to work, it has to be known at compile time whether the linked library is in PALACE-form with a RaTTle or not.

7.5.6 Debugging

Debugging information augments the executable or library file with annotations describing which memory addresses correspond to which variables or lines of code. These stored addresses must be compatible with Oxymoron randomised addresses. Since Oxymoron is implemented as a static translation tool, the original debugging information needs to be translated as well. Currently Oxymoron supports the common DWARF [D9] file format which can be read by `gdb` or other debuggers. This way, it is possible to teach `gdb` the randomised addresses so that `gdb` can still step through the code, inspect variables etc. – similar to non-randomised executables.

7.6 Evaluation

This section evaluates the effectiveness of Oxymoron empirically as well as theoretically. In order to demonstrate the efficiency, the de facto standard performance benchmark SPEC CPU2006 was used together with micro benchmarks to measure cache hit/miss effects.

The security of the RaTTle is crucial, as the disclosure of randomised addresses hinge on the proper protection of the RaTTle. Also, it is important to check that the introduction of the RaTTle did not open the flood gates for other attack vectors. Then, the slightly different randomisation of memory pages that this solution entails is compared to the more classical memory randomisation solutions in order to get an understanding of the implied security.

7.6.1 | Practical Security Evaluation

We tested our randomisation solution against real-life vulnerabilities and exploits. The documented vulnerabilities `CVE-2013-0249` and `CVE-2008-2950` both allow arbitrary code execution by means of return-oriented programming [L18]. `CVE-2013-0249` targets the `libcurl` library which handles web requests and is used in dozens of popular programs, including ClamAntiVirus, LibreOffice, and the Git versioning system. The exploit for this vulnerability is crafted in such a way that it triggers a buffer overflow in `libcurl` with the ability to overwrite a return address and ultimately execute a chain of ROP gadgets. The severity of this bug lies in the fact that it can be triggered remotely when `libcurl` accesses a prepared resource that is under the control of the adversary. In order to test the exploit, a program that uses `libcurl` was fed with the exploit. In particular, the stand-alone `curl` downloader executable in version 7.28.1 was used. Due to the vulnerability in that version, it was possible to run arbitrary code by chaining together ROP gadgets. After curl had been rewritten to use Oxymoron, the exploit was no longer possible as the addresses that are needed to successfully mount the attack are unknown due to the randomisation at every program start.

Similarly, the vulnerability `CVE-2008-2950` allows for arbitrary code reuse in the PDF library `poppler`, which is used by many popular programs such as LibreOffice, Evince and Inkscape. A specially prepared PDF file can trigger an arbitrary memory reference in the `poppler` library, ultimately leading to a code reuse attack. Not surprisingly, an exploit against `pdftotext` using `libpoppler` 0.8.4 was successful. After applying Oxymoron to the `pdftotext` executable, the memory address of the PALACE-protected process were no longer known and consequently the exploit was rendered unsuccessful.

7.6.2 | Security of the RaTTle

The RaTTle holds lots of random addresses and, at first glance, seems like a valuable target for an attacker. The security of the RaTTle originates from the fact that its address is unknown and that its content cannot be accessed.

All `%fs`-instructions are mere replacements for control flow branches and as such only use the RaTTle as a layer of indirection without ever knowing the actual address of the landing position. If an `%fs`-instruction is a replacement for data access, the same holds true: The RaTTle is only used for indirect access of the actual data. In general, the x86 architecture does not support revealing addresses that segments point to. The only way to read the address is to parse the GDT or LDT which both reside in kernel space. To access the LDT, a user mode program needs to issue a special syscall. Even if a program would consist of ROP gadgets to issue this syscall, an attacker would still need to know the addresses of the required ROP gadgets. Hence a successful attack can be reduced to finding special instructions that can be used as ROP gadgets. This has a negligible probability as explained in *"Theoretical Security Evaluation"*.

Because processes are protected by $W \oplus X$ (data execution prevention), no code can be injected by an attacker. Hence, the only possibility is to reuse existing code. This existing (PALACE) code is in fact littered with `%fs`-prefixed instructions that implicitly point to the RaTTle due to the sheer fact they incorporate a reference to `%fs`. Again, this situation is identical to finding ROP gadgets in a classical program, as an attacker needs to know their randomised position in memory in order to chain them together. The fact that this address is not known to an attacker prevents the reuse of code. In fact, the probability of guessing a correct address is negligible (see subsection *"Theoretical Security Evaluation"*).

7.6.3 | Enhanced Security of the RaTTle

It is possible to further enhance the security of the RaTTle by making it completely inaccessible. The segmentation principle of the x86 architecture allows to distinguish code access from data access. This way, it is possible to set up two different RaTTles, one for code going through `%fs` and one for data going through `%gs`. First of all, in a program without self-modifying code, there should be no instruction that reads from the code segment. Consequently, there should be no instruction in PALACE code that reads data using the `%fs` code segment selector. Even if there was such an instruction, the processor would prohibit such access. Further, it is possible to move the RaTTle completely outside of the normal, otherwise flat[1] data segment (`%ds`). This results in the inability for code to ever access the RaTTle without using proper segment selectors, because it no longer resides in the accessible segment. This is an effective protection against leakage and disclosure attacks (see subsection *"Disclosure Attacks"*). Also, the call stack could be protected using this method. If return

[1]A flat segment is a segment that covers the entire address space, i.e. `0x00000000` to `0xFFFFFFFF` on a 32-bit system. This is the default for Windows, Linux and MacOS.

addresses are not saved on the regular stack, but rather on a side stack in a reserved area inside the RaTTle, there is no way for memory disclosure exploits to ever read return addresses and thus they cannot gain information about function addresses.

7.6.4 | Theoretical Security Evaluation

This subsection elaborates why the entropy of memory page granularity randomisation is still sufficient for fine-grained randomisation and why it is much higher than traditional ASLR.

First, it is shown that the entropy induced by a page-granular randomisation is high enough in the sense that the adversary has only negligible probability of successfully guessing an address. The adversary's goal is to mount a code reuse attack against a running program consisting of the executable and its loaded libraries. Hence, his goal is to know the address of either a particular function f of interest (return-into-libc attack) or of several particular instructions $i_1, i_2, \ldots i_k$ to build gadgets for, i.e. a ROP attack. Since the contents of a memory page can be extracted from the executable file, the attacker can determine in which memory page the instruction in question resides. Therefore, the success of the adversary relies on the probability of knowing the address of a particular memory page.

Every memory page is assigned a random address at load time. Thus, the first page can choose 1 out of n possible page-aligned address slots. The second 1 out of $n-1$ and so forth. For p total process pages to lay out in memory, this yields a total of

$$n \cdot (n-1) \cdot (n-2) \ldots (n-p-1) = \frac{n!}{(n-p)!}$$

combinations. The adversary's probability of correctly guessing one address is hence the reciprocal

$$\frac{(n-p)!}{n!}.$$

In a 32 bit address space, we have

$$n = 2^{19} = 524,288$$

possible page addresses. The probability of guessing one page correctly therefore is 2^{-19}. That scenario is intuitively identical to ASLR which only randomises the base address of the code. However, when finding ROP gadget chains, the page granularity drastically lowers the chance of success compared to ASLR because several pages have to be guessed correctly. For a 128 KiB

($p = 32$ pages) executable to lay out in memory, the adversary's probability of guessing the correct memory layout therefore is:

$$Pr\left[Adv_{layout}\right] = \frac{(n-p)!}{n!} = \frac{(2^{19}-2^{5})!}{2^{19}!} = 2^{-608}$$

ASLR. The calculation shows that for one big page ($p = 1$), the formula above yields

$$Pr[Adv_{1page}] = \frac{(n-1)!}{n!} = \frac{1}{n} = Pr[Adv_{ASLR}] = 2^{-19}$$

According to Shacham et al., the number n of page-aligned addresses to choose from for ASLR is rather limited [P12]. In fact, for ASLR using Linux' PaX it is lower than the theoretical entropy of $n = 2^{19}$ for a 32-bit address space. Shacham et al. state that it is rather much closer to $n = 2^{16}$. The probability of an adversary correctly guessing the address of a function (ret2libc attack) or a ROP chain consisting of k instructions is hence:

$$Pr[Adv^{ASLR}_{ret2libc}] = 2^{-16} \quad \text{and} \quad Pr[Adv^{ASLR}_{ROP}] = 2^{-16 \cdot k}$$

Shacham et al. further showed that such a low probability of only 2^{-16} can be brute-forced in a couple of minutes. If a program additionally suffers from a so-called *leakage vulnerability*, an adversary might even be able to calculate any particular address in the ASLR-randomised program.

Leakage Attacks in ASLR. A leakage vulnerability inadvertently reveals a valid, fresh address inside the running program. If the adversary additionally knows which object or function has been leaked, he then knows where a particular object or function has been loaded. In the case of ASLR, the entire code segment has been shifted in memory en bloc. As a result, the relative distances between functions stay exactly the same and can attacker can thus infer the current addresses of all other objects or functions.

In the threat model (section 7.3), the attacker can exploit an existing leakage vulnerability in order to learn a valid address. More formally, the adversary has access to an oracle that can tell which function f has leaked and the adversary can use the leakage vulnerability to learn the current address A_f of the function f. The adversary actually wants to divert control flow to function g and hence wants to know A_g. To this end, he calculates the offset of the functions in the executable file on disk:

$$d = E_g - E_f$$

d now denotes the offset between the two functions in the executable file, which is identical in memory as the entire file has been loaded into memory at a random base address. Thus, the adversary can infer the current address A_g of g by calculating the difference to the leaked function f:

$$A_g = A_f + d$$

In the case of traditional ASLR, the address of any function g' can be calculated with probability 1. The adversary can calculate a function $A(g')$ that will always return the current address of g', given a leaked function f and its address A_f:

$$A(g') = A_f + E_{g'} - E_f$$

The success probability for a *return-to-libc* attack of the adversary, *given an exploitable leakage vulnerability*, is always 1. If the adversary needs several desired addresses, i.e. a ROP attack, the probability to find k gadgets is $1^k = 1$

$$Pr[Adv_{ret2libc}^{ASLR}] = 1 \quad \text{and} \quad Pr[Adv_{ROP}^{ASLR}] = 1^k = 1$$

Ultimately, the success probability of the adversary entirely depends on the likelihood of finding such a leakage vulnerability.

Leakage Attacks in Oxymoron. In Oxymoron's case of memory page granularity shuffling, the relative distance between functions varies in general since the code segment is not simply shifted en bloc. Even though the addresses of functions are slightly shifted towards higher addresses because instructions were inserted into the memory pages to connect them, it is assumed that the adversary may calculate the exact offset by which content of a memory page has been shuffled. The adversary may obtain such offsets by running Oxymoron on the vulnerable executable. The knowledge of the adversary hence encompasses the content of each memory page in the executable file. He does not how, however, the order of the memory pages.

In the following it is assume that the adversary wants to divert control flow to function g and that the adversary, again, has access to the leaked pointer f. If the desired function g coincidently resides in the same memory page, the attacker knows the exact address A_g of g. For any leaked pointer f, there is a chance that it resides in the same memory page as the desired function g. For a program of a total size of only one memory page (4 KiB), both functions f and g must reside in the same memory page. In an assumed probabilistic equal distribution of the desired function g in p pages, the likelihood of g being in the same page as f is $\frac{1}{p}$. Hence

$$Pr[Adv_{ret2libc}^{PALACE}] \leq \frac{1}{p} \quad \text{and} \quad Pr[Adv_{ROP}^{PALACE}] \leq \frac{1}{p^k}$$

Disclosure Attack. The following distinguishes between a leakage and a disclosure vulnerability:

- **A Leakage Vulnerability** is a single leaked pointer as described above. This enables an adversary to correlate an actual current address of an object in memory.
- **A Disclosure Vulnerability** allows an attacker to read arbitrary memory content given its address.

Snow et al. proposed just-in-time code reuse, which showed that a disclosure vulnerability can significantly reduce the security of fine-grained memory randomisation [P13]. Just-in-time code reuse repeatedly exploits a memory disclosure vulnerability to map portions of a process' address space with the objective of reusing the so-discovered code in a malicious way. In a fine-grained randomisation such as [M2], the memory pages are scattered across the address space and scanning with arbitrary memory addresses is very likely to end up in unmapped memory. In order not to trap into unmapped memory, they rely on a leakage attack to learn a valid address and then start from this address by disassembling the code in order to follow control flow instructions. Even fine-grained randomisation can be reversed using their technique by transitively following the control flow.

However, in this setting of PALACE code, no control flow branch can be followed by reading memory as such a branch only constitutes an offset into the RaTTle. In order to resolve branches such as `call *%fs:4`, the attacker would need to know the address of the RaTTle or the content of the `%fs` register. Neither is possible, as alluded to earlier. The only chance an attacker has is to rely on a leakage vulnerability to get a valid address. If that address points to data it is useless to the attacker. If it points to code, the attacker can only use a disclosure vulnerability to get the contents of up to a whole memory page (4 KiB). Otherwise, he is likely to overrun the page and end up in unmapped memory which triggers a page fault that kills the program.

Stack Leaks. Another possibility for an attacker is to hope for a leakage vulnerability that enables him to read addresses from the stack. Even though a `call %fs:04` instruction does not reveal addresses when disassembled, the `call` instruction will always put the current address of execution on the stack in order for the corresponding `ret` instruction to be able to return to that point. A possible way to circumvent this, is to replace all `call` and `ret` instructions by equivalents that follow the PALACE code principle. That means a PALACE-call instruction needs an index into the RaTTle that describes the address of the instruction *following* the PALACE-call. The PALACE-call then pushes that index onto the stack and the corresponding PALACE-ret instruction jumps to the index that was popped from the stack.

7.7 | Performance Evaluation

In order to evaluate the efficiency of Oxymoron, CPU benchmarks using de facto standard SPEC CPU2006 integer benchmark suite were conducted. The SPEC benchmark suite evaluates the performance using 12 benchmarks that have been derived from typical workloads of a variety of areas. All benchmarks compare the performance of unmodified SPEC executables to executables transformed to PALACE code. The benchmarks were performed on an Intel Core i7-2600 CPU running at 3.4 GHz with 8 GB of RAM.

7.7.1 | Static Translation Overhead

Before the executable and libraries can be shuffled in memory, they either need to be compiled with an PALACE-enabled compiler or they must be converted using static translation (see section 7.4). Even though the translation only needs to be performed once, it should be efficient in order not to introduce too much delay. The rewriting time for all benchmark programs of the Spec CPU suite was therefore also measured. The rewriting process is not exactly linear, but on average achieves between 35,000 and 700,000 instructions per second. Table 7.3 gives an overview of the timings of several programs.

Benchmark	Total # of Instructions	Rewriting Time (s)
429.mcf	12,268	0.024
462.libquantum	15,788	0.024
401.bzip2	28,087	0.056
473.astar	32,502	0.032
458.sjeng	40,438	0.101
456.hmmer	54,582	0.116
464.h264ref	170,942	0.396
445.gobmk	226,661	6.744
400.perlbench	322,084	1.084
471.omnetpp	238,978	0.316
403.gcc	942,244	3.667
483.xalancbmk	1,111,779	4.321

Table 7.3: Timings for static rewriting that needs to be done at least once. The total # of instructions include the executable and all its shared libraries.

The number of instructions per benchmark reflect the total number of instructions from the executable file itself plus its dependent libraries.

Note. This measurement rewrites the entire C-library and other dependent libraries again for each benchmark and is hence slower than just translating the main executable.

7.7.2 | Run-Time Overhead

Several factors influence the run-time overhead of Oxymoron:

- **Indirection Through RaTTle.** The processor has to execute a more complex instruction that involves an additional memory look-up.
- **`jmp` Instructions.** These jump instructions connect pages (see section 7.4) and result in additional instructions that are executed.
- **Cache Miss Penalty.** The fact that the code size increased and code has intentionally been moved away from its original dense layout introduces more cache misses due to poor locality.

The overall overhead of the SPEC benchmark tools was measured and is depicted in Figure 7.8. The average run-time overhead of all benchmarks is only 2.7% for the PALACE code and 0.5% for the additionally needed chunking into memory page-sized pieces (4096 bytes).

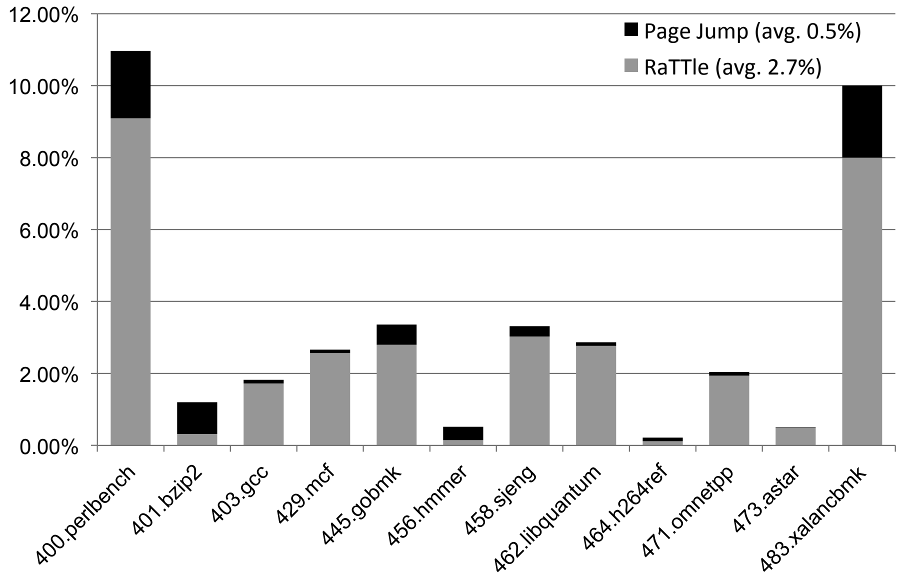


Figure 7.8: SPEC CPU2006 integer benchmark results.

Instruction Set. As different instructions have different execution times, it is worth taking a look at what happens on a micro-architectural level when replacing a normal `call` or `jmp` instruction with an inter-segment indirect equivalent. In Intel CPUs, the complex instructions are translated internally to simple instructions, so-called μOps [D10]. Each μOp can be executed in a single cycle of the CPU. Consequently, a complex instruction takes several cycles to execute. An indirect call for example gets internally divided in several μOps that actually read the address from the specified memory location and then transfer control to that dereferenced address. Replacing a simple call to a function with an indirect inter-segment call produces much more μOps than the original simple call. According to Agner Fog Research [D11], a simple call gets translated to only 2 μOps, while an inter-segment call gets translated to 31 μOps on a 'Nahelem' Core i7.

In PALACE code, accessing the GOT can be done in one instruction, whereas in 32-bit traditional x86 code, three instructions are needed. However, this single data operation is not necessarily faster than the original function call. A segment-relative move of data costs 6 cycles on the 'Nahelem Core i7' vs. the 2 cycle `call` $+1$ it replaces.

Cache Miss Penalty. The cache miss overhead that a randomisation introduces was measured separately. This is important, since modern processors assume locality of code, which might be thwarted by wild jumping in the code. The experiment was conducted on an Intel Core i7-2600 with 32 KiB L1 cache, 64 bytes per line. To measure the cache miss overhead, handcrafted code was used. This code consists of interdependent `add` instructions with a total length of one L1 cache line. These instructions are aligned in memory in such a way that they start at the beginning of a cache line and re-occur such that every cache set and every cache line is filled after execution. This resembles the baseline.

Afterwards, equidistant `jmp` instructions were inserted and the overhead of 100,000 runs was measured. The results show that the performance impact is not measurable for Oxymoron. Only when the density of `jmp` instructions increases, the overhead is measurable. If every sixth instruction is a `jmp`, a small overhead of 0.4% is introduced. If the `jmp`s get closer, the overhead rises drastically. In Oxymoron however, the `jmp` instructions that are inserted are 4,091 bytes apart (4 kiB page - 5 bytes instruction). Of course, the code already contains `jmp` instructions of the original code. Just for reference, an analysis of the `busybox` code showed that after translating it to PALACE, on average every 6^{th} instruction is a branch or jump. This suggests that roughly 0.4% of the total 2.7% run-time overhead can be attributed to cache misses.

7.7.3 | Effectiveness of Memory Page Sharing

The *busybox* project was used to resemble a set basic programs one would typically find on a Linux machine. *busybox* incorporates 298 standard Linux commands. Those command line programs were started and their memory footprint was measured using `/proc/<PID>/maps`. On average, they mapped 14.9% more code pages than their unmodified original. Their data pages were unmodified. Only the RaTTle consumes additional memory (see Subsection 7.7.4). Compared to fine-grained memory randomisation solutions that impede code page sharing, Oxymoron on average saves about 85% of program memory.

7.7.4 | Memory and Instruction Overhead

Compared to a traditional program, the introduction of PALACE code replaced control flow branches with other, `%fs`-relative, instructions. For all SPEC2006 benchmark executables, on average 9% (± 1.7%) of all instructions are calls that needed to be replaced by indirections through the RaTTle. GOT indirect calls through the RaTTle account for only 0.03% of all instructions.

Additionally, a PALACE binary executable file is slightly larger than a traditional executable file because each code page (4 KiB) is a separate ASLR-enabled section in the executable file.

During run-time, the memory footprint also slightly increases because the RaTTle has to be kept in memory. Of course, this run-time memory usage is accompanied with the achieved goal of memory savings due to the sharing of code pages with other processes.

Encapsulating each memory page in a separate segment in the ELF file requires the allocation of one section header and one program header per page. A section header is 40 bytes and the ELF program header is 32 bytes which leads to an overhead of 72 bytes per 4096 byte memory page, or $\approx 1.76\%$. Figure 7.9 depicts both the increase of instructions due the static translation as well as the increase of the ELF section and program headers.

Run-Time. The size of the RaTTle depends on how many references the code has. If a target is referenced more than once, e.g. the GOT, only one index is saved in the RaTTle. For all files that belong to the SPECint CPU2006 benchmark, on average 19% of the code segment had to be added in the form of a RaTTle.

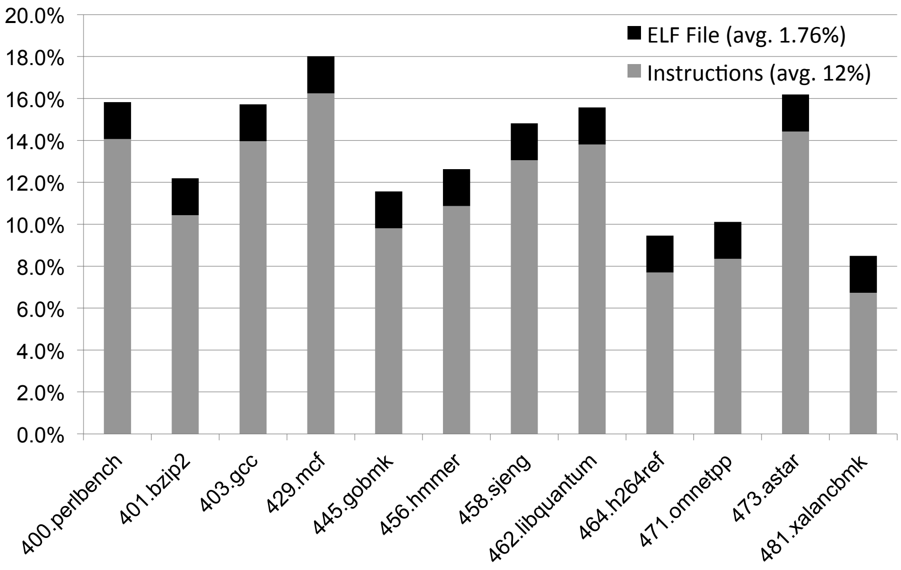


Figure 7.9: Memory overhead after static translation.

7.8 | Oxymoron Conclusion and Limitations

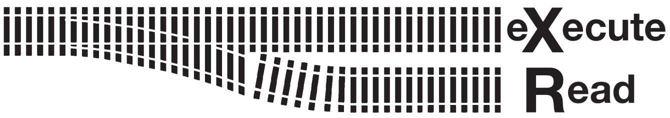
The used PALACE code only relies on segmentation as an additional hardware feature. Hence, Oxymoron also works in virtualised environments. Oxymoron was successfully tested in software (Qemu [L19]) and hardware (VirtualBox [L20] with Intel VT-X) virtual machines as well as on a para-virtualised Linux using the Xen hypervisor [L4].

The solution presented herein was implemented for the 32 bit x86 architecture. While its 64 bit successor has limited supported for segmentation, the necessary offset functionality of `%fs` segment registers is still available. However, in 64 bit mode, segmentation support is restricted and segment limits are not longer enabled by default. Newer AMD64 processors do support segment limit checking if the model specific register `Long Mode Segment Limit Enable (LMSLE)` is set to 1 [D1, Sec. 4.12.2]. However, the limit is only checked through `FS` segment register and only within the classical 0-4 GiB address space.

JIT. Just-in-time (JIT) compiled code, such as the Java run-time environment, is currently not protected as it is emitted during run-time. However, JIT-compilers can be adapted in order to emit PALACE-enabled code to benefit from the protections presented in this thesis.

While Oxymoron helps against code-reuse exploits in the light of leaked pointers, a far more dangerous vulnerability is a memory disclosure vulnerability. In contrast to a leaked pointer, a memory disclosure vulnerability allows an attacker to read content of the process' memory at her discretion. Thus, she can disassemble a known address and thereby reveal code addresses of fine-grained randomisation recursively. The next chapter presents a solution against that problem by mitigating the root cause, *memory disclosure vulnerabilities that read their own code*.

8 Execute-no-Read (XnR)



Fine-grained randomisation, as presented in chapter 6, helps against vulnerabilities that arise from the fact that a single leaked pointer might be all it takes to revert a whole address space layout. However, Snow et. al [P13] have shown how to exploit information leakage vulnerabilities to piecemeal discover even fine-grained randomisation inside a running process. These information leakages might originate from a buffer overflow that allows the attacker to control a read pointer which is later dereferenced, thereby revealing the contents of RAM at an attacker-chosen address. With this primitive at hand, an attacker can disassemble a known, valid address and thereby follow the chain of control flow to ultimately reveal almost all code addresses without tripping a page fault.

While it is hard, or even impossible, to statically analyse code for memory disclosure vulnerabilities, this chapter follows another approach: In the cain of *Fault* $\rightarrow$ *Error* $\rightarrow$ *Failure* (see Figure 2.1 in chapter 2), the approach presented in this chapter lets the Fault happen, i.e. the code itself is not protected. However, its ramifications in the sense of unwanted code reads are detected and stopped. This has the advantage that no matter how an attacker achieved to exploit a memory disclosure vulnerability, its effects are detected and prohibited before they can be exploited. This method fundamentally thwarts the root cause of those memory disclosure exploits by preventing the inadvertent reading of code while the code itself can still be executed. The newly proposed underlying primitive is called *Execute-no-Read* (XnR), which ensures that code can still be executed by the processor, but at the same time code cannot be read as data. This ultimately forfeits the self-disassembly which is necessary for *just-in-time code reuse attacks* (JIT-ROP) to work. Because contemporary x86 hardware does not support the primitives needed for XnR, it has been implemented in software by modifying the Linux kernel. Nevertheless, it has a performance overhead of only 2.2% for the Linux implementation and 3.4% for the Windows implementation.

Traditional ROP that uses a leaked pointer to circumvent ASLR is not prevented by XnR. This means that fine-grained ASLR, e.g. as described in chapter 6, or ROP gadget elimination must be in effect to ensure a holistic defence against code reuse attacks.

8.1 | The Contribution to Science and My Part in it

This chapter systematically studies the root causes behind disclosure vulnerabilities. The insight is that current processors only allow memory to be marked as non-writable or executable. However, code that is supposed to be executed must remain readable in memory and hence poses a risk for disclosure attacks.

The "Execute-no-Read" (XnR) primitive maintains the ability to execute code but prevents reading code as data, which is necessary to disassemble code and finally find ROP gadgets (especially when they are constructed on-the-fly). The XnR prototype implementation in software is a kernel-level modification for Linux and Windows. These hardware emulations are achieved by patching the memory management system in order to detect inadvertent reads of executable memory.

I have developed the design of the necessary kernel modifications to make XnR possible as well as implemented XnR for Linux.

8.2 | Attacker Model

An attacker may exploit a disclosure vulnerability to read the address space during run-time. The repeated application of the found vulnerability enables her to map portions of a process' address space with the objective of reusing the so-discovered code in a malicious way. This enables her to disassemble a running process with the intent of finding ROP gadgets.

8.2.1 | Assumptions

It is assumed that a vulnerable user mode process exists, which exhibits a memory disclosure vulnerability. The code is split, as usual, into executable code and data, while the code is randomised using fine-grained randomisation. The attacker knows the process' binary executable and the OS version of the victim system and can hence precompute potential gadget chains in advance using a local copy that features a different memory layout. The fine-grained randomisation of the victim process is unknown to the attacker. It is assumed that the process has at least one memory disclosure vulnerability, which makes the process read from an arbitrary memory location chosen by the attacker and report the value at that location. The attacker can exploit this to discover memory contents at chosen addresses. The ultimate goal is to know where all gadgets reside in memory. The attacker can also exploit an assumed control flow vulnerability to divert the control flow and execute arbitrary code of her choosing.

The attacker can also control the input of all communication channels to the process, especially including file content, network traffic, and data entered over the user interface. However, the attacker has not gained prior access to the operating system's kernel and the program's binary is not modified. Apart from that, the computational power of the attacker is unlimited. In particular, she can memorise disclosed memory, disassemble it, search it for gadgets, and find meaningful chains of those gadgets.

In summary, an attacker tries to exploit a process to reach arbitrary code execution.

Given this attacker model, the aim is to prevent JIT-ROP attacks. If the memory disclosure vulnerability can no longer be exploited to dynamically discover gadget chains, the execution cannot be diverted to a known, valid address. Before elaborating on the details of XnR, it should be stressed that other means to prevent arbitrary code execution must be in place in addition to XnR.

8.2.2 | Additional Requirements

While XnR is a powerful primitive, it is expected to be used in conjunction with two other security mechanisms:

1. $W \oplus X$ (Non-Executable Data)
2. Fine-grained randomisation (e.g. from chapter 6)

Non-Executable Data. The first requirement, $W \oplus X$, states that memory cannot be writeable and executable at the same time. This prevents an attacker from modifying existing code to suit her needs, or writing data and executing it as shellcode afterwards. On modern operating systems such as Linux, Mac OS, or Windows, the enforcement of $W \oplus X$ is a standard precaution and each process can decide to enable it. On Windows $W \oplus X$ is dubbed *Data Execution Prevention* [D12] and also must be requested by the application. However, Microsoft has released a tool called EMET (Enhanced Mitigation Experience Toolkit [L21]) that can be used to fore-enable DEP on processes that were either created before DEP existed or that do not request DEP for some other reason.

When the $W \oplus X$ primitive is enforced, an attacker has to take advantage of code that is already available in memory. In order to be able to make use of existing code, the attacker has to know the addresses of useful gadgets.

Fine-Grained Randomisation. Fine-grained randomisation is required so that an attacker cannot simply used a leaked pointer to calculate the addresses of useful gadgets. Then, the attacker is forced to discover the current address of gadgets by reading process memory, or effectively by disassembling process memory. Here, it is assumed that fine-grained randomisation as described in chapter 6 is used in conjunction with XnR.

8.3 | Design

XnR is aimed at preventing JIT-ROP attacks by eliminating its root cause, namely by detecting and preventing the underlying exploited disclosure vulnerability. More specifically, as soon as a process tries to read its own code as data, XnR considers this illegal behaviour. This prevents the first necessary step of a disclosure vulnerability. The implementation of XnR demonstrates that this new primitive can be enforced with a reasonable overhead on contemporary computer systems.

Von Neumann Architecture. Since contemporary processors all feature a von-Neumann memory architecture that mixes code and data, determining whether a particular piece of memory contains code is challenging. For XnR to work, different types of memory accesses need to be distinguished. From a CPU's perspective, this distinction is easy to make since different parts are responsible for these different actions. For the processor to do something useful, code must be executed, which then tells the processor what to do.

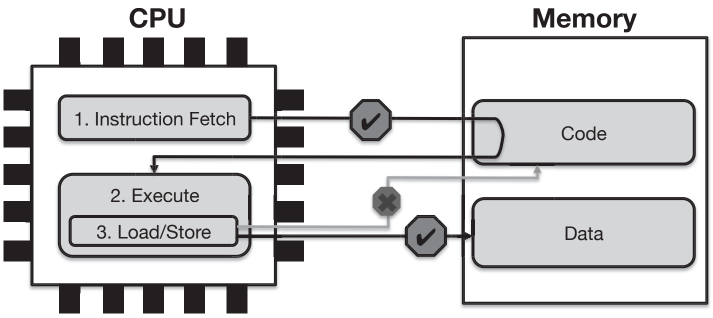


Figure 8.1: The XnR primitive distinguishes between legitimate code execution (instruction fetch) and illegal access to code using load/store instructions.

Figure 8.1 shows an abstract way of how a processor decodes and executes instructions. Before an instruction can be executed, the processor must first read the bytes that constitute the instruction from memory. This so-called *instruction fetch* resembles implicit access to memory. The processor stores the next instruction to be executed in the `RIP` (formerly `EIP`) register on x86. If the memory can deliver the requested data stored at `RIP`, i.e. the memory is valid, the processor tries to interpret that data as instructions. In the second phase, *Execution*, the according action of the interpreted instruction is performed, e.g. adding two register values. An instruction being executed by the processor might also *explicitly* access memory by means of *load/store* operations, i.e. reading memory to a processor register or storing a processor register value in memory. The effected address should typically contain data, otherwise code would be read or written. This is exactly what XnR detects: reading memory locations using a *Load* operation that actually contains code. Usually, programs do not read their own instructions as data.

This leaves us with three types of memory access that need to be distinguished for XnR:

Instruction Fetch: The processor fetches a few bytes from memory in order to decode and execute the instruction that it resembles.

This constitutes a legal operation that takes place during code execution.

Load/Store of Data: An instruction may access memory that either contains code or data. The load/store targets *data* if the address that the instruction referred to resides inside an area that contains data.

This constitutes a legal operation that is necessary to operate on data.

Load from Code: However, if a load instruction refers to an area of memory containing code, the program tries to read from itself.

This constitutes a programmatic disassembly that we consider illegal.

However, the distinction between load/store of data and load/store of code is not currently possible to do in hardware.

Hardware Limitations. The memory management unit (MMU) present in virtually all modern processors such as x86 and ARM introduced the notion of a *process*, which is a complete address space that exists from each process' point of view. Each such address space can have memory regions marked as writable and others as read-only. While the MMU can detect write attempts to any part of a process' memory, detecting read attempts is not supported. Read attempts can only be detected by declaring a certain memory region to be *non-present* in the MMU. However, a *non-present* memory region cannot be executed anymore. Moreover, the concept of non-readable, but executable, memory does not exist: memory permissions only allow to toggle the ability to write to memory or the ability to execute memory, where executable permissions imply read permissions. As a result, XnR cannot be implemented with current hardware.

Emulating XnR in Software & Design Decisions. To counteract the fact that hardware cannot distinguish between code and data reads, this XnR implementation is purely in software.

This has been achieved by extending the memory management system of Linux. Page faults are used to trap access to memory pages and make informed decisions about whether an XnR was detected and must be prevented. This decision to use page fault is deliberate as other solutions, foremost a split TLB, have unwanted drawbacks. Sherri Sparks and Jamie Butler, the authors of *Shadow Walker* [P84], introduced a similar technique with completely different intention. Sparks and Butler demonstrated how making memory containing a rootkit non-readable but executable a perfect hideout. They exploited the fact modern CPUs have different TLBs for code and data. By bringing them out of sync, executable code can reside in the code TLB while trying to read from the exact same memory location as data (e.g. a rootkit scanner) yields different memory. However, implementing a split TLB would merely rely on hardware side effect, which are different for AMD and Intel processors – and even between different revisions of the same brand processor. Implementing XnR with a split TLB technique would considerably increase the effort to apply the XnR technique in practice. Furthermore, the split TLB technique is more suited for smaller code as the TLBs have to be constantly kept out of sync. In contrast, the chosen method of using the well-defined page fault handler is also possible in virtual environments.

This page fault based XnR software emulation was realised by extending the memory management system of Linux. My co-authors Benjamin Kollenda, Jannik Pewny and Thorsten Holz did the analogous implementation for Windows by modifying Windows' memory management.

Both implementations make use of the so-called *page fault handler*. Every time the MMU detects a memory access violation in a process, the page fault handler of the operating system kernel is called. An access violation may occur when a process tried to read memory that is marked as *non-present* or when a process tried to write to memory that is marked as read-only. The granularity at which memory regions can have *writable* and *present* attributes is defined by the hardware as so-called *memory pages* (usually 4 KiB). When a page fault occurs, the process is halted and control is transferred to the kernel, which then tries to handle the page fault. A page fault is no exception in the ordinary run of a program but happens thousands of times during normal execution. The reason is *demand paging*, a performance feature that starts every process with an empty address space and only maps pages that are actually accessed. Demand paging is the underlying feature that XnR build upon since every first access to a page is caught due to the initially empty process space.

XnR Logic. The modified page fault handler (see Figure 8.2) checks the violation conditions and decides whether to continue normally (i.e. to map the missing page into the address space) or to terminate execution if a memory disclosure was detected. Each page fault is provided with additional information such as the address where the fault occurred and whether the access was generated during an instruction fetch. The latter is crucial information for XnR: If the CPU was trying to execute an instruction in a memory page that was *non-present*, this constitutes a legitimate operation and the usual *demand paging* routine of the kernel continues. It fetches the page and then marks it as *present*. If, on the other hand, the access violation did not occur due to an instruction fetch, then the processor was trying to read memory as data. In this case, XnR has to distinguish whether the accessed address is indeed inside a region containing data or if it tried to read from code. If the address resides inside a data region of the process, the modified page fault handler continues normally by mapping the missing page. Otherwise, the process tried to read from a code region, which is illegal, since the XnR rule defines executable code not to be readable. In this case, the faulting process is terminated with an error and any possible attack has been successfully prevented.

Distinguishing Code and Data. Distinguishing between data and code regions in a process is possible because the executable file formats provide information as to which memory region is executable (code) and which is readable (data). Both file formats, Linux' ELF and Windows' PE, incorporate attributes for each loadable segment that will be mapped into memory. When a process is created from an executable file or when an additional shared library is loaded into an existing address space, the occupied memory region is tagged with the type attributes copied over from the executable file. This way, when a page fault occurs, the faulting address must lie inside any of the mapped regions, which in turn has associated memory type information (code/data). This procedure is displayed in Figure 8.3.

Continuously Trapping Access. Without any additional trickery, the procedure described above would only trap on every first access to a memory page because it will be present in the address space afterwards. Therefore, pages need to be marked *non-present* again after execution in a particular page has finished. However, the halting problem dictates that it is not generally possible to decide when and if a program finished executing a particular memory region. A much simpler solution is to mark the first page that is accessed present and leave it marked *present* until execution moves to another page. This way, it is obvious that access continues somewhere else – at least for single-threaded

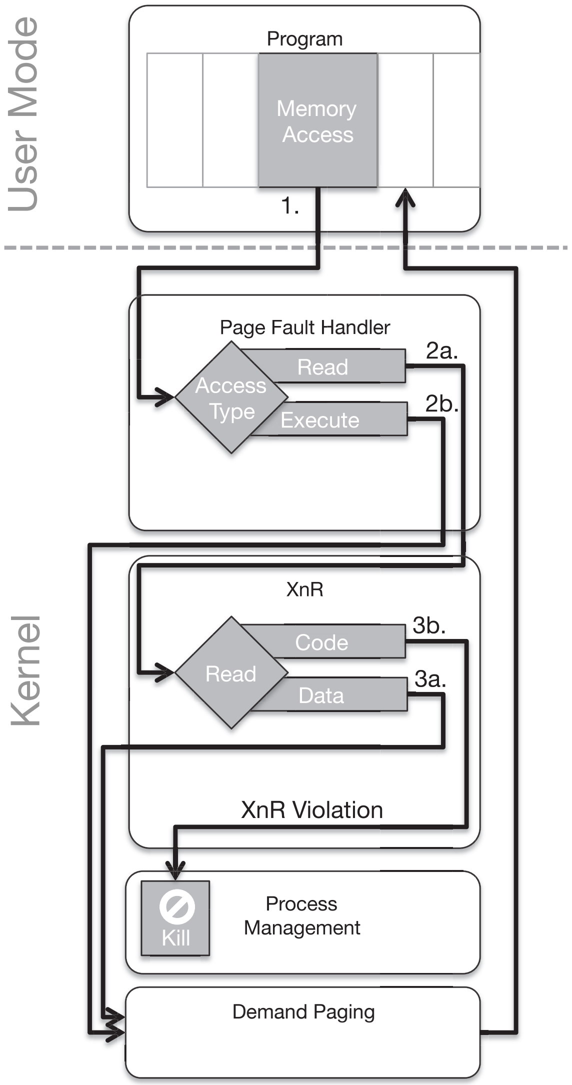


Figure 8.2: Flow diagram of the XnR logic

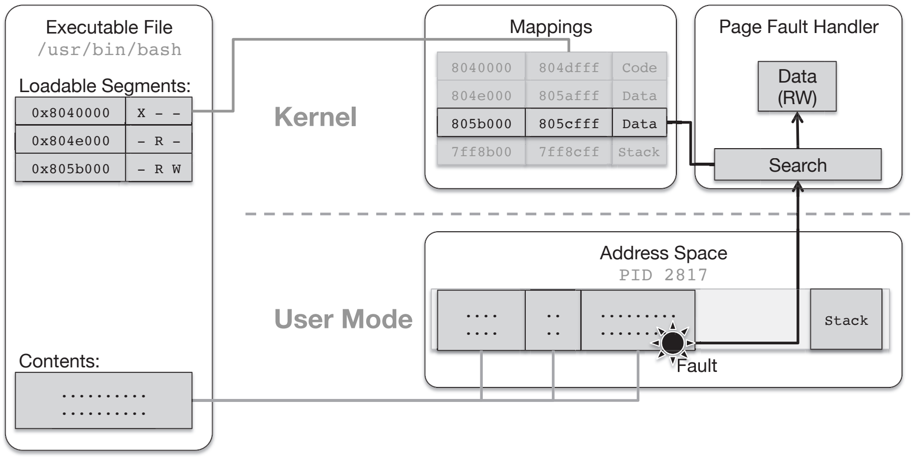


Figure 8.3: How a loadable segment from an executable file is mapped into memory and tagged accordingly

processes. The previously accessed page can then be deprived of its *present* bit, which will trigger future page faults on that page when accessed. Whenever execution runs outside of one memory page into another, the last page gets inaccessible while the new page is set to *present* in the same atomic operation. This method guarantees that an already accessed page will trigger new page faults as long as at least one other page has been accessed in between.

Performance Considerations. As read/write access to data regions is always deemed accepted program behaviour, data regions are exempt from the XnR logic of hiding memory pages. Consequently, access to data pages does not trigger a page fault and so no performance overhead at all occurs. The performance penalty (see section 8.5) only occurs when accessing a code page, i.e. when execution continues in another memory page or when code is attempted to be read as data.

Normally, programs are optimised for high locality to make most use of comparatively small cache sizes. This also means that execution is likely to execute a few hundred instructions before continuing in another memory page. This behaviour is advantageous to XnR, as it means it will not trigger too many page faults during execution. However, calling sub-routines in libraries causes a program to continue execution in another memory page. Even worse, concurrent execution results in several threads competing for the single active memory page. In terms of performance, such behaviour would practically be worse than single-threaded execution.

To counteract this, XnR keeps more than one page present at all times. In fact, the last recently used n pages are marked *present* while all the others remain flagged *non-present* to form a sliding window (see Figure 8.4). This allows the kernel to keep more than one page active at the same time, which reduces the chance of a congestion.

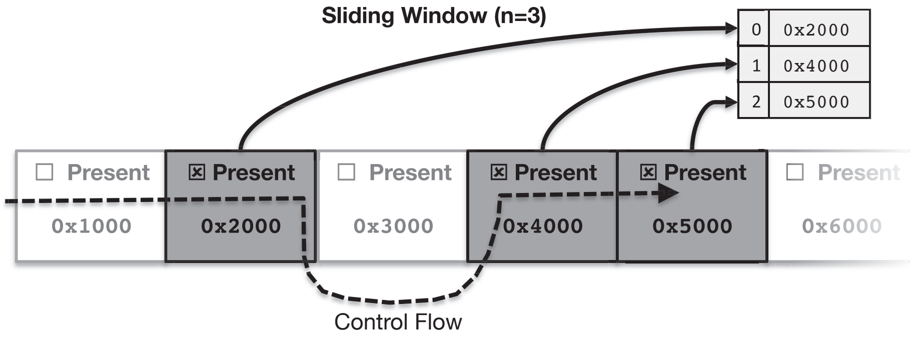


Figure 8.4: To improve performance, the parameter n adjusts the window length of the sliding window that keeps up to n pages in the address space. At the same time n is the security parameter.

Security Considerations. The parameter n is at the same time a security parameter as it correlates to the likelihood of trapping on an illegal code read access. It constitutes a trade-off between performance and security.

For $n = 1$, only the page in which execution currently takes place is mapped at any time. This is the only page that does not trigger a page fault when being accessed and hence is the only page that could potentially be read by an attacker without being noticed by XnR. Since the attacker can only read the page in which execution takes place, she has to know the current execution address (instruction pointer) at the time of the exploit. This is usually not possible due to the assumption of fine-grained randomisation – or at least ASLR. Even if it were possible to guess the execution address correctly and the attacker exploits the disclosure vulnerability, she would only have access to the single page in which the code for the disclosure vulnerability resides because accessing another position of code would trigger an XnR exception.

For $n > 1$ mapped pages, the situation is more complex as there are now n pages which can be accessed without triggering a page fault. If the attacker can determine the address of one of those pages at run-time, she can read that page and search it for gadgets. However, the set of visible n pages depends on the past control-flow. Hence, each memory disclosure can only see exactly the same n pages, namely those whose control flow lead to the memory disclosure vulnerability. If any other page, which is not in the set of n pages is accessed, XnR will detect it. If n is equal to the total number of code pages of a process, then all pages stay mapped in the process' address space forever. This would resemble XnR not being active at all.

8.4 | Implementation

As mentioned before, XnR is not yet available as a hardware primitive because hitherto, it did not make much sense to have a portion of memory executable but not be able to read from it. The software emulation, which is a kernel modification, was implemented for the Linux kernel and Windows 8 kernel – both in their x86-64 version. In theory, the user mode programs to be protected do not need any modifications at all. In practice, some of the binaries need to be re-linked though (see subsection 8.5.2).

While the Linux and Windows implementations share the same concept, their specific implementations differ vastly due to the different natures of Windows and Linux. For both kernels, profound memory management functionality needed to be changed. The kernels differ in the respect that Windows internals, such as memory management, cannot easily be modified. This naturally led to different approaches with respect to *how* and *where* XnR was engaged in the specific kernels. In general, the fact that the Linux kernel is provided in source code allows for a proper integration with the existing memory management and process run-time system, whereas Windows prohibits any modifications in the first place and hence forces the Windows implementation of XnR to be an outer shell around immutable Windows kernel functions.

Since I implemented the Linux kernel modifications, this is the main focus of this section. The Windows implementation details are elaborated on in the published CCS paper [M4].

Trap Distinction. The goal of the XnR Linux kernel modifications is to intercept any access to code or data, and to then decide whether that access was triggered due to an *instruction fetch* (i.e. the processor is executing code) or due to *data access* (i.e. load and store operations on data).

XnR uses the already existing memory management unit (MMU) of modern processors to efficiently intercept each access to memory in software. The MMU implements virtual memory and thereby enables process isolation. The operating system, in concert with the MMU, allows for the illusion of a contiguous virtual address space for every process. Only the used parts of each address space (i.e. each process) are actually mapped to physical memory, which is done completely transparent to each process. The MMU divides memory in the smallest addressable unit, a memory page, for which a translation from virtual memory address to physical memory pages can be set up for each page and for each process. The MMU also allows trapping access to a certain memory page. In particular, writing to read-only portions of memory will trigger a page fault – so will attempting to read from an address that is not mapped, i.e. non-existent.

This trapping mechanism is also used for purposes other than detecting illegal access. In fact, many modern memory optimisations use page faults as an underlying feature. For example, variable stack sizes and lazy loading. The stack can grow dynamically and only consumes as much memory as is actually used. Every time the stack grows into a non-existent memory page, the kernel jumps in and allocates a new, empty page in place where the fault occurred. This gives each process the illusion of an infinitely large stack. Also, so-called *demand paging* saves processing time and memory by loading parts of memory from disk only when they are actually accessed. The same holds for memory allocated using `malloc()` (internally `brk()` syscall). The memory is not actually mapped into the address space until it is accessed.

Because the *demand paging* already facilities a framework to detect access to memory, it was a suitable position to entrench the XnR logic into the Linux kernel. The advantage of an implementation in the *demand paging* subsystem of Linux is that an illegal access can be detected before it can actually happen, i.e. before the targeted code is accessible by the user mode process. The implementation was based on the then current Linux kernel version 3.13.7 for 64-bit x86 CPUs. The Linux memory management is fairly sophisticated and uses page faults not only to detect illegal access to memory but to transparently implement *demand paging*, Copy-on-Write (COW), and to map files to memory.

A general overview of how XnR is integrated in the Linux kernel is given in Figure 8.5. A typical XnR check works as follows:

For every page fault, Linux first checks

a) if the fault is due to access to invalid memory or

b) if the fault can be gracefully resolved by mapping a new page of memory.

In case a), the accessed position is not supposed to contain memory and was spuriously accessed by the program. Consequently, it is killed as any further access on invalid memory would be undefined behaviour by definition. The other case b), means the program has accessed logically valid memory, but it has not been loaded yet into the address space yet. *Demand paging* is invoked to pretend the accessed page existed in the first place. After it has been loaded, execution continues as if nothing happened. This way, the address space of

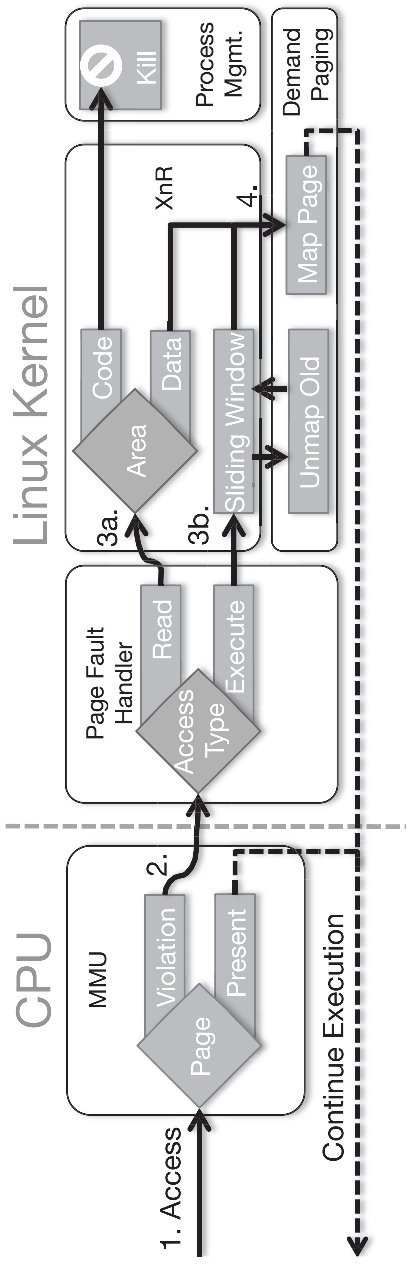
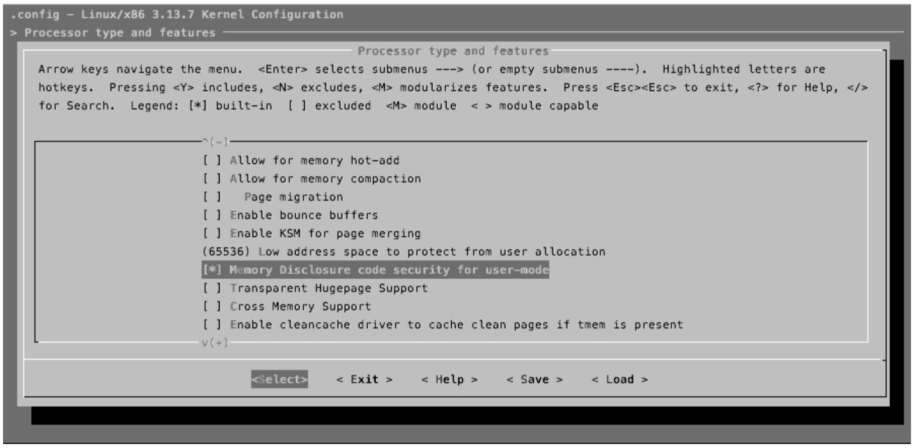


Figure 8.5: Flow diagram of how the CPU, MMU and parts of the Linux kernel interact in order to implement XnR.

a process can be built on demand, rather than wasting memory and time by pre-loading the entire address space at program start.

Integration nto *demand paging*. The Linux implementation is designed as a patch against the 3.13.7 kernel and works with 64 bit and 32 bit programs running on an x86-64 kernel. The patch modifies the existing minor page fault handler as well the file mapping part of *demand paging* and adds the XnR-sub-subsystem to the memory management subsystem of Linux. The entire XnR logic was implemented in a separate new Linux kernel module that can be switched on and off. The entire XnR patch is made available through an additional kernel `CONFIG` parameter called `CONFIG_EXECUTE_NO_READ`. The common `make menuconfig` command can be used to configure the kernel with XnR support (see Figure 8.6).



```
.config - Linux/x86 3.13.7 Kernel Configuration
> Processor type and features
                          Processor type and features
 Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are
 hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
 for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                      ^(-)
                      [ ] Allow for memory hot-add
                      [ ] Allow for memory compaction
                      [ ]   Page migration
                      [ ] Enable bounce buffers
                      [ ] Enable KSM for page merging
                      (65536) Low address space to protect from user allocation
                      [*] Memory Disclosure code security for user-mode
                      [ ] Transparent Hugepage Support
                      [ ] Cross Memory Support
                      [ ] Enable cleancache driver to cache clean pages if tmem is present
                      v(+)

                 <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 8.6: Screenshot of an XnR-enabled kernel configuration

Due to the architecture of Linux, the entire patch is spread across 13 files and the patch modified 570 lines of code.

```
arch/x86/include/asm/pf_errorcodes.h
arch/x86/mm/fault.c
include/linux/mm.h
include/linux/mm_types.h
include/linux/XnR.h
include/linux/XnR_types.h
kernel/fork.c
mm/filemap.c
mm/init-mm.c
mm/Kconfig
mm/Makefile
mm/memory.c
mm/XnR.c
```

`mm/Makefile, Kconfig` describe the patch itself, so that it can be integrated into the kernel as a module, which can be selected by the user.

`arch/x86/mm/fault.c` This file handles most of the *demand paging* and then transfers control to file mapping (`filemap.c`) if a page has not been loaded yet. Before the actual *demand paging* procedure is invoked, the page fault status has to be checked for *instruction fetch* operation. The x86 CPU pushes a `word` to the stack that encodes the type of access violation (e.g. read, write, user, kernel, instruction fetch, data access). If an instruction fetch happened, this is harmless. Nevertheless, the bookkeeping subroutines have to be called to save which page will be allowed to execute in order to keep track of the sliding window. Then, the *demand paging* logic does its regular job of mapping the particular page that was not present. If, on the other hand, the target address points to data, it must be checked to which logical area it belongs. If the faulting address lies in a segment that is marked executable but not readable, the process tried to read instructions from memory.

`mm/filemap.c` handles generic file systems whose files have been mapped into memory. This is the case for executable files and shared libraries. Parts of their content are mapped into different virtual address spaces.

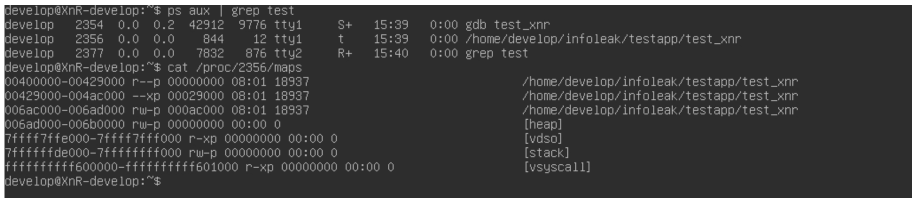
`mm/init-mm.c` is responsible for initialising a new internal memory structure that keeps track of all allocations of a process. The sliding window needs to be anchored here in order to be part of a process' memory allocation structure.

`kernel/fork.c` implements `fork()`ing new processes, which effectively constitutes cloning an existing process. Cloning is easy and inexpensive in terms in memory and processing power, as the CPU uses virtual memory mapping and hence two identical processes simply use the same mapping.

`mm/memory.c` implements the low level memory page handling such as mapping pages, unmapping pages, and traversing page table structures. Most importantly, a majority of the functions contained in `memory.c` can be called in an atomic way and these low-level functions take care of multiprocessor locks and accessing hardware page table information in a safe manner. The file also contains functions that retrieve the corresponding memory management structures given a faulting address. The functions that correlate a triggered page fault with internal memory management data structures have been patched to check if XnR is enabled and call the sliding window mechanism accordingly.

`mm/XnR.c` implements most of the XnR helper functions, such as the actual sliding window implementation, book-keeping and statistics. The sliding window checks are called for every page fault from `fault.c` and `memory.c`. The sliding window then ensures that pages that have already been paged into the address space of a process (e.g. by *demand paging*) will be checked again when they are accessed again.

XnR Flag in Executable Files. In order for an executable file to signal that it wants to be protected by XnR, it does not need an additional flag. Linux executable files and shared libraries are usually provided in the ELF executable format [D8]. ELF supports specifying access permissions of a particular memory area. In unmodified executables and shared libraries, the code segment would usually be marked `RX`, so `R`eadable and e`X`ecutable. Code that shall be protected by XnR simply has to remove the `R` from the ELF header of the respective code section. The loaded executables and shared libraries then have their `X` bit set but lack the `R` bit as shown in the active memory mappings of an XnR-enabled process in Figure 8.7.



```
develop@XnR-develop:~$ ps aux | grep test
develop   2354  0.0  0.2  42912  9776 tty1     S+   15:39   0:00 gdb test_xnr
develop   2356  0.0  0.0    844    12 tty1     t    15:39   0:00 /home/develop/infoleak/testapp/test_xnr
develop   2377  0.0  0.0   7832   876 tty2     R+   15:40   0:00 grep test
develop@XnR-develop:~$ cat /proc/2356/maps
00400000-00429000 r--p 00000000 08:01 18937                              /home/develop/infoleak/testapp/test_xnr
00429000-004ac000 --xp 00029000 08:01 18937                              /home/develop/infoleak/testapp/test_xnr
006ac000-006ad000 rw-p 000ac000 08:01 18937                              /home/develop/infoleak/testapp/test_xnr
006ad000-006b0000 rw-p 00000000 00:00 0                                  [heap]
7ffff7ffe000-7ffff7fff000 r-xp 00000000 00:00 0                          [vdso]
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
develop@XnR-develop:~$
```

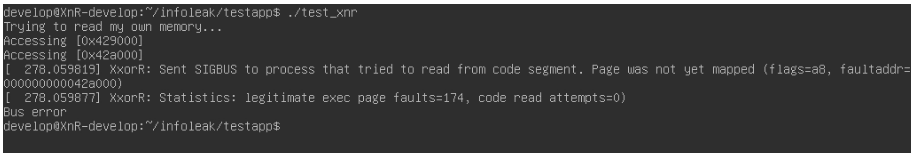
Figure 8.7: Screenshot of the loaded program segments of an XnR program

Test Case. A simple program that deliberately tries to read its own memory can be used to test the XnR functionality. A very basic test program is provided in Listing 8.1.

```c
#include <stdio.h>
extern void* wcschr;

#define PAGE_SIZE   4096
#define PAGE_BITS   (PAGE_SIZE-1)
#define ROUND_DOWN(x)    (x & ~PAGE_BITS)
#define ROUND_UP(x)      (ROUND_DOWN(x) + PAGE_SIZE)

int main(int argc, char* argv[])
{
    unsigned long address_start = (unsigned long)&main;
    unsigned long address_end = (unsigned long)&wcschr;

    printf ("Trying␣to␣read␣my␣own␣memory...\n");

    unsigned long something = 0;
    unsigned long* ptr;

    for(ptr = address_start;
          ptr < address_end;
          ptr += PAGE_SIZE/sizeof(unsigned long*) )
    {
            printf ("Accessing␣[%p]\n", ptr);
        // read it and store it into another variable so
        // it won't be optimised out
        something += *ptr;  // actual access
    }

    printf ("XnR␣did␣not␣trap␣if␣this␣is␣reached\n");
}
```

Listing 8.1: Tests for XnR functionality

The resulting compiled program then causes an XnR exception when run and is killed before the respective memory can be read (shown in Figure 8.8



```
develop@XnR-develop:~/infoleak/testapp$ ./test_xnr
Trying to read my own memory...
Accessing [0x429000]
Accessing [0x42a000]
[  278.059819] XxorR: Sent SIGBUS to process that tried to read from code segment. Page was not yet mapped (flags=a8, faultaddr=
000000000042a000)
[  278.059877] XxorR: Statistics: legitimate exec page faults=174, code read attempts=0)
Bus error
develop@XnR-develop:~/infoleak/testapp$
```

Figure 8.8: Screenshot of a program that violated XnR

8.5 | Evaluation

The effectiveness and performance of XnR has been evaluated on physical hardware as well as inside a virtual machine. The physical hardware tests were conducted on Intel Core i7 and AMD Phenom processors to ensure that no microarchitecture-specific subtleties are exploited. The use of a virtual machine emphasises that this implementation does not depend on hardware specific side effects such as cache or TLB layouts.

8.5.1 | Security Parameter n

The choice of n is roughly proportional to performance and inversely proportional to the security. For an attacker to be successful, she needs an exploitable memory disclosure vulnerability and a valid code address. This address can then be fed into the memory disclosure vulnerability to get memory content, which can then be disassembled in order to recursively uncover more valid addresses.

If the sliding window size n has been set to a single page, i.e. $n = 1$, then the attacker needs to know an address inside the currently active memory page. At the time of the exploit, the currently active memory page is the page storing the code that contains the memory disclosure vulnerability. Since the program memory is randomised, the attacker requires the current randomised position of code that contains the memory disclosure vulnerability. This could for example happen through a memory leak, e.g. a read buffer overrun into the stack where a valid return address is stored. However, this return address must at the same time be close to the memory disclosure vulnerability in order to reside in the only active page.

In summary, a memory leak must exist that reveals a valid address in the same memory page as the memory disclosure vulnerability and then the memory disclosure vulnerability must be exploited with the discovered address of its own code. Any other address would trigger an XnR exception. Likewise, reading across the page's boundary would trigger an XnR check as surrounding code pages are not mapped if $n = 1$. If $n > 1$, the remaining $n - 1$ pages are those pages where execution has been before. Hence, the only viable option to discover more content of a memory page through a memory disclosure exploit is by traversing control flow back through the return stack. However, this option is also not safe for the attacker as control flow might have diverted to other pages in between, which is not visible from inspecting the call stack. The attacker might have a local copy of the program, though. This enables her to find most likely access patterns for code pages that lead to the memory page, which contains the memory disclosure vulnerability. Using this guessing, she could determine the last used $n-1$ pages. However, due to randomisation their addresses are unknown to the attacker, unless they are also discoverable by a memory leak.

The aforementioned attack paths of course only apply to single-threaded execution. Otherwise, the $n-1$ other pages cannot be enumerated with high certainty as the order of execution in each thread greatly affects the set of active n pages. Ultimately, even if the attacker has found a way to get access to the single active page, which contains the disclosure vulnerability, a disassembly can at most reveal 4 KiB of instructions – the size of a single page. All usable gadgets must be found inside those 4 KiB.

8.5.2 | Precision and Effectiveness

The most important property of XnR is that it can fulfil its goal in terms of security, i.e. prevent illegitimate code reads. A desirable evaluation would need to show that any possible memory disclosure attack is successfully caught by XnR (i.e. no false negatives), while benign programs are not affected by the modifications (no false positives). As there is no enumeration of all possible memory disclosure exploits, it is impossible to test against successful prevention of all of them. Instead, this evaluation only tests a very generic but powerful memory disclosure to show that XnR can catch and prevent those illegitimate code reads. The other case, no false positives, is also impossible to show empirically. Instead, a typical large set of programs is tested for incompatibility (false positives) and the found problems are solved in a universal way.

Detection of Exploits. As an example exploit, the standard Linux `netcat` program was modified to contain a memory disclosure vulnerability. `netcat` is usually used to establish or listen on TCP/UDP network connections. In order to make the source vulnerable to a memory disclosure vulnerability, it was modified such that too long packets can trigger a buffer overflow which overwrites an internal buffer pointer. This buffer pointer is used to assemble network packets. This enables an attacker to craft malicious packets in such a way as to intentionally overwrite the buffer pointer and thereby directing the TCP response buffer to arbitrary memory. This resembles a memory disclosure vulnerability that can extract memory regions from an attacker-driven address.

When the modified version of `netcat` is run on an unprotected Linux, the memory disclosure vulnerability returns arbitrary memory containing data and code to the attacker. With enabled XnR protection, however, the offending `netcat` process is killed as soon as the memory disclosure vulnerability is directed to a region containing executable code. Since JIT-ROP relies on the necessary precondition of reading memory such that gadgets can be constructed on-the-fly, this behaviour is successfully detected and prevented by XnR.

Legitimate Code Reads. Since XnR prevents *all* read access to code, there must not be any legitimate code reads as XnR's behaviour would otherwise hinder normal execution. If a benign program reads its own code segment as data for a legitimate reason, blocking such access would constitute a false positive.

Even though it might be counter-intuitive that programs try to read their own code, our tests have shown that in fact a majority of tested Linux programs on Debian attempted to read code during normal program execution. In fact, all read attempts of different programs targeted the same object, the header of the ELF executable file or the header of an ELF shared library. Closer inspection revealed that this header is parsed by library functions that iterate over the loaded sections (PHDRS). In contemporary Linux ELF executables, this header resides in the `.text` segment (see Figure 8.9). Reading the header in memory triggers an XnR exception because it belongs to `.text`, which is marked executable. The fact that the ELF header resides in the loaded `.text` segment is a result of file size optimisations by the linker. The fact that all loadable segments have to be memory page-aligned also on disk, would waste a lot of space if the first loadable segment would start at offset 4,096 in the ELF file. Consequently, the first segment is designed such that it starts at 0, which by definition is a multiple of 4,096. However, this results in the inclusion of the ELF header in the first loadable segment. Strictly speaking, this default behaviour of all modern Linux programs is semantically wrong because the header (data) is not supposed to be executable. Should the program have a vulnerability, the ELF header unnecessarily resembles ROP gadgets.

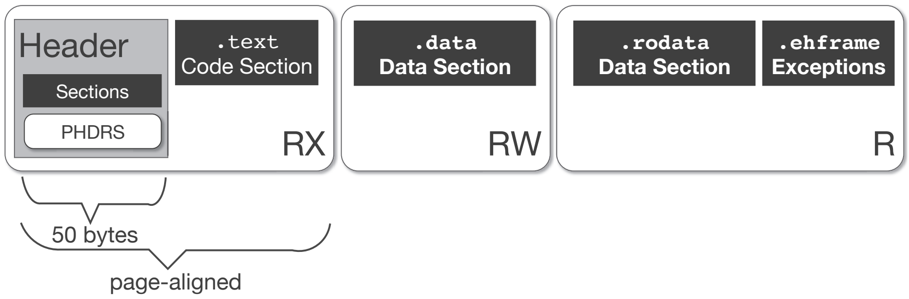


Figure 8.9: A typical ELF file layout: The first loadable program header (PHDR) spans across the ELF header in order to load it to memory as well. This results in an executable ELF header.

Instead, the ELF header should either be in its own PHDR loadable section or simply be part of read-only data. Moving the header to the read-only data section is not an option because the header must start at the first byte. Another viable option, and the solution here, is to swap the PHDR loadable sections such that read-only data is always the first segment in the ELF file. This does not change where the segments will be loaded in memory. However, it ensures that the ELF header is stored inside the read-only loadable segment. This solves two problems: The header no longer resembles ROP gadgets, and more importantly, reading the header in memory does not trigger an XnR exception, as it is no longer stored inside `.text`.

Linker Script Solution. The root cause of this default behaviour lies in the way the commonly used Linux Linker `ld` arranges the loadable segments by default. The `ld` linker is part of the `binutils` package and is used by `gcc` [L22] and `llvm` [L7] on many Linux distributions. Both, `gcc` and `llvm`, also work with the alternative *Gold Linker* (`ld.gold`) that has initially been developed by Google and is nowadays part of the official GNU `binutils` source.

The suggested solution for the ELF header problem is to use a Linker Script. Linker Scripts are usually implicitly used and are specified when `binutils` are compiled. However, it is possible to specify an alternative linker script with the `-T` option of `ld`. The changes the linker script can be narrowed down to changing the order of the default PHDRS and its memory access flags. The latter removes the readable flag from the code loadable segment and thereby enables XnR in the kernel. Listing 8.2 shows the important PHDRS parts of the modified linker script for x86-64 ELF64 executables and shared libraries.

```
/* Default linker script for XnR executables */
OUTPUT_FORMAT("elf64-x86-64", "elf64-x86-64",
          "elf64-x86-64")
OUTPUT_ARCH(i386:x86-64)
ENTRY(_start)

PHDRS
{
          rodata  PT_LOAD FLAGS(4) ;       /* read */
          text    PT_LOAD FLAGS(1) ;       /* execute only */
          data    PT_LOAD FLAGS(6) ;       /* read + write */
          tls     PT_TLS;
          stack   PT_GNU_STACK;
}
```

Listing 8.2: Modified x86-64 linker script with read flag removed from code and order of PHDRS changed in order not to include the ELF header in the code segment.

After modifying the standard Linux linker script that creates executables and shared libraries, the ELF headers reside in the read-only data section (`.rodata`) and can be accessed without triggering an XnR violation. As a side effect, this prevents the ELF headers from being executable.

Unaffected Execution. Testing the modified linker script with 352 standard command line programs for Linux additionally gave good confidence that

a) the modified linker script is compatible with those programs

b) no XnR exceptions are caused during normal program execution.

The 352 programs used stem from the *busybox* [L23] project, which incorporates the most helpful Linux commands to install as a single package. The test for unaffected execution was conducted as follows:

1. Call each of the 352 programs without any arguments and re-route the program's output (`stdout` and `stderr`) to a file. Additionally, each program's exit code was routed to a different file. This constitutes the expected behaviour for each program later in the test.
2. After XnR was enabled, the programs are expected to produce exactly the same output (`stdout` and `stderr`) and exactly the same exit code. All programs were run again with XnR enabled and with their outputs and exit codes stored in files.
3. The output files (`stdout`, `stderr` and exit code) were compared for each of the 352 programs. There was not a single difference.

This is a very good indication that XnR does not affect the normal execution of programs.

8.5.3 | Performance Evaluation

XnR's performance was evaluated using the de facto standard SPEC CPU2006 integer benchmark suite. SPEC measures the entire system's performance for their 12 typical benchmark workloads. To compare performance, all benchmarks were run on an unmodified kernel 3.13.7 to set the baseline. Then, the same benchmark executables were run with XnR enabled in the kernel. All benchmarks were performed on an Intel Core i7-3770 CPU running at 3.4 GHz with 4 GB of RAM. This particular CPU features four hardware cores with two symmetric hardware threads each (*HyperThreading*).

As the programs themselves were not modified, any performance overhead can only occur inside the modified page fault handler of the Linux kernel, as this is the only part that was changed. The nominal delay in each page fault that occurs is comparatively low. However, the implementation of the sliding window results in the number of page faults to increase drastically. While data page faults are not affected at all, the code page faults increase due to the fact the window mechanism ensures that at most n code pages are mapped into an XnR-enabled process at any time.

Sliding Window Impact. During a program's execution, its control flow takes place in different memory pages. While it is executing a loop, it is very likely to stay inside a single memory page and thus not trigger any page fault. However, for programs that heavily jump between different code positions, the probability is higher that they access different memory pages. Usually, compilers optimise code layout so that the locality of code is high, i.e. functions that call each other or functions that resemble hot spots reside next to each other in the compiled code. Additionally, non-control flow instructions are simply executed one after another, i.e. they reside in the same memory page or eventually cross a page boundary. If a program, however, continuously accesses more than n pages, this results in constant eviction of pages from the process' address space and results in a performance degradation compared to a stock Linux kernel without XnR checks.

As the window size n influences performance, the benchmarks are conducted in dependence of n. Figure 8.10 depicts the performance depending on the window size $n = 2$ pages, $n = 4$, $n = 6$ and $n = 8$ pages.

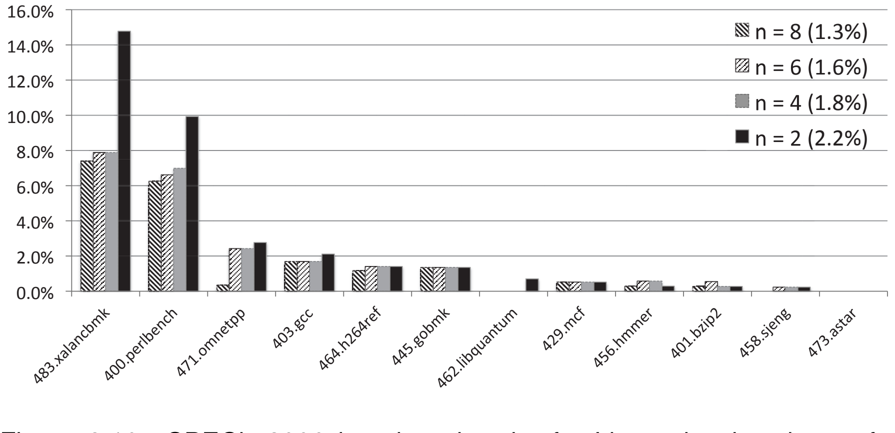


Figure 8.10: SPECint2006 benchmark suite for Linux showing the performance for each of the 12 benchmarks (and averages) dependent on the parameter n.

Even for a small window size of only $n = 2$ pages, the average overhead is a moderate 2.2%. These good performance figures make choosing the right

n fairly easy as small values of n allow for high security but remain decent performance.

Working Set and Cache Effects. The low overhead contradicts the traditional assumption that a large working set is necessary for good performance. The working set describes the set of pages that are actually used by a process [P85]. With *demand paging*, the working set should hence increase until all necessary pages have been touched once and remain in memory. However, XnR does not actually revert the work done by *demand paging*. The sliding window mechanism only marks the pages that fall out of the window as *not present*. This does not actually reduce the working set size. In contrast to an unmapped page, removing the *present* bit leaves the content of a memory page intact and also does not touch any caches. The effort of *demand paging*, namely fetching the contents of a page from disk, still only have to be done once and do not change with XnR. Instead, XnR only re-enables the *present* bit, which causes a TLB miss but at the same time profits from a cache that is still filled. Hence, the overhead is mainly due to the CPU switching to kernel mode after the hardware page fault and switching back to user mode after enabling the *present* bit.

Data vs. Code Access. Since XnR applies the sliding window technique to code pages only, the performance impact of XnR is different for data-intense applications vs. computationally intense applications.

An expected result is that the performance overhead is more distinct for applications that heavily jump between many code areas. To measure the different effects, benchmarks from SPEC were taken and their page faults inspected more closely.

- **Data-Intense** programs consume a lot of data, operate on it and either refine a result or produce a stream of output. Compression algorithms fulfil that description as they consume a lot of data, encode the data differently and hence produce almost as much output. Such a candidate is the *BZip2* compression, which is part of the SPEC benchmark anyway. SPEC's version of BZip2 (`401.bzip2`) is optimised to perform almost no I/O as SPEC is a CPU benchmark.
- **Computationally-Intense** programs crunch numbers and (repeatedly) execute a lot of code on comparatively little data. Examples of such are the *Perl* interpreter (benchmark `400.perlbench`), which runs the email indexer `MHonArc` and the anti-spam filter `SpamAssassin`.

The XnR page fault overhead for the chosen benchmarks `400.perlbench` and `402.bzip2` is shown in Table 8.1. The used window size was $n = 4$. As expected, the data-intense `402.bzip2` benchmark caused was not affected by the sliding window as much as `400.perlbench`. While the number of page

Program	∅ Page Fault Duration		Page Faults / s	
	Stock Kernel	With XnR	Stock Kernel	With XnR
`402.bzip2`	9.1 μs	12.9 μs	307	401
`400.perlbench`	5.4 μs	8.3 μs	108	512

Table 8.1: Microbenchmarks for data-intensive example (`bzip2`) and code-intensive example (`perl`). Sliding window set to $n = 4$.

faults only increased by one third for `402.bzip2`, `400.perlbench` suffered from more than three times as many page faults. This is to be expected as data pages are not evicted from the address space by the sliding window and the increase in page faults per second originate from the BZip2 algorithm not fitting within $n = 4$ pages. The difference in page fault duration for the stock Linux kernel stems from the fact that the different programs use the memory differently and hence the page fault handler takes different paths. The overall time spent in the page fault handler for each program is rather low. On average, `402.bzip2` spends only 0.28% of the entire benchmark duration (several minutes) in the page fault handler. With XnR enabled, `402.bzip2` spends only 0.52% in the page fault handler. The overall slowdown due to the page fault handler is thus a mere 0.24%. As expected, `400.perlbench` spends even less time in the page fault handler on a stock Linux kernel: 0.06%. However, with XnR enabled, Perl spends 0.42% in the page fault handler (an increase of 600%), but the overall slowdown due to the page fault handler is a mere 0.37%, as the time spent per page fault is significantly less than for BZip2.

The distribution of page fault durations for the two programs reveals even more insight, as shown in Figure 8.11. The area under the curve has increased for data-intense benchmark Bzip2 as the page faults take longer (shift towards right) but at the same time more page faults occur. The additional checks on every access result in a slightly broader distribution of page fault times. In contrast, Perl's few and short page faults have increased significantly with XnR enabled, but each does not take much longer.

The total overhead in terms of run-time for the same input is almost negligible for *BZip2* (only 0.3% for $n = 4$), whereas the total run-time given the identical input to *Perl* increased by 7.0%.

8.6 | Discussion

Our prototype implementations for Windows and Linux show the general feasibility of our approach. However, similar to $W \oplus X$, to become widespread and usable without restrictions, they need compiler support in the future. Before

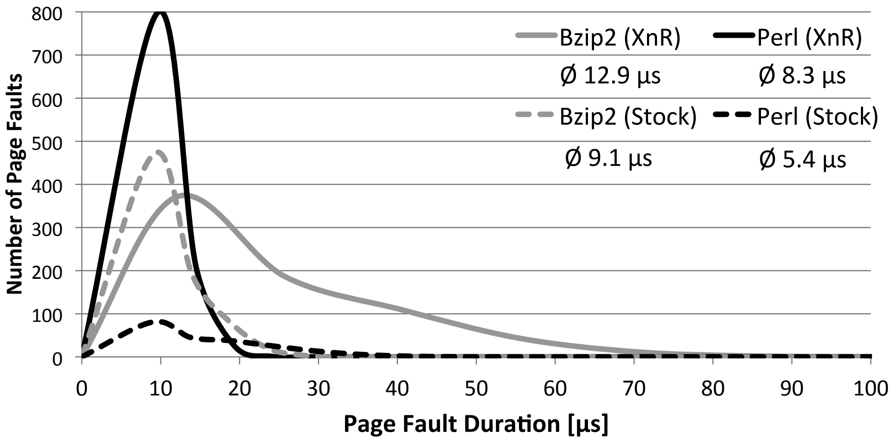


Figure 8.11: Distribution of the duration of page faults that have been triggered over a period of 2s.

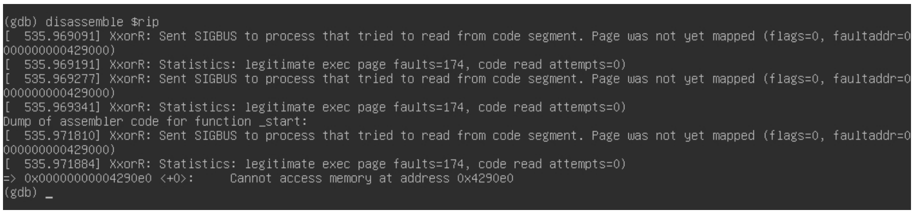
$W \oplus X$ was introduced, executing the stack was normal[1] and a paradigm shift was needed to mark the stack not executable by default. We have observed that both Windows and Linux programs were linked such that the resulting executable files contained data stored in the code segment. This is not just semantically wrong, but also hindered our XnR solution. While it was easy to fix open source programs by re-compiling them with a modified linker script, the Windows core DLLs remain closed source. With a change of the default `binutils` linker script, XnR could become default. The same is true if Microsoft changed their default linker to put data in read-only data sections of the EXE and DLL files, rather than code.

It is noteworthy that XnR prevents memory disclosures, but the protected program might still suffer from a pointer leakage vulnerability. This is possible because the address a pointer points to (integer value) is considered data even though it might point to code. Such a leaked pointer (e.g. through a malformed `printf`) may reveal function addresses. Therefore, the most basic form of code reuse attacks – return-to-libc – would potentially be possible. In a broader sense, the attacker could use functions as very coarse grained gadgets. However, return-to-libc attacks can be detected by the callee. The callee needs to check whether the last address on the stack is its own address. Usually, it must be any other address but the callee's address because it represents the return address of the caller who used a regular `call` instruction. A `call` pushes the return location to the stack, whereas a `return` pops the address

[1]e.g. trampolines need an executable stack

of the targeted function (*callee*) off the stack. This prevention technique is a very simple form of control-flow integrity checks [P53, M1]. It could either be patched to the prologue of every function by means of binary rewriting or require compiler support.

Our XnR solution might hinder debuggers. On a Linux machine, the `gdb` debugger reads bytes from the code section and even overwrites bytes in the code section to place breakpoints. This is prevented by XnR as shown in Figure 8.12. However, a developer using a debugger can probably also control the XnR kernel feature in order to allow debugging.



```
(gdb) disassemble $rip
[  535.969091] XxorR: Sent SIGBUS to process that tried to read from code segment. Page was not yet mapped (flags=0, faultaddr=0
000000000429000)
[  535.969191] XxorR: Statistics: legitimate exec page faults=174, code read attempts=0)
[  535.969277] XxorR: Sent SIGBUS to process that tried to read from code segment. Page was not yet mapped (flags=0, faultaddr=0
000000000429000)
[  535.969341] XxorR: Statistics: legitimate exec page faults=174, code read attempts=0)
Dump of assembler code for function _start:
[  535.971810] XxorR: Sent SIGBUS to process that tried to read from code segment. Page was not yet mapped (flags=0, faultaddr=0
000000000429000)
[  535.971884] XxorR: Statistics: legitimate exec page faults=174, code read attempts=0)
=> 0x00000000004290e0 <+0>:     Cannot access memory at address 0x4290e0
(gdb) _
```

Figure 8.12: Screenshot of gdb debugger: XnR also disallows disassembly

However, return-to-libc can be easily detected by checking the stack layout at every function prologue. A `call` pushes the return location to the stack, whereas a `return` pops an address off the stack. This means that the callee can easily distinguish full-function ROP-chains from legitimate calls.

The technical building block behind XnR, the ability to dynamically show and hide pages, was also used to implement transparent encryption. This work has been lead by Tilo Müller and Johannes Götzfried from University of Erlangen-Nuremberg and was loosely based on my XnR implementation. The need for transparent encryption originates from the fact that it is usually easier to attack the endpoints where encryption takes place than to attack the underlying cryptographic primitives. RamCrypt allows an unmodified Linux processes to transparently work on encrypted data. It can be deployed and enabled on a per-process basis without recompiling user-mode applications. In every enabled process, data is only stored in cleartext for the moment it is processed, and otherwise stays encrypted in RAM. This work has been published at the ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS) [M5].

9 Conclusion

This dissertation has presented four different mitigation techniques against code reuse attacks – still one of the most prevalent attacks today. MoCFI (chapter 5) is a proof-of-concept for backward compatible protection of legacy binary applications. MoCFI demonstrates that it is indeed possible to protect applications on a closed source ecosystem such as Apple's iOS without access to source code of the application or the operating system. Even though the root cause, the possibility to divert control flow, resides in the way the compiler creates executable code, the MoCFI solution does not need to tackles the code reuse problem at source code or compiler level. Instead, MoCFI shows that it is sufficient to protect the existing binary at the last link of the chain Fault $\rightarrow$ Error $\rightarrow$ Failure by monitoring the symptoms of an exploited vulnerability. Deviations from expected and allowed behaviour are then caught and the hijacked application is stopped before the changed behaviour can cause any harm.

In contrast to MoCFI, the other two code reuse preventions (fine-grained randomisation in chapter 6, Oxymoron in chapter 7) engage one link earlier in the chain by preventing the root cause, diverting control flow, in the first place. These two methods also work on binaries for which no source code is available. They both build on the idea that introducing memory entropy deprives an attacker of necessary address information for a successful exploit.

The fourth proposed solution acknowledges that hiding valuable information from an attacker is a probabilistic game whose outcome depends on the additional information an attacker has. Given an additional memory disclosure vulnerability, the attacker is suddenly in the position to revert hidden addresses, which are vital for a successful attack. This is why XnR (eXecute no Read, chapter 8) prevents such memory disclosure attacks in order to make fine-grained randomisation more secure.

While the proposed methods of this dissertation defend against attacks that have evolved during the course of writing this thesis, the stepwise building of defences is a witness for the fact that this is indeed an arms race. The next defence for a new attack will sure be found, but systems stay vulnerable if the story of code reuse continues to be a never-ending cat-and-mouse game. In order to provide security guarantees against code reuse, it is necessary to put an end to the arms race by discovering a fundamental solution to the code reuse problem. Otherwise, a system is always at risk because new attacks against existing defences could be found. Unfortunately, the other extreme to the arms race is the fact that Rice's theorem [P86] tells us that it is impossible to decide a non-trivial property of a program. Such a non-trivial property is whether a given program is vulnerable to code reuse attacks. That means, we know that it is impossible to generally prove a given program will never suffer from a code reuse vulnerability. This is why this dissertation pushed the state-of-the-art in defences in order to ride on the top of the wave in the ongoing arms race to make it at least impractical for attackers.

List of Figures

References

The following is a list of references that use letters to quickly indicate the kind of reference to the reader. I introduced this enumeration scheme to avoid constantly browsing back and and forth to discover what (kind of) reference was referred to. The reader can thus easily recognise whether I refer to my own work ([M1], [M2], ...) or to other peer-reviewed papers ([P1], [P2], ...). Additionally, technical documentation such as handbooks or manuals are referred to as [D1], [D2], ... whereas Internet links start with [L1], [L2] and so on.

Author’s Peer-Reviewed References

[M1] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones”. *Symposium on Network and Distributed System Security (NDSS)*. 2012.

[M2] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “Gadge Me if You Can: Secure and Efficient Ad-Hoc Instruction-Level Randomization for x86 and ARM”. *ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS)*. 2013.

[M3] Michael Backes and Stefan Nürnberger. “Oxymoron - Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing”. *USENIX Security Symposium*. 2014.

[M4] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. “You Can Run but You Can’t Read: Preventing Disclosure Exploits in Executable Code”. *ACM conference on Computer and communications security (CCS)*. 2014.

[M5] Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. “RamCrypt: Kernel-based Address Space Encryption for User-mode Processes”. *ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS)*. 2016.

[M6] Johannes Krupp, Dominique Schröder, Mark Simkin, Dario Fiore, Giuseppe Ateniese, and Stefan Nürnberger. “Nearly Optimal Verifiable Data Streaming”. *IACR International Conference on Practice and Theory of Public-Key Cryptography (PKC)*. 2016.

[M7] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. “How to Make ASLR Win the Clone Wars: Run-Time Re-Randomization”. *23rd Annual Symposium on Network and Distributed System Security (NDSS 2016)*. 2015.

[M8] Michael Backes, Rainer W. Gerling, Sebastian Gerling, Stefan Nürnberger, Dominique Schröder, and Mark Simkin. “WebTrust - A Comprehensive Authenticity and Integrity Framework for HTTP”. *International Conference on Applied Cryptography and Network Security (ACNS)*. 2014.

[M9] Sören Bleikertz, Sven Bugiel, Hugo Ideler, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “Client-controlled Cryptography-as-a-Service in the Cloud”. *International Conference on Applied Cryptography and Network Security (ACNS)*. 2013.

[M10] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “XIFER: A Software Diversity Tool Against Code-Reuse Attacks”. *ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3)* (2012).

[M11] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. “AmazonIA: When Elasticity Snaps Back”. *ACM conference on Computer and communications security (CCS)*. 2011.

[M12] Sven Bugiel, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. “Twin Clouds: Secure Cloud Computing with Low Latency”. *Communications and Multimedia Security (CMS)*. Springer, 2011.

[M13] Sven Bugiel, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. “Twin Clouds: An Architecture for Secure Cloud Computing”. *Workshop on Cryptography and Security in Clouds (CSC)* (2011).

[M14] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “Poster: Control-flow Integrity for Smartphones”. *ACM conference on Computer and communications security (CCS)*. ACM. 2011.

[M15] Stefan Nürnberger Martin Steinebach Sascha Zmudzinski. “Re-synchronizing Audio Watermarking after Non-linear Time Stretching”. *Electronic Imaging 2011 - Media Watermarking, Security, and Forensics*. 2011.

[M16] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. “CFI Goes Mobile: Control-Flow Integrity for Smartphones”. *International Workshop on Trustworthy Embedded Devices (TrustED)*. 2011.

[M17] Stefan Nürnberger, Thomas Feller, and Sorin A. Huss. “Ray - A Secure Microkernel Architecture”. *IEEE International Conference on Privacy Security and Trust (PST)*. 2010.

[M18] Stefan Thiemert, Stefan Nürnberger, Martin Steinebach, and Sascha Zmudzinski. “Security of Robust Audio Hashes”. *IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE. 2009.

Other References

[P1] James P Anderson. *Computer Security Technology Planning Study*. Tech. rep. U.S. Air Force Electronic Systems Division (AFSC), Bedford, Massachusetts, 1972.

[P2] U.S. Court of Appeals. “United States of America v. Robert Tappan MORRIS”. *928 F.2d 504* 774 (1991).

[P3] MITRE. *Common Weakness Enumeration*. `http://cwe.mitre.org/top25/`. 2015.

[P4] William R Bevier, Warren A Hunt Jr, J Strother Moore, and William D Young. “An Approach to Systems Verification”. *Journal of Automated Reasoning* 5.4 (1989), pp. 411–428.

[P5] Norbert W. Schirmer. “Verification of Sequential Imperative Programs in Isabelle/HOL”. PhD thesis. Technische Universität München, 2006.

[P6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: Formal Verification of an OS Kernel”. *ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM. 2009, pp. 207–220.

[P7] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. “Basic concepts and taxonomy of dependable and secure computing”. *IEEE Transactions on Dependable and Secure Computing* (2004).

[P8] Trevor Jim, Gregory Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. “Cyclone: A Safe Dialect of C.” *USENIX Annual Technical Conference*. 2002, pp. 275–288.

[P9] Hovav Shacham. “The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)”. *ACM Conference on Computer and Communications Security (CCS)*. 2007.

[P10] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. “Microgadgets: size does matter in turing-complete return-oriented programming”. *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association. 2012, pp. 7–7.

[P11] Tyler Durden. “Bypassing PaX ASLR protection”. *Phrack Magazine* 59.9 (2002), pp. 9–9.

[P12] Hovav Shacham, Eu jin Goh, Nagendra Modadugu, Ben Pfaff, and Dan Boneh. “On the Effectiveness of Address-space Randomization”. *ACM Conference on Computer and Communications Security (CCS)*. 2004.

[P13] Kevin Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization”. *IEEE Symposium on Security and Privacy*. 2013.

[P14] Frederick B. Cohen. “Operating system protection through program evolution”. *Computer & Security* 12.6 (1993), pp. 565–584. ISSN: 0167-4048. DOI: `10.1016/0167-4048(93)90054-9`.

[P15] Michael Franz. “E unibus pluram: massive-scale software diversity as a defense mechanism”. *ACM New Security Paradigms Workshop (NSPW)*. Concord, Massachusetts, USA, 2010, pp. 7–16. ISBN: 978-1-4503-0415-3. DOI: `10.1145/1900546.1900550`.

[P16] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. “Compiler-Generated Software Diversity”. *ACM Workshop on Moving Target Defense (MTD)*. 2011.

[P17] Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. “ILR: Where’d My Gadgets Go?” *IEEE Symposium on Security and Privacy (S&P)*. 2012.

[P18] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. “Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization”. *USENIX Security Symposium*. 2012.

[P19] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. “Efficient techniques for comprehensive protection from memory error exploits”. *USENIX Security Symposium*. Baltimore, MD: USENIX Association, 2005.

[P20] H. Xu and S.J. Chapin. “Address-space layout randomization using code islands”. *Journal of Computer Security*. IOS Press, 2009, pp. 331–362.

[P21] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. “Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code”. *ACM Conference on Computer and Communications Security (CCS)*. 2012.

[P22] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. “Differentiating Code from Data in x86 Binaries”. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2011.

[P23] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. “Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection”. *USENIX Security Symposium*. 2014.

[P24] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. “Out of control: Overcoming control-flow integrity”. *IEEE Symposium on Security and Privacy (S&P)*. 2014, pp. 575–589.

[P25] Aleph One. “Smashing the Stack for Fun and Profit”. *Phrack Magazine* 49.14 (1996).

[P26] Jonathan Pincus and Brandon Baker. “Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns”. *IEEE Symposium on Security and Privacy (S&P)* 2.4 (2004), pp. 20–27. ISSN: 1540-7993.

[P27] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. “Missing the Point(er): On the Effectiveness of Code Pointer Integrity”. *IEEE Symposium on Security and Privacy (S&P)*. 2015.

[P28] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. “Jump-oriented programming: a new class of code-reuse attack”. *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2011, pp. 30–40.

[P29] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. “Return-oriented Programming Without Returns”. *ACM Conference on Computer and Communications Security (CCS)*. 2010.

[P30] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. “Non-Control-Data Attacks Are Realistic Threats.” *USENIX Security Symposium*. Vol. 5. 2005.

[P31] Nicholas Carlini and David Wagner. “ROP is Still Dangerous: Breaking Modern Defenses”. *USENIX Security Symposium*. 2014.

[P32] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. “Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard”. *USENIX Security Symposium*. 2014, pp. 417–432.

[P33] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. “Counterfeit Object-oriented Programming - On the Difficulty of Preventing Code Reuse Attacks in C++ Applications”. *IEEE Symposium on Security and Privacy (S&P)*. 2015.

[P34] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. “Hacking Blind”. *IEEE Symposium on Security and Privacy (S&P)*. 2014, pp. 227–242.

[P35] Jeff Seibert, Hamed Okkhravi, and Eric Söderström. “Information leaks without memory disclosures: Remote side channel attacks on diversified code”. *ACM Conference on Computer and Communications Security (CCS)*. 2014, pp. 54–65.

[P36] Ali José Mashtizadeh, Andrea Bittau, David Mazières, and Dan Boneh. “Cryptographically Enforced Control Flow Integrity”. *CoRR* abs/1408.1451 (2014). URL: `http://arxiv.org/abs/1408.1451`.

[P37] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. “G-Free: defeating return-oriented programming through gadget-less binaries”. *ACSAC’10, Annual Computer Security Applications Conference*. Austin, Texas, USA, 2010.

[P38] Zhi Wang and Xuxian Jiang. “HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity”. *IEEE Symposium on Security and Privacy (S&P)*. 2010.

[P39] John Criswell, Nathan Dautenhahn, and Vikram Adve. “KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels”. *IEEE Symposium on Security and Privacy (S&P)*. 2014, pp. 292–307.

[P40] Ben Niu and Gang Tan. “Modular Control-Flow Integrity”. *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. Vol. 49. 6. 2014, pp. 577–587.

[P41] Ben Niu and Gang Tan. “RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity”. *ACM Conference on Computer and Communications Security (CCS)*. 2014, pp. 1317–1328.

[P42] Bin Zeng, Gang Tan, and Greg Morrisett. “Combining control-flow integrity and static analysis for efficient and validated data sandboxing”. *ACM Conference on Computer and Communications Security (CCS)*. 2011.

[P43] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. *USENIX Security Symposium*. 1998.

[P44] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. “Preventing Memory Error Exploits with WIT”. *IEEE Symposium on Security and Privacy (S&P)* (2008).

[P45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “AddressSanitizer: A Fast Address Sanity Checker.” *USENIX Annual Technical Conference*. 2012, pp. 309–318.

[P46] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. “Adapting Software Fault Isolation to Contemporary CPU Architectures”. *USENIX Security Symposium*. 2010.

[P47] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. *IEEE Symposium on Security and Privacy (S&P)* (2009). ISSN: 1081-6011. DOI: `10.1109/SP.2009.25`.

[P48] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. “Code-Pointer Integrity”. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[P49] Albert Kwon, Udit Dhawan, Jonathan Smith, Thomas F. Knight Jr, and Andre DeHon. “Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security”. *ACM Conference on Computer and Communications Security (CCS)*. 2013, pp. 721–732.

[P50] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. “ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks”. *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2011.

[P51] Tzi-cker Chiueh and Fu-Hau Hsu. “RAD: A Compile-Time Solution to Buffer Overflow Attacks”. *IEEE Conference on Distributed Computer Systems*. 2001.

[P52] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. “Readactor: Practical code randomization resilient to memory disclosure”. *IEEE Symposium on Security and Privacy (S&P)*. Vol. 15. 2015.

[P53] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow integrity”. *ACM Conference on Computer and Communications Security (CCS)*. 2005, pp. 340–353.

[P54] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow Integrity: Principles, Implementations, and Applications”. Vol. 13. 1. 2009, p. 4.

[P55] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, George C. Necula, and Michael Vrable. “XFI: Software Guards for System Address Spaces”. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006.

[P56] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dong Song, and Wei Zou. “Practical control flow integrity and randomization for binary executables”. *IEEE Symposium on Security and Privacy (S&P)*. 2013, pp. 559–573.

[P57] Mingwei Zhang and R Sekar. “Control Flow Integrity for COTS Binaries.” *USENIX Security Symposium*. 2013, pp. 337–352.

[P58] Vishwath Mohan, Per Larsen, Stefan Brunthaler, K. Hamlen, and Michael Franz. “Opaque Control-Flow Integrity”. *Symposium on Network and Distributed System Security (NDSS)*. 2015.

[P59] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. “Secure Execution via Program Shepherding”. *USENIX Security Symposium*. 2002.

[P60] Mike Frantzen and Mike Shuey. “StackGhost: Hardware Facilitated Stack Protection”. *USENIX Security Symposium*. 2001.

[P61] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. “Transparent ROP Exploit Mitigation Using Indirect Branch Tracing.” *USENIX Security Symposium*. 2013, pp. 447–462.

[P62] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert Deng. “ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks”. *Symposium on Network and Distributed System Security (NDSS)*. 2014.

[P63] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. “Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software”. *Annual Computer Security Applications Conference (ACSAC)*. 2006.

[P64] Periklis Akritidis. “Cling: A Memory Allocator to Mitigate Dangling Pointers.” *USENIX Security Symposium*. 2010, pp. 177–192.

[P65] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. “Randomized instruction set emulation to disrupt binary code injection attacks”. *ACM Conference on Computer and Communications Security (CCS)*. 2003.

[P66] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. “librando: Transparent Code Randomization for Just-in-Time Compilers”. *ACM Conference on Computer and Communications Security (CCS)*. 2013, pp. 993–1004.

[P67] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. “Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization”. *IEEE Symposium on Security and Privacy (S&P)*. 2012.

[P68] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. Vol. 42. 6. 2007, pp. 89–100.

[P69] 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”. *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. Vol. 40. 6. 2005, pp. 190–200.

[P70] D. Bruenning. “Efficient, Transparent and Comprehensive Run-time Code Manipulation”. PhD thesis. Massachusetts Institute of Technology, 2004.

[P71] Jason Gionta, William Enck, and Peng Ning. “HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities”. *ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2015, pp. 325–336.

[P72] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Snow, and Fabian Monrose. “Isomeron: Code Randomization Resilient to (Just-in-Time) Return-Oriented Programming”. *Symposium on Network and Distributed System Security (NDSS)* (2015).

[P73] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. “Pointguard^{TM}: Protecting Pointers From Buffer Overflow Vulnerabilities”. *USENIX Security Symposium*. 2003.

[P74] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. “On the expressiveness of return-into-libc attacks”. *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*. Springer-Verlag, 2011.

[P75] Charlie Curtsinger and Emery D Berger. “STABILIZER: statistically sound performance evaluation”. *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Vol. 41. 1. ACM. 2013, pp. 219–228.

[P76] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. “MINIX 3: A highly reliable, self-repairing operating system”. *ACM SIGOPS Operating Systems Review (OSR)* 40.3 (2006), pp. 80–89.

[P77] Dino Dai Zovi. “Apple iOS Security Evaluation: Vulnerability Analysis and Data Encryption”. *Black Hat USA*. 2011.

[P78] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. “PiOS: Detecting Privacy Leaks in iOS Applications”. *Symposium on Network and Distributed System Security (NDSS)*. 2011.

[P79] Mark Weiser. “Program slicing”. *IEEE Conference on Software Engineering (ICSE)*. 1981, pp. 439–449.

[P80] Jerome Saltzer and Michael D Schroeder. “The protection of information in computer systems”. *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[P81] Matthew Smithson, Kapil Anand, Aparna Kotha, Khaled Elwazeer, Nathan Giles, and Rajeev Barua. *Binary Rewriting without Relocation Information*. Tech. rep. University of Maryland, 2010.

[P82] B. De Sutter, B. De Bus, and K. De Bosschere. “Link-time binary rewriting techniques for program compaction”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27.5 (2005), pp. 882–945.

[P83] A. Eustace and A. Srivastava. “ATOM: A flexible interface for building high performance program analysis tools”. *USENIX Annual Technical Conference*. 1995, pp. 25–25.

[P84] Sherri Sparks and Jamie Butler. ““Shadow Walker”: Raising the Bar for Rootkit Detection”. *Black Hat Japan* 11.63 (2005).

[P85] Peter J Denning. “The Working Set Model for Program Behavior”. *Communications of the ACM* 11.5 (1968), pp. 323–333.

[P86] H. G. Rice. “Classes of recursively enumerable sets and their decision problems”. *Transactions of the American Mathematical Society*. 1953.

Documentation

[D1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*.

[D2] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[D3] Microsoft Virtual Academy. *Windows 8 Security Insights*. http://download.microsoft.com/download/9/9/4/994592CB-C248-464F-93A6-A50E339BE19B/Windows%208%20Security%20-%20ASLR.pdf.

[D4] *Procedure Call Standard for the ARM Architecture*. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf. 2009.

[D5] 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.

[D6] Microsoft Corp. *The SAFESEH compiler flag*. https://msdn.microsoft.com/en-us/library/9a89h429.aspx.

[D7] Apple Inc. *App Store Review Guidelines*. https://developer.apple.com/app-store/review/guidelines/. 2015.

[D8] *Executable and Linking Format (ELF)*. Tool Interface Standards Committee. 1995.

[D9] *DWARF 2.0 debugging format standard*. http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf.

[D10] Intel. *64 and IA-32 Architecture Software Developer's Manual, Volume 1, Basic Architecture*. http://download.intel.com/design/processor/manuals/253665.pdf.

[D11] Agner Fog Research. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. http://www.agner.org/optimize/instruction_tables.pdf.

[D12] Microsoft. *Data Execution Prevention (DEP)*. http://support.microsoft.com/kb/875352/EN-US/. 2006.

Links

[L1] Thomas Bruckschlegel. *Microbenchmarking C++, C# and Java*. http://www.drdobbs.com/cpp/microbenchmarking-c-c-and-java/184401976?pgno=2. 2005.

[L2] Chromium Blog. *A new crankshaft for V8: Benchmark suite, version 6*. http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html.

[L3] PaX Team. *PaX Address Space Layout Randomization (ASLR)*. http://pax.grsecurity.net/docs/aslr.txt.

[L4] The Linux Foundation. *The Xen Hypervisor*. http://www.xenproject.org/.

[L5] Japan University of Tsukuba. *BitVisor hypervisor*. http://www.bitvisor.org.

[L6] The FreeBSD Project. *The FreeBSD Operating System*. https://www.freebsd.org/.

[L7] LLVM Developer Group (formerly University of Illinois at Urbana-Champaign). *The LLVM Compiler Infrastructure*. http://llvm.org/.

[L8] Apple Inc. and others. *clang: a C language family frontend for LLVM*. http://clang.llvm.org/.

[L9] Intel. *Pin - A Dynamic Binary Instrumentation Tool*. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[L10] Ivan Fratrić. *ROPGuard: run-time Prevention of Return-Oriented Programming Attacks*. https://github.com/ivanfratric/ropguard. 2012.

[L11] Apple, Inc. *Secure Coding Guide*. https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf.

[L12] Jon Oberheide. *A look at ASLR in Android Ice Cream Sandwich 4.0*. https://www.duosecurity.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0.

[L13] Apple, Inc. *iOS 8.3 Security Guide*. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[L14] Hex Rays. *IDA Disassembler*. https://www.hex-rays.com/products/ida/.

[L15] Jordi Perez Barreiro. *Gensystek Benchmark*. https://itunes.apple.com/us/app/gensystek-benchmark/id447054674.

[L16] *ROPgadget*. http://shell-storm.org/project/ROPgadget/.

[L17] *How to hijack the Global Offset Table with pointers for root shells*. http://www.open-security.org/texts/6.

[L18] *Common Vulnerabilities & Exposures (CVE) database*. http://cve.mitre.org/.

[L19] Fabrice Bellard. *Qemu FAST! Processor Emulator*. http://www.qemu.org/.

[L20] Oracle Corporation. *Oracle VM VirtualBox*. https://www.virtualbox.org/.

[L21] Microsoft. *Enhanced Mitigation Experience Toolkit (EMET)*. http://www.microsoft.com/emet.

[L22] GCC Project. *The GNU Compiler Collection*. https://gcc.gnu.org/.

[L23] *BusyBox: The Swiss Army Knife of Embedded Linux*. http://www.busybox.net/.