Praktische beschermingen voor binaire programma's

Practical Protections for Native Programs

Bert Abrath

Promotoren: prof. dr. ir. B. De Sutter, dr. B. Coppens
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2018 - 2019

ISBN 978-94-6355-263-9
NUR 980
Wettelijk depot: D/2019/10.500/71

Examination Board

Prof. Hennie De Schepper, *chair*
Department of Electronics and Information Systems
Faculty of Engineering and Architecture
Ghent University

Prof. Koen De Bosschere, *secretary*
Department of Electronics and Information Systems
Faculty of Engineering and Architecture
Ghent University

Prof. Bjorn De Sutter, *supervisor*
Department of Electronics and Information Systems
Faculty of Engineering and Architecture
Ghent University

dr. Bart Coppens, *supervisor*
Department of Electronics and Information Systems
Faculty of Engineering and Architecture
Ghent University

Prof. Bart Dhoedt
Department of Information Technology
Faculty of Engineering and Architecture
Ghent University

Prof. Sebastian Schrittwieser
Department Informatik und Security
FH St. Pölten

Prof. Stijn Volckaert
Department of Computer Science
Faculty of Engineering Technology
KU Leuven

dr. Brecht Wyseur
Kudelski Group

Life is what happens to you while you're busy ~~making other plans~~ writing a PhD

Dankwoord

Tegenwoordig is onderzoek een groepsactiviteit. Het is dan ook quasi onmogelijk voor één persoon om de eer op te strijken van het resultaat. Nadat het bestaan van het Higgs-boson in 2012 bevestigd werd, resulteerde verder onderzoek naar de precieze massa van het deeltje in een publicatie met maar liefst *5154* auteurs. Hoewel de situatie in mijn onderzoeksveld helemaal niet zo extreem is, zou dit proefschrift niet volledig zijn zonder een welgemeende dankbetuiging aan iedereen die er aan bijgedragen heeft. Hiermee bedoel ik niet enkel diegenen die op directe wijze betrokken waren bij het onderzoek—inclusief iedereen die ik ooit de rol van *rubber ducky* heb toebedeeld—maar ook diegenen die indirect bijdroegen, via enthousiasme, morele steun, of algemene afleiding.

Eerst en vooral wil ik mijn promotoren, Bjorn en Bart, bedanken. Bjorn, je gaf mij al die jaren geleden de opportuniteit om een doctoraat aan te vatten, en bood me ondersteuning in de vorm van je kritische zin en scherp inzicht. Bart, hoewel je slechts sinds kort mijn copromotor geworden bent, had je deze rol al lange tijd officieus op je genomen.

I would also like to thank the members of my examination board: Koen De Bosschere, Hennie De Schepper, Bart Dhoedt, Sebastian Schrittwieser, Stijn Volckaert, and Brecht Wyseur. I very much appreciate the time and effort spent on reading and providing feedback on this dissertation, and on attending the internal defence.

Ik dank ook graag al mijn (ex-)collega's, zowel in het Technicum als te iGent: Bart, Christophe, Jens, Jeroen, Jonas, Niels, Panagiotis, Pieter, Ronald, en Stijn. De vele (al dan niet werkgerelateerde) discussies, weetjes, board game nights, zwarte humor, en karakteristiek ongevraagde droom-uiteenzettingen hebben jarenlang voor een omgeving gezorgd waar het goed was om in te vertoeven, te werken, en te leren. Daarnaast wil ik ook graag Eneko, Marnix, Michiel, Ronny, en Vicky bedanken voor de administratieve en technische ondersteuning die zij doorheen de jaren geboden hebben.

I had the good fortune to pursue a large part of my doctoral research within the ASPIRE project. This meant plentiful conference calls and in-person meetings all over Europe, during which not only a lot of creativity was unleashed, but many good memories were made as well. I especially want to thank Aldo, Alessandro, Alessio, Brecht, and Paolo for their close and pleasant collaboration.

Ik heb veel goede vrienden overgehouden aan mijn tijd op Home Fabiola. Specifiek wil ik Jason, Joffrey, Sofie D.L. en Sofie M. bedanken voor het aangename gezelschap—en de vreemde discussies—op de vele barbecues en movienights, en Dimitri verzekeren dat het Star Trek project vervolledigd zal worden (hoewel een voorzichtige schatting een einddatum in 2022 à 2023 doet vermoeden).

Ongeveer sinds het begin van mijn doctoraat woon ik samen met Jeroen en Klaas. Hoewel het einde van dit doctoraat weldra gepaard zal gaan met mijn vertrek uit ons appartement, zullen de vele series, spelletjes, en verhalen me nog lang bijblijven. Bedankt voor jullie gezelschap tijdens deze lange en soms moeilijke jaren; ik wens jullie het allerbeste in de toekomst (en met de nieuwe huisgenoot).

Mijn vriendengroep uit het middelbaar is doorheen de jaren wat uitgebreid en uitgedijd (al dan niet in fysieke zin) en lag aan de basis van vele amusante road trips, ski-trips, en kampeertrips (waarvan sommige met bevroren tenten). Ik ben Diede, Jannes, Joachim, Jonas, Roel, en Wouter dankbaar voor hun ongekende vermogen om sfeer (en taalmopjes!) te maken, en kijk uit naar de rest van de vrijgezellenfeesten (hopelijk met een minimum aan brandwonden).

De weekendjes en barbecues van ons groepje CW'ers waren een dankbare bron van amusement, enige beroepsmisvormde opmerkingen, en toffe quizzen.

Dan wil ik nog mijn familie bedanken: mijn ouders, mijn zussen Margo en Sofie, en de gezinsuitbreidingen van de voorbije jaren Francis, Mathiz, en Ellis. Bedankt voor niet té vaak te vragen hoe lang het nog ging duren tot ik mijn doctoraat had, wat daar nog voor moest gebeuren, en mijn algemene verstrooidheid te aanvaarden, maar ook voor de steun, verjaardagsfeestjes, en spelletjesavonden.

Tot slot: Het geluk zit hem in de kleine dingen. Sigy, tijdens deze lange tocht kwam je onverwachts in mijn leven, en was je er al gauw niet meer uit weg te denken. Bedankt voor je steun en aanmoediging, *the mutual weirdness*, en het voorzien van Pudding tijdens moeilijke momenten.

Gent, 3 juli 2019
Bert Abrath

Samenvatting

Eens de ontwikkeling van een programma of een ander stuk software voltooid is en het naar zijn eindgebruikers verspreid is, willen de ontwikkelaars dat hun software op de bedoelde manier uitvoert. Het kan echter in het voordeel van een aanvaller zijn dat de software van zijn bedoelde gedrag afwijkt. Beschouwen we twee scenario's: in het eerste scenario bevat de software bugs die aanvallers kunnen uitbuiten om het programma over te nemen en te laten gedragen op onbedoelde wijze; in het tweede scenario loopt het programma op de aanvallers' eigen machine, en kunnen deze met het gedrag knoeien—zelfs zonder bugs. Het eerste beschreven scenario is dat van *arbitraire code-uitvoering*, het tweede is een *man-at-the-end* of *MATE*-aanval. Arbitraire code-uitvoering kan door aanvallers gebruikt worden om een server op afstand over te nemen, en diens waardevolle data zoals een paswoordendatabank in gevaar te brengen. Voor MATE (Man-At-The-End)-aanvallen daarentegen, zijn de aanvallers de vermeende eindgebruikers van de software, en ligt hun doel in de eigendommen vervat in het programma zelf: cryptografische sleutels, gepatenteerde algoritmes, of licentiecontroles. Ongeacht het soort aanval, is het duidelijk dat software beschermd moet worden om ernstige economische schade te vermijden. Het is dus belangrijk dat de mensen die software ontwerpen en ontwikkelen de veiligheid ervan in acht nemen. We kunnen echter niet verwachten dat elke software-architect en ontwikkelaar een expert is op het vlak van beveiliging. Ze zijn niet op de hoogte van alle mogelijke aanvallen, noch hebben ze kennis van alle gepaste beschermingen. Het softwareontwikkelingsproces hoort dus meer gedachtig te zijn op het vlak van beveiliging, maar we kunnen de betrokken mensen ook helpen door het toepassen van beschermingstechnieken te integreren in de ondersteunende hulpmiddelen die gebruikt worden door de levenscyclus van software. Die hulpmiddelen staan in het algemeen bekend onder de noemer *systeemsoftware*.

Ik heb mijn onderzoek verricht op het niveau van systeemsoftware, waarbij mijn focus lag op het verdedigen tegen arbitraire code-uitvoering en MATE-aanvallen. Mijn doelen waren het verbeteren van bestaande en het introduceren van nieuwe beschermingen, alsook het verbeteren van de ondersteuning voor beschermingen om hun gebruiksgemak te verbeteren. Het merendeel van mijn werk situeerde zich op het *binaire* niveau, waarbij binaire programmacode getransformeerd wordt met het oog op bescherming.

Als bescherming tegen aanvallers wordt er vaak gebruik gemaakt van diversiteit. Diversiteit houdt in dat elke gebruiker zijn eigen, gediversifieerde versie van het programma heeft. Bij de vorm van diversiteit die wij bestudeerden, verschillen de programmabestanden die verspreid worden aan de gebruikers. Diversiteit voorziet een probabilistische bescherming tegen aanvallen die arbitraire code-uitvoering beogen, en belemmert daarnaast ook MATE-aanvallers. Elke gebruiker zijn eigen programmabestand geven leidt echter tot een aantal uitdagingen die het wijdverspreid gebruik van diversiteit tegenhouden. Eén zo'n uitdaging is met crashrapporteersystemen, die het genereren en aggregeren van crashrapporten automatiseren. Google Breakpad is een voorbeeld van een type crashrapporteersysteem waarbij de crashrapporteerserver de debuginformatie van het programmabestand bijhoudt. Om echter gediversifieerde programma's te ondersteunen, zou deze server de debuginformatie voor elk gediversifieerd programmabestand moeten bijhouden. Voor programma's met miljoenen gebruikers is dit natuurlijk niet schaalbaar. We hebben voor dit probleem een oplossing ontworpen, waarbij Google Breakpad uitgebreid wordt tot ΔBreakpad. ΔBreakpad werkt door programma's te diversifiëren, en deze uit te breiden met informatie die de toegepaste diversificaties beschrijft. Dit laat de server toe om de debuginformatie te reconstrueren voor een gediversifieerd programmabestand dat crashte. Deze oplossing vergemakkelijkt het gebruik van diversiteit, en zorgt zo voor meer veiligheid.

Hoewel diversiteit ook kan verdedigen tegen MATE-aanvallen, wordt het voornamelijk als bescherming tegen arbitraire code-uitvoering gebruikt. Als we onze focus verleggen naar MATE-scenario's, kan een programma aangevallen worden via talrijke aanvalsvectoren. Bijgevolg bestaan er ook talrijke beschermingstechnieken die het gebruik van specifieke aanvalsvectoren bemoeilijken. We hebben twee van zulke technieken verbeterd: self-debugging, en code mobility.

Self-debugging is een beschermingstechniek waarbij aanvallers worden belet hun debugger aan het programma aan te hechten, gezien debuggers vaak worden gebruikt in MATE-aanvallen. Het programma transformeren zodat het niet van buitenaf gedebugged kan worden voorziet het dus van bescherming. Omdat slechts één enkele debugger kan aangehecht worden per proces, kan het kwaadwillig gebruik van een debugger door aanvallers tegengehouden worden door het programma zichzelf te laten debuggen. Deze bescherming wordt geïmplementeerd door een debuggercomponent in te bedden in het programma tijdens het buildproces, en het programmabestand zo te herschrijven zodat het afhankelijk wordt van de debugger, die dan niet meer simpelweg verwijderd kan worden. Hoewel er reeds implementaties van self-debugging bestaan, zijn de gebruikte methodes om een afhankelijkheid creëren tussen het programma en de ingebedde debugger onvoldoende. Onze verbeterde techniek zorgt voor een hechtere koppeling tussen het programma en zijn debugger, waardoor de techniek minder kwetsbaar is voor geautomatiseerde aanvallen.

Code mobility is een online beschermingstechniek waarbij delen van het programma worden gedownload van een vertrouwde server tijdens de uitvoering. Dit bemoeilijkt het analyseren van en knoeien met programma's door aanvallers. Vooral statische aanvallen worden hierdoor belemmert, gezien slechts een deel van het volledige programma aanwezig is in het programmabestand. Echter, bestaande implementaties van code mobility werken oftewel niet op het binaire niveau, oftewel laten het mobiel maken van uitgekozen delen van het programma niet toe, wat het combineren met andere beschermingstechnieken op het binaire niveau belemmert. Daarom ontworpen we ons eigen raamwerk voor code mobility. Dit raamwerk werd op binair niveau geïmplementeerd, middels een binaire herschrijver. Door middel van annotaties in de broncode kunnen ontwikkelaars specificeren welke delen van het programma mobiel gemaakt moeten worden. Het raamwerk splijt de binaire code verbonden aan deze annotaties af van het programma en vormt er mobiele codeblokken uit. Dit bemoeilijkt het verstaan van de code door aanvallers, maar laat ook een fijnere granulariteit in de codeblokken toe. Daarenboven kan deze beschermingstechniek ook gebruikt worden om delen van andere beschermingen bij te werken, wat krachtigere combinaties toelaat.

Hoewel het verbeteren van individuele beschermingstechnieken belangrijk is, kan het gebruik van één aanvalsvector bemoeilijken er toe leiden dat aanvallers hun aandacht vestigen op andere aanvalsvectoren die,

in vergelijking, gemakkelijker geworden zijn. Bijgevolg is het nog belangrijker om meerdere beschermingstechnieken te combineren zodat alle mogelijke wegen van de minste weerstand verhard worden. Bovendien kost het tijd voor aanvallers om nieuwe aanvalsvectoren te identificeren, en vervolgens een aanval op te zetten en op te schalen. Nieuwe aanvallen ontwikkelen is een iteratief proces, en herhaalbaarheid is dus belangrijk: als het doelprogramma verandert tijdens dit proces moeten aanvallers hun hulpscripts aanpassen, en misschien hun aanpak heroverwegen of de aanvalsvectoren opnieuw identificeren. Als beschermingstechnieken gediversifieerd kunnen worden voor verschillende gebruikers alsook doorheen de tijd, dan zal dit zowel de benodigde inspanningen om een nieuwe aanval te ontwikkelen doen toenemen, als de tijdspanne verkleinen waarbinnen de aanval werkt en opbrengsten genereert voor de aanvallers. Op basis van onze code mobility techniek hebben we dus een raamwerk ontworpen dat beschermingscombinaties kan variëren doorheen de tijd, en dus de beschermingen *vernieuwen*. Dit raamwerk maakt frequente aanpassingen aan de software mogelijk, en maakt het bijgevolg nog moeilijker—en minder winstgevend—om deze aan te vallen.

Samenvattend kunnen we stellen dat dit werk de grenzen verlegd heeft op het vlak van beschermingen tegen MATE-aanvallen, en het gebruik vergemakkelijkt van diversiteitstechnieken die beschermen tegen zowel MATE-aanvallen als arbitraire code-uitvoering.

Summary

After development of a piece of software has finished and it has been released to its end users, the developers want their software to execute in the way it is intended to execute. However, there might be attackers out there who might benefit from the software deviating from its intended behavior. Consider two scenarios: in the first, the software contains bugs that can be exploited by attackers to take over the program and make it behave in unintended ways; in the second, the program runs on the attackers' own machine, and can be tampered with—even without any bugs and exploits. This first described scenario is known as *arbitrary code execution*, the second as a *MATE* (*Man-At-The-End*) attack. Arbitrary code execution can be used by attackers to take over a remote server and compromise the valuable data there such as a database containing credentials. In the case of MATE attacks, on the other hand, the attackers are the supposed end users of the software, and they target valuable assets embedded in the program itself: cryptographic keys, proprietary algorithms, or licence checks. No matter the specific attack, it is clear that software needs to be protected to avoid serious economical damage. It is thus important for those that design and develop software to keep its security in mind. However, we cannot expect every software architect or developer to be a security expert, knowing every possible way in which their software can be attacked and every possible way to guard it. The software development process should be more mindful of security, but we can also help by integrating the application of protection techniques into the supporting tools used during the life-cycle of the software, which are generally known as *system software*.

I performed my research at the system software level, and focused on protecting against arbitrary code execution and MATE attacks. My aims were improving existing protections, introducing new protections, and improving support for protections to ease their adoption. Most of my

work on protections was at the *binary* or *native* level, directly transforming the binary code of a program to protect it.

Diversity is often used to thwart attackers. Diversity means every user has their own, diversified version of the program. In the form of diversity we studied, all of the program binaries distributed to users differ. Diversity provides a probabilistic defense against attacks aiming to gain arbitrary code execution, and obstructs MATE attackers as well. Having every user run their own program binary presents some challenges, however, slowing the adoption of diversification techniques. One issue is with crash-reporting systems, which automate the generating and aggregation of crash reports. Google Breakpad is an example of a type of crash-reporting system that requires the crash-reporting server to store the binary's debug information. To support diversified programs the server would have to store the debug information for every diversified binary. For programs with millions of users, this is not scalable. We designed a solution to this problem, extending the Google Breakpad crash-reporting system to create ΔBreakpad. ΔBreakpad works by diversifying programs and extending them with information describing the applied diversifications, allowing the server to reconstruct the debug information for a diversified binary that crashed. This solution facilitates the adoption of diversity, thus improving security.

Although diversity can defend against MATE attacks, it is mostly used as a defense against arbitrary code execution. When shifting our focus to a MATE scenario, there are many attack vectors through which the program can be targeted. Consequently, there are also many protection techniques that make it harder for attackers to use specific vectors. We improved on two of these techniques: self-debugging, and code mobility.

Self-debugging is a protection technique that aims to keep attackers from attaching a debugger. Debuggers are often used in MATE attacks, and transforming a program in such a way that is cannot be debugged thus provides protection. As only a single debugger can be attached to a process, the malicious use of a debugger by attackers can be countered by having the protected program debug itself. This protection is implemented by embedding a debugger component in the program during its build process, and rewriting the program binary so that it depends on the debugger, meaning the debugger cannot simply be removed. While self-debugging implementations already exist, the methods through which they make the program depend on the embedded debugger are inadequate. Our improved technique creates a tighter coupling between

the program and its debugger, thus making the technique less vulnerable to automated attacks.

Code mobility is an online protection technique where parts of the program are downloaded at run time from a trusted server, making it harder for attackers to analyze and tamper with the program. In particular, this hinders static attacks, as only part of the program is present in the binary. However, existing implementations of code mobility either do not operate on binary code, or do not allow making only select parts of a program mobile, hindering composability with other protections at the binary level. We therefore present our own code mobility framework, implemented at a binary level using a binary rewriter. Through source code annotations, developers can specify which parts of the program are to be made mobile. The framework then splits off the pieces of native code associated with these annotations, and forms mobile code blocks out of them. This makes code comprehension harder, but also allows for finer granularity in the code blocks that are made mobile. Furthermore, our code mobility technique can be used to replace parts of other protections, which allows more powerful combinations.

While improving individual protection techniques is important, making it harder for attackers to use one attack vector might focus their attention on other attack vectors which have become easier by comparison. Therefore, it raises the importance of combining multiple techniques to ensure possible paths-of-least-resistance are hardened. Furthermore, it takes time for attackers to identify successful attack vectors, and subsequently use these to set up and scale up attacks. Developing new attacks is an iterative process, and repeatability is thus important: Attackers expect repeated program executions to be sufficiently similar that their attacks keep working, both during development and after distribution of these attacks. If the target program changes while an attack is being developed, attackers have to adapt their helper scripts and perhaps rethink their approach or re-identify attack vectors. If protections can be diversified across user instances, as well as across time, this will both increase the attackers' effort in developing an attack, and decrease the window of opportunity during which this attack works and generates income for them. Building upon our code mobility technique, we therefore developed a framework that diversifies protection combinations across time, i.e., *renews* the protections. This renewability framework allows for frequent changes to the software components under attack, and consequently makes it even harder—and less profitable—to attack the software.

In conclusion we can summarize this work has pushed the state of the art in protecting against MATE attacks, as well as eased the adoption of diversification-based techniques that are defend against both MATE attacks and arbitrary code execution.

Contents

Nederlandstalige samenvatting **VII**

English Summary **XI**

1 Introduction **1**
1.1 Contributions 2

2 Background **9**
2.1 System Software and the SDLC 9
2.1.1 Building Software 10
2.1.2 Executing Software 13
2.1.3 Debugging Software 15
2.1.4 Maintaining Software 17
2.2 Attack Models 20
2.2.1 Security Principles 20
2.2.2 Arbitrary Code Execution 21
2.2.3 MATE Attacks 22
2.3 Existing Defenses 24
2.3.1 Obfuscation: Protecting Confidentiality 24
2.3.2 Tamperproofing: Protecting Integrity 27
2.3.3 Anti-Debugging 28
2.3.4 Diversity 29

3 ΔBreakpad **31**
3.1 Background & Problem Statement 33
3.1.1 Offset Diversification 33
3.1.2 Necessary Debug Information 35
3.1.3 Indirect Effects in x86 Binaries 37
3.1.4 Indirect Effects in ARMv7 Binaries 38
3.2 The ΔBreakpad Approach 39
3.2.1 Crash Handling & Stack Trace Generation 41

3.2.2 Generating the Δdata . . . 43
3.2.3 Combining Multiple Diversification Processes . . . 43
3.2.4 Δ-Minimization . . . 46
3.2.5 Profile-Guided Diversification . . . 50
3.3 Prototype Diversification Tool Flow . . . 50
3.3.1 Stack Padding . . . 50
3.3.2 NOP Insertion . . . 51
3.3.3 Function Shuffling . . . 53
3.3.4 Δdata . . . 54
3.4 Experimental Evaluation . . . 54
3.4.1 Benchmarks and Correctness . . . 54
3.4.2 Overhead . . . 55
3.5 Discussion . . . 60
3.5.1 Alternative Designs . . . 60
3.5.2 General Applicability . . . 61
3.6 Related Work . . . 63
3.7 Conclusions and Future Work . . . 64

4 Tightly-Coupled Self-Debugging **65**
4.1 Design . . . 67
4.2 Tool Support . . . 70
4.2.1 Source Code Annotations . . . 70
4.2.2 Standard Compilers and Tools . . . 71
4.2.3 Binaries, Libraries, and Processes . . . 72
4.3 Implementation . . . 72
4.3.1 Initialization & Finalization . . . 72
4.3.2 Multithreading Support . . . 73
4.3.3 Control Flow . . . 74
4.3.4 Memory Accesses . . . 76
4.3.5 Combining Self-Debugging With Other Protections 77
4.4 Evaluation . . . 79
4.4.1 Evaluation Platform . . . 79
4.4.2 Use Cases . . . 79
4.4.3 Correctness . . . 81
4.4.4 Execution Overhead . . . 82
4.4.5 Security Analysis . . . 82
4.4.6 Penetration Tests . . . 84
4.5 Practical Considerations . . . 85
4.5.1 Fragment Selection . . . 85
4.5.2 Impact on Multithreading . . . 85
4.5.3 OS Limitations . . . 86

4.6 Conclusions and Future Work . . . 86

5 Native Code Mobility 89
5.1 Architecture . . . 90
5.1.1 Binder . . . 92
5.1.2 Downloader . . . 95
5.1.3 Server-Side Components . . . 96
5.2 Offset-Independent Mobile Code . . . 96
5.3 Automated Tool Support . . . 100
5.3.1 Specifying Mobile Regions . . . 102
5.3.2 Compilation With Standard Compilers . . . 103
5.3.3 Binary Code Rewriting . . . 103
5.3.4 Current Status and Limitations . . . 105
5.3.5 Testing . . . 106
5.4 Performance Analysis . . . 106
5.5 Related Work . . . 109
5.6 Conclusions and Future Work . . . 109

6 Native Code Renewability 111
6.1 Attack Model . . . 112
6.2 Architecture . . . 115
6.3 Integrating Renewability into Existing Applications . . . 118
6.3.1 Renewability Policies . . . 119
6.3.2 Renewability Communication Design . . . 120
6.4 Mobile Data Blocks . . . 121
6.5 Tool Flow Support . . . 123
6.5.1 Existing Static Protections and Mobility Tool Flow 123
6.5.2 Renewable Code Generator Generation . . . 125
6.5.3 Renewable Code Generation . . . 126
6.5.4 Discussion . . . 126
6.6 Mitigations Against Concrete Attacks . . . 128
6.6.1 Syntactically Diversified Mobile Code . . . 128
6.6.2 Semantically Diversified Mobile Code . . . 129
6.6.3 Diversified Static-To-Procedural Conversion . . . 132
6.6.4 Dynamic and Time-Limited WBC . . . 132
6.6.5 Diversified Instruction Set Randomization . . . 133
6.6.6 Evolving Protections . . . 134
6.7 Experimental Evaluation . . . 136
6.7.1 Target Platform of Prototype Implementation . . . 136
6.7.2 Validation on Use Cases . . . 136
6.7.3 Performance Overhead . . . 141

6.8 Related Work 145
6.9 Conclusions and Future Work 147

7 Conclusions and Future Work 149

A Quantitative Analysis for Δ-Minimization 153

Bibliography 157

List of Tables

3.1 Data sizes and execution times for ΔBreakpad 56

4.1 Feature matrix of the self-debugging use cases 80
4.2 Overhead of self-debugging transformations 81

5.1 Code mobility performance overhead 106
5.2 Code mobility computational overhead 108

6.1 Feature matrix of the renewability use cases 137
6.2 Effects of increasing the number of diversified versions for function parameter reordering 140
6.3 Client wall-clock execution times and network throughput of renewability on bzip2 143
6.4 Server CPU consumption for bzip2 143
6.5 Baseline overhead of code mobility on bzip2 143
6.6 Client CPU consumption and wall-clock execution times of renewable WBC 144
6.7 Network throughput of renewable WBC 144

List of Figures

2.1 Google Breakpad overview 19

3.1 Stack frames in original and diversified binaries 33
3.2 Source line mapping in the symbol file 35
3.3 Stack unwinding information in the symbol file 36
3.4 ΔBreakpad overview 40
3.5 Correlation between binary code size and Δdata size . . 58
3.6 Impact of FP optimizations on Δdata size 59

4.1 Self-debugging overview 68
4.2 Tool flow of self-debugging support 70

5.1 Code mobility high-level architecture 91
5.2 Function calls before and after code mobility transformations 93
5.3 Calling an already downloaded mobile function 94
5.4 Example of offset-independent code 97
5.5 Code mobility tool flow 101

6.1 ASPIRE Renewability Architecture 116
6.2 Tool flow for generation of renewable blocks 124

A.1 Histograms of the variation in function size 154

List of Acronyms

ABI . . . . Application Binary Interface

ACCL . . . . ASPIRE Communication Control Logic

API . . . . Application Programming Interface

ASLR . . . . Address Space Layout Randomization

CFG . . . . Control Flow Graph

FP . . . . Frame Pointer

GMRT . . . . Global Mobile Redirection table

GOT . . . . Global Offset Table

GP . . . . Global Pointer

I/O . . . . Input/Output

IR . . . . Intermediate Representation

ISA . . . . Instruction Set Architecture

LTO . . . . Link-Time Optimization

MATE . . . . Man-At-The-End

OS . . . . Operating System

PC . . . . Program Counter

PIC . . . . Position-Independent Code

PID . . . . Process ID

SDLC . . . . Software Development Life Cycle

SP . . . . Stack Pointer

VM . . . . Virtual Machine

WBC . . . . White-Box Cryptography

WPCFG . . . . Whole-Program Control Flow Graph

Chapter 1

Introduction

"The universe is run by the complex interweaving of three elements: energy, matter, and enlightened self-interest."

—G'Kar, *Babylon 5*

After development of a piece of software has finished and it has been released to its end users, the developers want their software to execute in the way it is intended to execute. However, there might be attackers out there who might benefit from the software deviating from its intended behavior. Consider two scenarios: in the first, the software contains bugs that can be exploited by attackers to take over the program and make it behave in unintended ways; in the second, the program runs on the attackers' own machine, and can be tampered with—even without any bugs and exploits. This first described scenario is known as *arbitrary code execution*, the second as a *MATE* (*Man-At-The-End*) attack. Arbitrary code execution can be used by attackers to take over a remote server and compromise the valuable data there such as a database containing credentials. In the case of MATE attacks, on the other hand, the attackers are the supposed end users of the software, and they target valuable assets embedded in the program itself: cryptographic keys, proprietary algorithms, or licence checks. No matter the specific attack, it is clear that software needs to be protected to avoid serious economical damage. It is thus important for those that design and develop software to keep its security in mind.

We cannot expect every software architect or developer to be a security expert, knowing every possible way in which their software can be attacked and every possible way to guard it. Neither can we expect every company to hire security experts for all the software they develop, as this would become too expensive. Next to that, humans are fallible creatures. Programmers make mistakes, and almost all software has bugs in it. Thus, while the software development process should be more mindful of security, we can help the people involved by introducing security in an automated, integrated manner.

The past five years I have worked at the Computer Systems Lab, in a research group that focuses on system software. This is the software that allows other software to execute: It performs tasks ranging from creating program binaries out of separate source code files, to sorting and triaging automatically filed bug reports for a certain piece of software. It consists of numerous specialized tools continuously used by developers and users, throughout the SDLC (Software Development Life Cycle), mostly without them even realizing it. System software is the ideal place to integrate tools that aim to improve programmer productivity by automating some of their tasks. Over the years, our lab has performed research at the system software level, with different goals: optimizing programs for code size, extending high-level programming languages with support for accelerators that are usually programmed at a lower level, and, of course, transforming programs to protect them.

I performed my research at the system software level, and focused on protecting against arbitrary code execution and MATE attacks. My aims were improving existing protections, introducing new protections, and improving support for protections to ease their adoption. Most of my work on protections was at the *binary* or *native* level, directly transforming the binary code of a program to protect it.

1.1 Contributions

This dissertation contains four chapters based on my research contributions. These contributions were implemented at different stages in the SDLC, with differing goals and attacks in mind. All but one of my research contributions originate from the ASPIRE project. In this European FP7 project various protections against MATE attacks were developed. It was important that these protections could easily be combined on the same program, and the techniques (and the entire tool chain) were therefore designed with composability in mind. The protection techniques

were also designed under the assumptions of the attack model agreed upon by the ASPIRE consortium [115]. Next to the research chapters, this dissertation also introduces the necessary background, and draws conclusions. The remaining chapters are:

- **Chapter 2 – Background** provides the necessary background for the rest of the dissertation.

- **Chapter 3 – ΔBreakpad** presents our extension of the Breakpad crash-reporting system for diversified programs. Diversification means every user has their own, diversified version of the program. In the form of diversity we studied, all of the program binaries distributed to users differ, making it harder for attackers to scale up an attack against the program. Supporting all of these diversified programs is not easy, however, which slows the adoption of diversification. One issue is with crash-reporting systems, which automate the generating and aggregation of crash reports. Generating a crash report for a binary requires its debug information. If this information is not included with the distributed program binaries—which it often is not, for security reasons—it has to be stored server-side. For diversified programs this becomes harder, however, as every user has their own, separate version of the program, for which the debug information would have to be kept. We designed a solution to this problem, extending the Google Breakpad crash-reporting system to create ΔBreakpad. ΔBreakpad works by diversifying programs and extending them with information describing the applied diversifications. This then allows Breakpad to correctly handle crash reports from diversified programs.

  This chapter is based on:

  ΔBreakpad: Diversified Binary Crash Reporting
  Bert Abrath, Bart Coppens, Mohit Mishra, Jens Van den Broeck, and Bjorn De Sutter
  In *IEEE Transactions on Dependable and Secure Computing*, 2018

  The seeds for this research lay in an internship by Mohit Mishra. The initial design and implementation were lacking, however. To improve upon these, I almost completely redesigned his work, and re-implemented most of the diversifications and the Python framework, leading to heavily improved results.

- **Chapter 4 – Tightly-Coupled Self-Debugging** presents our self-debugging protection technique. Debuggers are often used in MATE attacks. Consequently, a program can be protected by transforming a program in such a way that it cannot be debugged. As only a single debugger can be attached to a process, the malicious use of a debugger by attackers can be countered by having the protected program debug itself. This protection is implemented by embedding a debugger component in the program during its build process, and automatically transforming the program through a binary rewriter so that it depends on the debugger, meaning the debugger cannot simply be removed. While self-debugging implementations already exist, the methods through which they make the program depend on the embedded debugger are inadequate. Our improved technique creates a tighter coupling between the program and its debugger, thus making the technique less vulnerable to automated attacks.

  This chapter is based on:

  Tightly-Coupled Self-Debugging Software Protection
  Bert Abrath, Bart Coppens, Stijn Volckaert, Joris Wijnant, and Bjorn De Sutter
  In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, 2016

  Self-debugging was developed in the context of the ASPIRE project. I contributed to the first design, and guided the initial implementation by Joris Wijnant in his master's thesis. Afterwards I enhanced this design to allow more complex code fragments to be moved to the mini-debugger, and to make the technique applicable to a broader class of environments. To realize this enhanced design and improve its robustness, I had to rework both the mini-debugger and the implementation in the binary rewriter. I also performed the evaluation.

- **Chapter 5 – Native Code Mobility** presents our code mobility protection technique, which protects against MATE attacks. Code mobility is an online protection technique where parts of the program are downloaded at run time from a trusted server, making it harder for attackers to analyze and tamper with the program. In particular, this hinders static attacks, as only part of the program is present in the binary. However, existing implementations of

code mobility either do not operate on binary code, or do not allow making only select parts of a program mobile, hindering composability with other protections at the binary level. We therefore present our own code mobility framework, implemented at a binary level using a binary rewriter. Through source code annotations, developers can specify which parts of the program are to be made mobile. The framework then splits off the pieces of native code associated with these annotations, and forms mobile code blocks out of them. This allows for finer granularity in the code blocks that are made mobile, and furthermore can be built upon to combine other protections in a more powerful way.

This chapter is based on:

Software Protection with Code Mobility
Alessandro Cabutto, Paolo Falcarin, Bert Abrath, Bart Coppens, and Bjorn De Sutter
In *Proceedings of the 2nd ACM Workshop on Moving Target Defense*, 2015

Code mobility was developed in the context of the ASPIRE project. I designed the technique in cooperation with Alessandro Cabutto from the University of East London. As for implementation and evaluation, I worked on the client side, and Alessandro Cabutto worked on the server side. Next to implementing the code transformations in the binary rewriter, I also designed the format of the exchanged mobile blocks.

- **Chapter 6 – Native Code Renewability** presents our renewability framework, built on our code mobility technique. MATE attackers have various tools at their disposal, and a broad range of attack vectors through which they can achieve their goals. Each protection technique only affects a small set of attack vectors, and applying only a few will merely divert the attacker's attention to the remaining unprotected attack vectors. Thus, multiple techniques need to be combined to ensure all these possible paths-of-least-resistance are hardened. Furthermore, it takes time for attackers to identify successful attack vectors, and subsequently use these to set up and scale up attacks. Developing new attacks is an iterative process, and repeatability is thus important: Attackers expect repeated program executions to be sufficiently similar that their attacks keep working, both during development and after distribution of these

attacks. If the target program changes while an attack is being developed, attackers have to adapt their helper scripts and perhaps re-think their approach or re-identify attack vectors. If protections can be diversified across user instances, as well as across time, this will both increase the attackers' effort in developing an attack, and decrease the window of opportunity during which this attack works and generates income for them. Our framework enables the renewing (i.e., updating) of parts of a program and the protections embedded in it, through the downloading of diversified code blocks. By continually updating not only the valuable assets the attacker is after, but also the protections defending it, we make dynamic attacks harder.

This chapter is based on:

Code Renewability for Native Software Protection
Bert Abrath, Bart Coppens, Jens Van den Broeck, Brecht Wyseur, Alessandro Cabutto, Paolo Falcarin, and Bjorn De Sutter
Submitted to *Transactions on Privacy and Security*, 2019

The renewability framework was developed in the context of the ASPIRE project, and as such employs and combines various ASPIRE protections. I built upon the code mobility technique, and made the necessary adaptations to enable the renewal of specific protections. In particular, I supervised Dimitri Vernemmen's master's thesis on semantically diversified mobile code. I guided its design and implemented supporting functionality in the binary rewriter. Afterwards I extended the design and made the implementation more robust. I also performed the experimental validation.

- **Chapter 7 – Conclusions and Future Work** draws conclusions from the preceding chapters and describes some future work.

My research also resulted in other publications that are not included in this dissertation. These publications are:

- **Obfuscating Windows DLLs**
  Bert Abrath, Bart Coppens, Stijn Volckaert, and Bjorn De Sutter
  In *Proceedings of the 1st International Workshop on Software Protection*, 2015

- **Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks**
  Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bert Abrath, and Bart Coppens
  In *Proceedings of the 2016 ACM Workshop on Software PROtection*, 2016

- **Self-Debugging**
  Bert Abrath, Stijn Volckaert, and Bjorn De Sutter
  European Patent Application EP3330859A1, 2018

Chapter 2

Background

This chapter provides the necessary background for the rest of the dissertation. Section 2.1 explains the concept and applications of system software throughout the SDLC. The attack models and existing defense techniques relevant to the dissertation are described in Section 2.2 and Section 2.3, respectively.

2.1 System Software and the SDLC

The *SDLC* (*Software Development Life Cycle*) spans the entire life of software from its conception through implementation, to release, and support. It is commonly divided into distinct phases to improve productivity. Phases such as design, implementation, installation, and maintenance, differ in their focus and in the roles of the people involved. Here, we will not so much focus on the different phases and their order, but rather on the aspects of software development that happen behind the scenes, and on the tools that improve the productivity of today's developers.

When developing software, programmers write source code in a programming language of their choice. In doing so they rely heavily on functionality already implemented by others, in the form of libraries. This can be the general functionality provided by a programming language's standard library, or more specialized functionality provided by third-party libraries. The resulting source code is then turned into a *binary*: either a *program executable*, or a *library* implementing functionality on which other executables or libraries depend. This operation of turning source code into binaries is nowadays much obscured and glanced over, but is an example of the application of system software, a class of software that is paramount to this dissertation.

System software is the software that allows other software to execute. It provides the interface between the hardware and user programs, and thus also includes a computer's *OS* (*Operating System*). In this section we focus on the system software that is continuously used throughout the SDLC, by developers and users. More specifically, we cover the specialized tools used under the hood when building, executing, debugging, and maintaining software. We limit our scope to software consisting of native code, and disregard software that is distributed in the form of source code or bytecode.

2.1.1 Building Software

The first step in building a piece of software is translating its source code into binary code that a processor can execute. The tool doing this translation is the *compiler*. It translates every separate source file into an object file, containing *machine code*, also known as *binary* or *native* code. In the second step, another tool called the *linker* will then combine these object files into the final output, which can be an executable or a *dynamic library* [77]. These two types of binary differ in their purpose: An executable is used to launch a program, while a dynamic library is a collection of functionality that an executable or dynamic library can be linked to and use at run time.

There are thus two types of linking: the *static linking* of object files and libraries into binaries which happens when building software, more specifically at *link time*; and the *dynamic linking* of an executable with the dynamic libraries it depends on, which happens during execution, i.e., at run time. Dynamic linking has some practical advantages: The functionality contained in dynamic libraries can be used by multiple executables, which saves both storage and memory space on the machine. It also has some security disadvantages. Splitting a program up into multiple, self-contained binaries means these individual binaries contain meta-information that makes them easier for attackers to comprehend and interfere with.

Compilation

A compiler takes as input a source file written in a certain programming language, and outputs an object file containing binary code for a certain *target*. Targets are a combination of: an *ISA* (*Instruction Set Architecture*), such as x86 or ARM; an *ABI* (*Application Binary Interface*), which defines

how data structures or functions are accessed in binary code; and a *platform*, such as Windows, GNU/Linux, Android, etc. To facilitate this, modern compilers generally consist of three stages: a *front end*, a *middle end*, and a *back end*. The front end is specific to a certain programming language; it verifies the syntax and semantics of the source code, and translates it into an *IR* (*Intermediate Representation*). The middle end takes this IR as input and performs target-independent analyses and transformations on it. These transformations might be optimizations (e.g., function inlining, and removing code that can never be executed), or transformations that intend to confer some kind of property on the code, such as making it more secure. After these transformations, the middle end again outputs the transformed IR. The back end is target-specific; it takes the IR as input, and after some target-specific analyses and transformations it outputs an object file containing binary code. A compiler can then have many different front ends and back ends, but a single middle end that implements the bulk of its analyses and transformations, providing its benefits to all front and back ends.

Object Files

Whereas a source file contains functions and variables, an object file contains *sections* and *symbols*. Symbols are usually the names of the functions and variables defined in an object file, and can be referred to from other object files. An object file consists of headers describing its sections, their types, sizes, and offsets within the file [77]. Some section types of interest are:

- **.text section**: containing binary code, instructions for a processor
- **.data section**: containing data such as global variables, and can both be read and written
- **.rodata section**: containing data just like the .data section, except that it is read-only
- **.bss section**: containing zero-initialized data

A `.bss` section is not actually present in an object file. As its contents are known a priori, only its size is stored in the object file.

A collection of (usually related) object files can then be gathered into a *static library*, also known as an *archive*. This forms a library of precompiled functionality that can be used in building future binaries. This functionality can then easily be included when building new binaries, without requiring further compilation.

Static Linking

A linker is given a number of object files and archives (containing other object files) that it combines into a binary. These object files contain references to each other through symbols. For example, if a function from `object_a.o` calls the function `fun_B` contained in `object_b.o`, this results in a reference from `object_a.o` to the `fun_B` symbol, defined in `object_b.o`. The linking process consists of three steps: determining all object files to link, merging their sections, and performing *symbol resolution* [77].

From the object files and archives passed to it as an argument, the linker determines all of the object files it needs to link. It starts from the passed object files, and adds object files from the passed archives that define required symbols. This is an iterative process that ends when there are no undefined symbols left. Subsequently it merges the sections by taking the identically named sections from all object files and merging them into one section with the same name. For example, all the separate `.text` sections are merged into one large `.text` section in the binary. Finally, symbols are resolved by calculating their address in the final binary, and patching/rewriting any instructions that refer to them to use this address.

The linker is the only tool that has an overview of the entire binary that is being built. This means it can do whole-program analysis, allowing for more aggressive optimizations and other transformations. The past few years, *LTO* (*Link-Time Optimization*) has been gaining traction [49, 82]. In a common implementation of LTO, the compiler outputs object files containing not native code, but IR. The linker then again combines these object files, but turns the resulting IR binary over to a Link Time Optimizer. This is usually the same compiler that generated the object files, invoked in a different way. Possessing more information and less constraints as in its original run, the compiler further optimizes the IR and has a target-specific back end output the final binary.

Another form of LTO, however, is *link-time rewriting*. Here, a binary being linked is rewritten by a tool called a link-time rewriter. Just like

a compiler, this link-time rewriter can analyze the binary code and apply transformations with different goals: optimizations for code size, execution speed, or to protect the program. Whereas the previously described LTO operates directly on IR generated from source code, a link-time rewriter starts from binary code produced by a compiler. This means a link-time rewriter is working with less information about the intended behavior of the code, and has to be be more conservative in its analyses and transformations.

One example of a link-time rewriting framework is *Diablo*, which was used extensively in my research [104]. It was originally developed for optimization, but applications for many other domains were built on top of this framework. Diablo emulates the original linking step, starting from the binary, object files, and the decisions made by the original linker. Starting from the object files and emulating the linker process allows Diablo to build a more in-depth representation of the binary and the dependencies between its constituent object files, improving its analytical power over post-link-time rewriters that only start from the binary.

2.1.2 Executing Software

When a program is launched, execution does not just start at its `main` function. There are several steps that happen before—and after—control is turned over to the actual program code. Depending on the OS and the exact manner of invocation, either a new process is created for the program, or an old one is replaced. Subsequently, the program is loaded into the address space of this process, and any dynamic linking still to be done is performed. Finally, the actual program execution begins. System software plays a supporting role in these steps.

Loading

A binary is loaded into an address space by the *loader*. The entire binary is mapped into the address space, with its sections being mapped onto pages with the right permissions. For example, a page associated with the `.text` section is both readable and executable, a page associated with the `.rodata` section is only readable. Although no space was allocated for the `.bss` section in the binary, the pages required for its size are now allocated, and subsequently zeroed out.

Dynamic Linking

It is possible for an executable not to depend on any dynamic libraries, in which case it is fully statically linked, and we call it a *static executable*. For a static executable, simply loading it into the address space is sufficient to prepare it for execution. *Dynamic executables*, however, do depend on dynamic libraries, and still require dynamic linking at run time. This dynamic linking is usually also performed by the loader. In this context of multiple binaries being loaded into the same address space, the loaded binaries are also known as *modules*.

After a dynamic executable has been loaded, the loader determines the dynamic libraries upon which it depends, and loads these. It iteratively loads more dynamic libraries—not loading any specific library more than once—until all dependencies have been satisfied. Subsequently, it resolves the specific symbols these modules require, or *import*, from each other. For every module, the run-time addresses of these imported symbols are calculated, and the associated entries in a global table are patched, so that these symbols can be accessed at run time. This symbol resolution does not *have* to happen when the binary is loaded. To improve the speed of program startup, this process can happen in a lazy manner, with every symbol only being resolved the first time it is actually used.

Program Execution

After a binary has been loaded and its dependencies have been resolved, actual code execution can begin. For both executables and dynamic libraries, the first thing to be executed are their *initialization routines*. As their name suggests, these are functions that contain initialization code, such as `C++` static constructors, for example. For executables, the `main()` function is then invoked, and the main program logic is executed. After `main()` returns—or the program exits, because of a bug or otherwise—the *finalization routines* of the modules are run, performing potential cleanup. Finally, the process ends, and its exit code is given to the kernel.

It is possible for dynamic libraries to be loaded and unloaded while the program is already executing. In these cases, the corresponding initialization and finalization routines are invoked as well.

2.1.3 Debugging Software

The goal of debugging a piece of software is finding the programming errors or bugs in it, so that they can be fixed. In order to do this, the program is usually executed in a controlled environment, under supervision of a tool called a *debugger*. Conversely, the debugged program is also known as the *debuggee*. The debugger can inspect and modify the debuggee's state at every moment, which allows one to locate the points where program behavior deviates from the programmer's intentions, and thus to find bugs.

Through its control over the program, the debugger can observe the execution of binary instructions, but often it is more interesting to know what source code these actually represent. When a user places a breakpoint at a certain source line, the debugger has to insert the actual breakpoint at the associated location in the binary code. To enable this, the debugger requires a mapping between binary code and source code, which is known as *debug information*. In this section we describe the workings of a debugger, the `ptrace` API (Application Programming Interface) which is used on Linux to debug programs, and debug information.

Debugger Operation

A debugger process has complete control over the processes it debugs. The debugger first has to request this privilege from the kernel, by attempting to *attach* to the target process. It is possible to attach to every running process, if the kernel allows it. Although there are instances where debugging an already running process is useful—such as inspecting the state of a server daemon—usually, the debugger itself starts the program it wants to debug. In this case typically no extra permissions are required to attach to the process, and the debugger controls the program right from the start.

Once attached to a process, the debugger can control its execution: stopping or continuing the process, or letting it run until a certain point in the execution is reached, by placing a breakpoint. When the debuggee's execution is stopped, the debugger can also read or change its state: It can read or write program variables, specific registers, and specific memory addresses. It can even rewrite the instructions being executed (which is a common method of inserting breakpoints). All of the debuggee's registers can be changed, including the register holding the PC (Program Counter). This means the debugger can decide at will which code is executed.

One specific way of inspecting the program's state is through its *stack trace*: a list of the active stack frames at a certain moment. This list can be created by *unwinding* the stack: determining the start and end addresses of the current stack frame, using its registers to determine the start and end of the previous stack frame, and so on. Debug information makes this process more reliable, but is no strict requirement. It *can* be used to make the stack trace human-readable though: Every stack frame has an associated function, whose name and location of variables can be found using debug information. When debugging a program a developer can then, for example, place a breakpoint to stop the process at a certain function invocation, and investigate what function invoked it (and what function invoked *this* function, and so on).

After it has fulfilled its job, the debugger can surrender its hold over the debuggee process, and detach from it. The debuggee will then continue executing normally, undebugged. If this is not what the debugger wants, it can always just end the debuggee process instead of detaching from it.

`ptrace` – Linux Debugging API

To support debugging, an OS has to provide an API that allows for all of the operations described in the previous section. For Linux, this is the *`ptrace`* API. This API consists of the `ptrace` call, the first argument of which denotes the requested action. A debugger can only request an action when the target process is prepared for this, meaning roughly that it must be in a stopped state. The only exceptions to this are attaching actions, and `PTRACE_INTERRUPT`, whose express purpose is putting the debugged process into that stopped state. Some of the possible actions are:

- `PTRACE_ATTACH`: attach to a process, and stop it
- `PTRACE_SEIZE`: updated version of `PTRACE_ATTACH`, also attaches to a process but does not stop it
- `PTRACE_TRACEME`: used by a process to request to be debugged by its parent process
- `PTRACE_DETACH`: detach from a process, and continue it
- `PTRACE_INTERRUPT`: stop a debugged process; this can only be used when the process was attached to using `PTRACE_SEIZE`

- `PTRACE_CONT`: restart a debugged process that is stopped
- `PTRACE_GETREGS`: get the register contents of a debugged process
- `PTRACE_SETREGS`: set the register contents of a debugged process
- `PTRACE_PEEKDATA`: read a word of memory from a debugged process
- `PTRACE_POKEDATA`: write a word of memory from a debugged process

Debug Information

Debug information provides a mapping between source code and binary code, for both functions and variables. For functions a mapping is provided between source lines and the binary instructions that are actually generated from these. Likewise, for variables a mapping is generated between their names in the source code and their location during execution: for global and static variables this is their address in a section, for variables local to a function this is their offset in that function's stack frame.

Aggressive compiler optimizations can affect the precision of debug information, as they make it harder to find the right connection between the original source code and the resulting binary code. Therefore, when building binaries for debug purposes, these optimizations are generally disabled.

The debug information for a binary is usually a part of the binary itself. It is encoded in a certain format, and then added to the binary in *debug sections*. When binaries are delivered to their end users, they do not necessarily include this debug information, however. Not only do debug sections make the binary larger, but there are also security considerations. A binary's debug information gives a lot of insight into how the binary works, and is thus rather valuable to potential attackers.

2.1.4 Maintaining Software

After software has been released and users run it on their system, it is maintained through the release of updates. Among others, these updates contain fixes for bugs that have been found by users running it in the real world, outside of mere testing environments. Users might file a bug report, explaining the kind of a bug it is and how to reproduce it.

Filing bug reports can require some effort and programming knowledge however. An alternative therefore is to automate this process, using an instance of system software: a *crash-reporting system*. We first introduce the general workings of crash-reporting systems, and then go into detail with a specific example.

Crash-Reporting Systems

Implementations differ, but we consider crash-reporting systems consisting of two parts: a client library included in the binary, and a centralized server. When a binary crashes, the client library embedded in it creates a crash report, and sends it to the crash-reporting server. This server then aggregates all the crash reports it receives, and can do some analysis on them such as filtering duplicates, triaging bugs based on their severity, etc. In such a system, crash reports contain things such as version information, the type of the crash, and stack traces (already described in Section 2.1.3).

Reliably unwinding a crashed process' stack to reconstruct a stack trace requires access to the binary's debug information. Even more so when a human-readable stack trace is needed. Therefore, if a crash report containing stack traces is to be sent, the binary distributed to users has to contain debug information. Debug information is valuable to potential attackers of the software, however, and takes up a lot of space in the binary. For these reasons, many binaries are distributed without it. Consequently, when these binaries crash, the client library cannot generate and send a crash report. Another option then is to have the server reconstruct the stack traces and generate the crash report. Upon a crash, the client library takes a *snapshot* of the crashed process' relevant state and sends this to the server. The crash-reporting server—which *does* possess the binary's debug information—can then use this snapshot to create the stack traces and the actual crash report. The crashed process' snapshot typically contains the CPU context and stack contents for every thread, as well as a list of the modules loaded in the process (executable and shared libraries).

Google Breakpad

One specific instance of a crash-reporting system is Google *Breakpad* [54], which was used in my research. Its operation involving three parties is visualized in Figure 2.1. When the program crashes on a user's system,

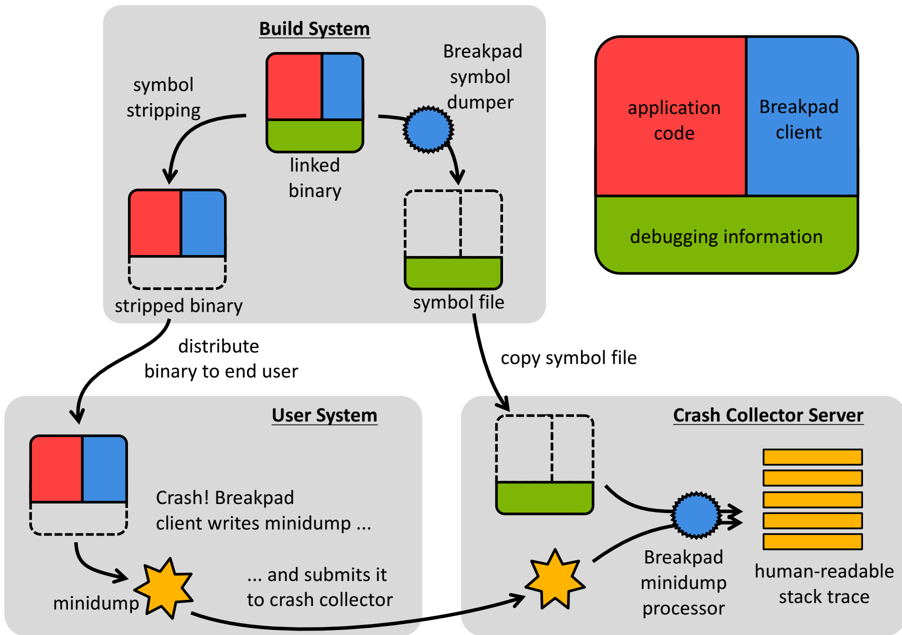


Figure 2.1: Overview of Google's Breakpad tools for crash collection (redrawn after the Breakpad website [54])

the embedded Breakpad client library creates a dump of the process state, called a *minidump*. This minidump is then sent to the crash-reporting server, also known as a *crash collector server*. On that server, a tool then combines the minidump information with the debug information stored in a *symbol file* on the server. The tool generates a stack trace, and the resulting crash report is most often first analyzed and classified automatically. If no equivalent reports are found in a database of previously received reports, the vendor's developers are notified that a previously unknown bug or previously unknown trigger has been identified, at which point they can start to study the report manually.

2.2 Attack Models

Before we can talk about defenses, we have to discuss what is being defended, and against who or what. The field of software security is broad, featuring a great many different types of attackers, differing in their goals, methods, capabilities, and targets. One example of an attacker would be a so-called *script kiddie* cheating in a first-person shooter game, using an exploit developed by a more skilled hacker to make himself invulnerable to other player's shots. Another example would be professional hackers motivated by profit, seeking to compromise a public server with valuable data they can ransom or steal. A final example would be a government agency engaged in cyberwarfare, infiltrating the computer systems of a foreign government's military facility with the goal of sabotage. The motives and capabilities of the attackers in these examples are clearly different, and so are the environments in which the attacks take place: while attacking a public server usually happens over the internet, the script kiddie might be able to cheat simply by tampering with the local version of the game. It is impossible for one defensive technique to counter all of these possible attacks, and new defenses are thus best developed with a specific attack model in mind.

An *attack model* describes the asset under attack, the vector through which the attack occurs, and the capabilities of the attackers. In this section we describe two attack models, namely those of *arbitrary code execution* and MATE *attacks*. These models cover only a small portion of the attacks possible, but are the ones that are central to this dissertation. In order to more clearly define our attack models, we first lay out some security principles.

2.2.1 Security Principles

Information is valuable. While this has always been the case, the advent of multi-user and networked computer systems forced us to reason in theoretical terms about information security: where and how it is stored, who can access it, and so on. An important concept in information security is the *CIA triangle*, based on three characteristics that give value to information: confidentiality, integrity, and availability [111]. Although the CIA triangle is somewhat outdated, these characteristics are still relevant to us. *Confidentiality* means the information can only be accessed by authorized individuals. It is thus protected from disclosure—accidental or otherwise—to unauthorized individuals. *Integrity* means the infor-

mation is whole, complete, and uncorrupted. The integrity of a piece of information is compromised when someone modifies it without authorization. Finally, *availability* means authorized individuals can access the information when required.

On a multi-user computer system these principles are enforced through *access control*: Users are authenticated before logging onto the system, and are then only authorized by the system to access those resources they are permitted to [111]. Users are limited in their permissions: They can only read or write certain files, or start up certain programs. Usually, when a program is run on behalf of a user, its permissions are those of the user. It can only access the same files the user would be allowed to. The administrator or root account is an exception: It has more privileges than other accounts, and the computer system will allow it to access whatever it wants.

2.2.2 Arbitrary Code Execution

In this attack model, the attackers want to compromise the security characteristics of an asset, without authorization from the computer system that holds the asset. They might want to gain access to a database containing credentials, compromising its confidentiality; override software providing keyless access to a car, compromising its integrity; or take down a competitor's website, compromising its availability. The attackers thus lack authorization to access their target assets, but can get around this by taking control of a process that *does* have the required authorization. The name of this attack model refers to the method the attackers use to perform the takeover: *arbitrary code execution*. The attackers exploit a *vulnerability* in a program, and so gain the ability to execute whatever code they choose in the exploited process, with the permissions of the victim process. In effect, the attackers are expanding their permissions without authorization. When this happens locally with a higher-privileged process with more permissions being taken over, we speak of *privilege escalation*. When arbitrary code execution is achieved in a process on a remote system, it is known as *remote code execution*.

To gain arbitrary code execution in a target process, an attacker first has to find a vulnerability to exploit: either a *flaw* in the program's design, or a *bug* in its implementation [111]. In this dissertation we only provide solutions for the case where the attack occurs through the exploitation of a bug. Once a bug has been found, the attacker crafts an input for the process that triggers the bug and corrupts parts of the process state. Con-

trol over the process is commonly taken over by specifically corrupting control data—such as function pointers, return addresses, the PC—with attacker-chosen values. There are many different types of bugs to exploit and control data to target, but these are beyond our scope.

2.2.3 MATE Attacks

In this attack model, just like in the previous one, attackers want to compromise the security characteristics of an asset. In this case, however, the assets are embedded in a program the attackers *are* authorized to run, as they are its end users [88]. The attackers might want to compromise the confidentiality of cryptographic keys or proprietary algorithms embedded in the program, or compromise the integrity of digital watermarks or licence checks embedded in the program. The program wants to control access to these embedded assets, denying unauthorized access. It is on its own, however. Unlike in the previous attack model, the computer system will not stand in the way of the attackers reaching their goals. It is assumed to be owned or controlled by the attackers, and indeed, can be considered complicit. As these attacks are assumed to take place by users on their own systems, they are known as Man-At-The-End attacks. There are two parts to MATE (Man-At-The-End) attacks: *analysis* and *tampering*, and their targets are compromising confidentiality and integrity, respectively.

Analysis is performed through observing a program's execution or inspecting its binary, with the aim of gaining an understanding and/or construct a representation of the program at a higher abstraction level than binary code. Analyzing and deconstructing an object—in this case, a program binary—to reveal its design is also known as *reverse engineering*. If the goal of an attack is to reconstruct an algorithm, analysis in itself might be enough to succeed.

Tampering works by making a program's execution deviate from its intended behavior, towards the behavior the attacker desires. The goal might be to avoid the execution of a licence check, to force a protected algorithm to execute out of context, or to remove a digital watermark.

Although we present them separately, differing in goals and methods, in reality analysis and tampering are used concurrently, and reinforce each other [16]. An attacker could attempt to observe program execution but be obstructed by some protection mechanism, analyze and tamper with the binary to remove the mechanism, attempt observing again, and so on.

Analysis

We consider two types of program analysis: *static analysis* consists of inspecting the binary code, analyzing it and gathering information for *all* executions; *dynamic analysis* consists of executing the program for a set of concrete inputs, analyzing this execution and collecting information from it [88]. A third, intermediary type of analysis is *symbolic execution* where the program is symbolically executed for a set of abstract inputs [71]. Symbolic execution is not considered in this dissertation. We present static and dynamic analysis separately, but the reality is more hybrid, with both being used and their results feeding into each other.

Static analysis entails converting the program's binary code into actual instructions through *disassembly*, and then structuring these instructions in a higher-level representation called a *CFG* (*Control Flow Graph*). To perform these steps, attackers can rely on tools such as IDA Pro [36] or Radare2 [93]. The resulting representation can then be used to better understand the program's workings, or even to attempt to decompile it back into source code [21]. Any debug information included in the binary comes in handy during analysis, giving away which functions start where in the binary, for example. This goes more broadly for any meta-information present in the binary, such as the symbolic information required for dynamic linking.

Dynamic analysis, on the other hand, consists of executing a program on concrete inputs and observing its execution [6]. This can be done by executing the program under supervision of a debugger, executing it in an emulated environment, or using binary instrumentation [83]. During execution, special events such as library calls and system calls can be logged for later analysis. If this is not enough, a record can be kept of all executed instructions to form a complete *execution trace*. At different moments during execution the entire memory of the process can also be dumped. All of these results are precise for those inputs on which the program was executed and analyzed, but not for all other inputs, and thus far from complete.

Tampering

Tampering can be done in both a static and a dynamic manner. *Static tampering* comes down to modifying the program binary, and changing its instructions or data. *Dynamic tampering*, on the other hand, comes down to modifying a process' state. This can again consist of modifying instructions, but also of modifying important data structures, registers,

and processor flags—or simply setting the PC where an attacker wants it to go. All of these examples are swiftly accomplished with a debugger, and it is thus no great surprise that debuggers are often used in dynamic tampering. Another method of dynamic tampering is *hooking*: intercepting function calls. The arguments of these calls can be logged for analysis, or modified. Hooking usually happens through the injection of a shared library into the process. All of the shared library's code is injected into process, and its initialization routines are executed. During initialization, the shared library can rewrite the instructions on some of the program's execution paths, and place redirects or *hooks* on them to ensure the shared library's injected code is invoked whenever these execution paths run. The shared library can then modify or log the arguments, and complete its task by returning control flow back to its original path.

2.3 Existing Defenses

Just like with attacks, there exists a great diversity in defensive techniques. Amongst others, they differ in their assumed attack models, effectiveness, and performance overhead. These techniques are applied in an automated manner by tools. There is thus also variation in where in the SDLC they are applied, and whether they are integrated into other system software, or implemented using specialized tools. In this section we only consider techniques defending against arbitrary code execution and MATE attacks, and focus on those relevant to the rest of the dissertation. This is by no means an exhaustive list, nor are these techniques usually deployed in isolation. When one attack vector is obstructed, the other attack vectors become more promising avenues for attackers. Therefore, various defensive techniques targeting different attack vectors are typically composed to form a strong, broad defense.

2.3.1 Obfuscation: Protecting Confidentiality

MATE attackers often want to compromise the confidentiality of a binary, and understand how it works at a higher level. Making it harder to comprehend a program just from its binary code or execution is therefore a valid approach to obstructing MATE attackers, and preserving confidentiality. We call this approach *obfuscation*. The idea is to take the program, and to transform it in such a way as to increase its apparent complexity, while the observable I/O (Input/Output) behavior of the

program remains the same. It is thus harder for an attacker to comprehend the program, but it still works as intended. As a side effect there might be a performance overhead: The program might run slower or require more memory.

Both code and data can be obfuscated, but here we will focus on *code obfuscation*. Many different code obfuscation techniques have been proposed throughout the years, defending against both static and dynamic analysis. *Static obfuscations* transform the program code prior to execution, while *dynamic obfuscations* on the other hand transform the program code while it is running.

Static Obfuscation

Static obfuscation usually consists of making the CFG more complex, and thus harder to reconstruct and comprehend. Two examples of such control flow obfuscations are opaque predicates [25] and control flow flattening [109]. Opaque predicates introduces new branch statements, that branch on a value that is known to be constant to an obfuscator yet hard to evaluate statically. Control flow flattening transforms a function's CFG from a logical, informative structure to a flat structure where every basic block returns to a function-wide switch block.

Program code can be obfuscated at many points during the SDLC, before it is actually run. We describe obfuscating transformations being applied on three levels: source code, IR code, and binary code. These three levels differ in their abstraction level, and thus also in what kinds of obfuscations are possible. Some obfuscations can be implemented at every level, such as opaque predicates and control flow flattening.

The source code can itself be directly obfuscated by a source-to-source rewriter. An example of this is the Tigress obfuscator [24], which operates on C code. Operating directly on source code has some disadvantages though: for one, it requires the obfuscating tool to have access to the source code, which might be an issue for some developers; for another, the tool only supports obfuscating specific programming languages. On top of that, when the compiler is handed obfuscated source code it will start optimizing it, in the process removing any obfuscations it sees through.

Operating on IR code entails implementing the obfuscations in a pass in the compiler's middle end, in effect turning the compiler into an obfuscator. After running its optimization passes on the IR code, the compiler can transform the code further using the obfuscation passes.

When implemented this way, obfuscation is supported for all programming languages the compiler supports. An example of this approach is Obfuscator-LLVM [66].

It is also possible to implement obfuscations on the lowest level of abstraction commonly available, that is, on binary code. This can be done either in the compiler back end—ideally after any back-end optimization passes—or by a binary rewriter, operating directly on the binary code emitted by a compiler. No source code is then required, but only specific target machines are supported. Some obfuscation techniques can only be implemented at binary level, such as instruction set randomization schemes, also known as emulator-based protection. Here, native binary code is replaced with custom, diversified bytecode that is interpreted or JIT-compiled by an embedded virtual machine or interpreter [61].

The freedom available at binary level also allows for techniques whose aim is not purely making the program harder to comprehend, but instead confuse the tools the attackers use. Linn et al. proposed a tool for inflating binary code with redundant and garbage instructions to defeat disassemblers [78]. These garbage instructions will never be executed and thus have no effect on the actual execution, but can confuse disassemblers and result in them disassembling incorrect code.

Dynamic Obfuscation

Dynamic obfuscation techniques work by changing the program—both its code and the execution path taken—while it is running. One method is using *self-modifying code*, where the binary code rewrites itself at run time [34, 69]. Another method is for the program binary to simply store a part of its code in an compressed (or even encrypted) form, only to decompress it at run time. This technique, known as *packing*, is often used by malware authors whose malicious code will only be unpacked and observable while the program is executing [12]. A more sophisticated version of these techniques was developed by Aucsmith [4]. His technique breaks a binary program into individually encrypted segments, so that the hash value of a block is the secret key for decrypting the next block; if the program was altered the hash value is changed and then the next block cannot be decrypted properly and the program cannot continue to run; in this case finding the first key allows recovering the full chain of keys.

All of the previous techniques have in common that the code to be executed still has to come from *somewhere* inside the binary. An

alternative to this approach is to remove a part of the code from the program and store it at a secure server, to be downloaded only when needed. The code is removed at some point during the SDLC before the binary is delivered to end users, and replaced with stubs that request its download. It is then impossible to analyzed this code statically, as it is not even present in the binary. This approach is known as *code mobility* and has been implemented in different ways by both Collberg et al. [27] and Falcarin et al. [40].

2.3.2 Tamperproofing: Protecting Integrity

Another common goal of MATE attackers is to compromise a program's integrity, subverting the program—or parts of it—to make it work to their advantage. Developers on the other hand want their program's integrity to remain intact, and have it execute as they intended. This is the goal of tamperproofing. To protect against illicit modifications, anti-tampering approaches are utilized. Methods exist that directly attempt to prevent tampering, such as instruction set randomization [61]. Here, however, we focus on methods that detect when code has been tampered with and react by stopping or delaying program execution. Such tamper-resistant software typically uses built-in integrity checks to detect code tampering. Some examples are: computing a hash over the code being executed [17], checking that the flow of control through the program confirms to the expected flow [18], or checking that functions produce the expected outputs for chosen inputs [62].

While detection and reaction are often implemented locally, they can also include a remote component. Tamper detection can be extended with a trusted server to create remote attestation [23]. The trusted server then periodically requests attestation of parts of the program, and verifies the proofs produced by the program. The reaction to tampering being detected can also be remote. For example, if the program requests data from an application server, after tamper detection the server can decide to refuse or increasingly delay any requests made by the tampered program.

A last method of tamperproofing is through code splitting techniques: These split security-sensitive parts off from client-side programs, and transfer them to be executed on secure servers instead [15]. These techniques have been combined with remote attestation: When remote attestation detects tampering, the server executing the split-off part of the program is notified and stops serving the client [105].

2.3.3 Anti-Debugging

Debuggers are often used by MATE attackers, for both dynamic analysis and tampering. One method of protecting against attackers is thus *anti-debugging*: incorporating mechanisms into the program, aimed specifically at preventing program execution with a debugger attached.

A myriad of simple anti-debugging techniques—most of which are hacks really—consist of dynamic checks to query the run-time environment for signs of active debugging [43, 84, 88, 97]. If such signs are detected, the program can shut down or degrade its execution. These techniques do not provide strong protection, however; many counter-techniques (i.e., debugger hacks) have been proposed to thwart the checks [13, 44, 84, 85].

Instead of simply checking whether a debugger is attached, it might be better to prevent a debugger from attaching at all. All major OSs (Windows, GNU/Linux, Android, OS X, ...) support only one debugger process per debuggee process. On top of that, hardware support for debugging (such as debug registers and hardware breakpoints) is designed for one debugger only (even though they might also be useful beyond that limitation). Investing in the development of effective and efficient OS support for multiple concurrent debuggers per debuggee is for the time being considered infeasible for most attackers. With these OS limitations in mind, *self-debugging* has been proposed as stronger, more complex anti-debugging technique [65, 95]. This technique works by occupying the single available debugger seat with a custom "debugger" that is launched by the protected program itself, offers no useful debugging capabilities, and cannot trivially be replaced by an attacker's own true debugger.

One method of ensuring the custom self-debugger cannot simply be detached and replaced by another debugger is provided by Nanomites [42]. Here, control flow transfers in the program are replaced by exception-inducing instructions (typically breakpoint instructions). The self-debugger is injected into the program during its build, and at program launch it is launched as well, after which it attaches itself to the program's process. Whenever an exception is then thrown, the self-debugger intervenes and implements the original control flow transfer by transferring control to the original continuation point in the program. If the debugger is detached to make room for an attacker's debugger, the program itself lacks the necessary control flow and can hence no longer be traced or debugged live.

2.3.4 Diversity

A well-studied problem in the field of security is the monoculture of software [22, 45]. When a program is released, the same binary is distributed to all users of the software, and every user then runs the same program binary. This means that when an attack is developed that works on one user's instance of the software, it is very likely to work against other users' instances. One might expect it to immediately work against all running instances, but although the distributed binaries are all the same, the associated processes running on all the different users' machines differ. They run in different environments, and might be configured differently, leading to differences in their memory layout. Depending on the attack, these differences might be substantial enough to make the attack fail on a number of instances, while it succeeds on all the others. The basic premise of *diversity* is to make all running instances of a program differ sufficiently from each other, and thus drastically decrease the probability of an attack developed on one instance succeeding on another instance.

When defending against arbitrary code execution, diversity provides a probabilistic defense: The more entropy is added to a program, the lower the likelihood of the attack succeeding. If "enough" entropy is added, the probability of success is—for all intents and purposes—zero. Attackers can overcome the added entropy through information leaks from the program, however. Similarly, diversity can provide added protection against MATE attacks: It introduces the need for attackers to customize their attack for every instance, forcing them to redo their analyses or adapt their scripts. Put in economic terms, if MATE attackers have to put more effort into scaling up their attacks, their expected profits will be lowered.

One method of automatically increasing the diversity of running instances is ASLR (Address Space Layout Randomization), where the program binary—and any libraries it might depend on—is loaded at a randomized base address in the memory [94]. The diversity introduced by ASLR is rather limited and only introduced at run time. This means developers still have to distribute only a single program binary, but also that attackers have an easier time overcoming the diversity in an automated manner, through information leaks [101]. The randomized base address of the program binary is confidential information, to which the attackers are not privy. Just a single information leak is enough to compromise this confidentiality, however. When binaries are diversified more comprehensively and more entropy is introduced throughout the binary, more information leaks are required to overcome it. One example

of such a diversification is randomly introducing no-op instructions all over the binary; these instructions do nothing when executed, but result in sufficiently different binaries.

In general it is possible to automatically add diversity to the program at all points in its SDLC: this can happen before distribution, so that every user receives his own, unique, diversified binary; or it can happen at run time, with the program diversifying itself during startup or even while running [74]. Diversifying transformations can be applied on source code, IR code, or binary code. They can be applied by a source-to-source rewriter, a compiler, a binary rewriter, or some other specialized—possibly embedded—tool.

Chapter 3

$\triangle$Breakpad

Program diversification provides a probabilistic defense, as discussed in Section 2.3.4. When academics present new and more advanced diversification schemes, industrial developers typically appreciate their protection strength, but their costs and limitations with respect to the SDLC severely restrict their practicality. Just generating and distributing N diversified versions of the program binary already incurs a significant overhead compared to distributing just a single, undiversified binary. On top of that, once these N versions are out in the real world, they need to be supported. One of the open issues with supporting diversified software is handling the associated crash reports.

Crash reports are often handled in an automated manner through crash-reporting systems. One instance of such a system is Google Breakpad, which we discussed in Section 2.1.4. Symbol files are stored on a crash collector server, allowing the distribution of program binaries that do not contain debug information. Upon a program's crash the embedded Breakpad client library sends a minidump to the server, which can combine it with the stored symbol file to generate stack traces. These stack traces are then analyzed and classified automatically.

When every user runs his own diversified version of the program, however, this system no longer works out of the box. Unless the crash collector stores symbol files for all of the diversified versions, it lacks the necessary information to identify and interpret the information in the received minidumps. According to feedback we get from developers of large, popular open source projects, simplistic solutions to overcome the mismatch between diversified minidumps and a single symbol file—such as permanently storing debug information for all diversified versions—are infeasible because symbol files are quite large. The alter-

native solution of completely rebuilding a diversified version and its debug information on the server when a crash report comes in is considered impractical as well: For larger programs, recompilation of every crashed version would be compute-intensive, and it requires the precise reproduction of the developer's build environment in the crash collection environment, which might reside on a third party's infrastructure.

Alternatively, we propose to extend the diversified minidumps with a small amount of *delta data* [75], which allows the server to overcome the discussed mismatch without requiring large amounts of persistent storage, compute power, or communication bandwidth. We present such an extension for Breakpad: ΔBreakpad. It supports crash reporting of binaries diversified with a combination of three existing diversifications. Our contributions are the following:

- An analysis of the effects of the three existing diversification schemes on x86 and ARM debug information.
- An open-source implementation of those schemes based on minimal adaptations to the widely used, state-of-the-art LLVM 5.0 compiler.
- The ΔBreakpad approach, and an open-source implementation thereof, in which Δdata bridges the gap between a diversified binary crash report and debug information from a non-diversified binary. This implementation consists of scripts that prepare and manipulate inputs for Breakpad components, but it involves no changes to the existing code base.
- Two techniques to minimize the amount of Δdata necessary to bridge that gap.
- An evaluation on a set of benchmark programs, measuring the size of the Δdata, as well as the computational cost of building and handling it.

This chapter is structured as follows. Section 3.1 provides background information, and analyzes the problem at hand for different types of CPU architectures in terms of: offset diversification schemes, debug information required for crash reporting, and the impact of the diversification on this information. Next, Section 3.2 presents an overview and detailed discussion of the ΔBreakpad approach as an extension of Google Breakpad. Section 3.3 discusses practical aspects of the diversifying tool flow implementation. The results of an experimental evaluation

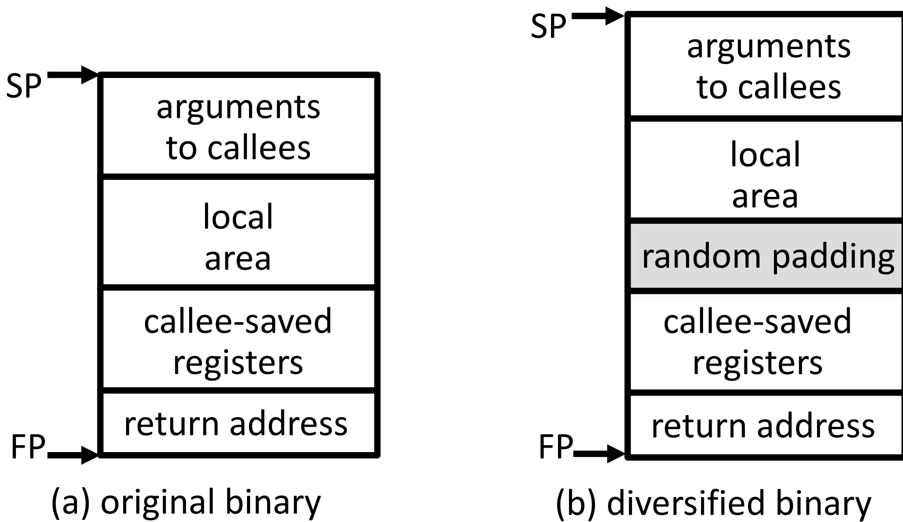


Figure 3.1: Stack frames in original and diversified binaries

are presented in Section 3.4, after which Section 3.5 discusses alternative designs and generalization issues. Section 3.6 discusses related work and Section 3.7 draws conclusions.

3.1 Background & Problem Statement

3.1.1 Offset Diversification

In this chapter, we focus on diversification schemes that alter offsets between instructions in a program and offsets between elements in stack frames. We focus on compiled languages such as C and C++ that provide no memory safety [101]. The studied types of diversification have proven to be useful on top of basic ASLR, because they raise the bar for information leak attacks: When offsets within memory segments are diversified on top of their start addresses, one leaked address no longer directly informs attackers about the locations of other potentially interesting elements. We deploy three existing offset diversification schemes:

1. **Function Shuffling** The order of all the functions in a whole binary is randomized. This randomizes inter-procedural code offsets with high entropy [19].

2. **Randomized NOP Insertion** At random locations, for some average frequency, NOPs (no-operations) are inserted into the code bodies of all the functions. This randomizes intra-procedural code offsets [60].

3. **Randomized Stack Padding** A random number of bytes is inserted in between the stack locations of buffers (present in the local area) and those of the return addresses [45]. The impact on the stack frames is visualized in Figure 3.1. It randomizes the distance from buffers to stored return addresses, as well as the distances between return addresses in different stack frames.

We do not claim that these three schemes offer the most powerful protection that diversification can offer. They do offer substantial protection, however, and as we will demonstrate, can be made compatible with crash reporting.

To implement these schemes, stochastic decision processes decide on the function ordering, on the locations to insert NOPs, and on the amounts of stack padding to insert. The stochastic decision processes are deterministic as they are based on pseudo-random number generators (PRNGs). To generate diversified code fragments, it suffices to feed the PRNGs different random seeds.

As the three schemes are conceptually simple, their decision processes do not involve checks of complex preconditions on the code fragments to be diversified. Hence no complex compiler technology is needed to replicate the decision processes, even in cases where the application of a scheme in one compilation step can trigger hard-to-predict indirect effects by triggering additional code transformations later down the compilation process. All of the necessary information to replicate them (such as function names, function body sizes, ...) is readily available in standard debug information (as will be discussed in the next section) or can trivially be generated during the compilation process, without needing to make large changes to the compilers.

A direct effect of the three schemes is that offsets encoded in the code section of a binary change. With the first two schemes, the displacements between instructions change, as does the offset of all instructions relative to the start of the code segment of the binary. In the code section, this implies changes to the PC-relative offsets encoded in, e.g., direct control flow transfers. With the third scheme, the direct changes occur in the displacements between the base pointer and SP (Stack Pointer) on the one hand, and the data items in a stack frame on the other hand. So

Description:

```
FUNC address size parameter_size name
address size line filenum
```

Example excerpt:

```
FUNC 157c 34 0 google_breakpad::LineReader::PopLine
157c 4 113 4
1580 30 116 4
FUNC 15b0 38 0 sys_close
15b0 4 2979 16
15b4 1c 2979 16
15d0 10 2979 16
15e0 8 2979 16
FUNC 15e8 5c 0 google_breakpad::PageAllocator::FreeAll
15e8 4 142 13
15ec 8 142 13
```

Figure 3.2: Source line mapping in the symbol file

offsets encoded in stack memory operations change, as do the immediate operands of instructions that produce pointers to stack-allocated data. In all three schemes, the diversification hence results in changes to offsets encoded in instructions as immediate operands. The indirect effect of those changes on the debug information depends significantly on the type of processor architecture, as we discuss in Sections 3.1.3 and 3.1.4.

3.1.2 Necessary Debug Information

The debug information of interest is embedded in the symbol files used by Breakpad. Conceptually, it consists of source line information and stack unwinding information. For both of those, the code is partitioned in regions: short sequences of consecutive instructions. The line information consists of a single list of regions. For each region, the start address, the size, and the corresponding source file and source line number are stored. In the symbol files that Breakpad uses, this information is stored in human-readable form, as shown in Figure 3.2. Each line consisting of hexadecimal numbers corresponds to one region.

The stack unwinding information also consists of a list of regions, described by their start address and size. Each region also comes with a description of the locations in the program state where the debugger's stack unwinder will find the necessary information to unwind the stack.

Figure 3.3 shows an excerpt of an ARMv7 symbol file. The postfix expressions on registers (`sp`, `r11`, `lr`, ...) express how to compute

Description:

```
STACK CFI INIT address size reg1: expr1 reg2: expr2 ...
STACK CFI address reg1: expr1 reg2: expr2 ...
```

Example symbol file excerpts:

```
STACK CFI INIT 1bdc f0 .cfa: sp 0 + .ra: lr
STACK CFI 1be0 .cfa: sp 8 + .ra: .cfa -4 + ^ r11: .cfa -8 + ^
STACK CFI 1be4 .cfa: r11 4 +

STACK CFI INIT 28a4 f8 .cfa: sp 0 + .ra: lr
STACK CFI 28ac .cfa: sp 20 + .ra: .cfa -4 + ^ r4: .cfa -20 + ^
             r5: .cfa -16 + ^ r6: .cfa -12 + ^ r7: .cfa -8 + ^
STACK CFI 28b4 .cfa: sp 904 +
```

Corresponding assembler code excerpts:

```
<function1>:  push    {fp, lr}
              add     fp, sp, #4
              sub     sp, sp, #16
              ...
<function2>:  push    {r4, r5, r6, r7, lr}
              cmp     r3, #0
              sub     sp, sp, #884    ; 0x374
              ...
```

Figure 3.3: Stack unwinding information in the symbol file

the necessary properties of the frames on the stack when execution has reached a given region. These properties are the canonical frame address (`.cfa`), the return address (`.ra`), and the values of callee-saved registers in a function's caller. The first three records in the excerpt relate to `function1`, of which the prologue's assembly code shows it has a FP (Frame Pointer) (`r11` according to the ARM EABI). The expression for `.cfa` on the first line encodes that on entry to `function1`, the SP still points to the start of the function's stack frame. The second line clarifies that after the push instruction, two callee-saved registers can be found on the stack, and the SP points 8 bytes beyond the start of the frame.

To enable the construction of a source-level stack trace on a crash server on the basis of undiversified debug information and a diversified, crashed binary's minidump, ΔBreakpad needs to be able to replicate the diversification's effect on the symbol file. Given the discussed format of that file, ΔBreakpad needs to replicate the diversification-induced changes to the number and ordering of regions, changes to their start addresses and sizes, and changes to the locations where relevant pieces of program state are stored.

We observed that in the symbol files of our benchmark suites, about 90% of the records specify line number information, and about 7% provide stack unwinding information, with the rest spent on descriptions of the files and paths, and on the interfaces that are exported. Those 7% do occupy about 20% of the symbol file size, however: As can be seen in Figures 3.2 and 3.3, stack unwinding records are much longer than code/line region records.

3.1.3 Indirect Effects in x86 Binaries

On variable-width CISC architectures such as Intel's x86, the indirect effects of the three diversification schemes are mostly limited to additional changes in the displacements between instructions. When, as a result of a changed offset, less or more bytes are required to encode that offset as an instruction's immediate operand, the x86 compiler will simply generate another form of the same instruction that uses less or more bytes. In addition, as the compiler might put certain instructions on specific alignments to optimize instruction fetching or instruction caching, it might insert different amounts of padding as a result of the diversification. Such changes only alter the addresses and sizes of regions in the symbol files.

More or less the same happens as a result of the randomized stack padding. In many functions, no instructions are present in the function prologues/epilogues that only increment/decrement the SP. To allocate/deallocate the additional randomized padding in such functions, additional instructions have to be inserted in the prologue/epilogue. In the symbol file, this comes mostly down to splitting regions in the stack unwinding information: one region before the SP increment/decrement, and one region after it.

Consequently, replicating the effects of a diversification on the debug information stored on a crash collector requires updating the number, addresses, and sizes of regions, as well as the offsets where relevant state is stored in stack frames. To do so, it suffices for the crash collector to have (i) the original, undiversified binary including its debug information; (ii) a script that replays the deterministic decision processes of the randomizing diversification schemes; (iii) the seeds and keys that were used for generating the diversified binary.

We thus conclude that on architectures like x86 it suffices to embed the seeds and keys in the binaries, to extend the Breakpad client to send them along with the minidump to the crash collector, and to extend the

Breakpad minidump processor to let it replicate the impact of the diversification process on the symbol file. For that replication, the complete original compiler is not required. Instead, a simple script suffices that replays the stochastic diversification decision processes for the program at hand, i.e., taking into account the alignment requirements of the individual program fragments and the locations where different types of offsets are encoded in the code. A complete approach that covers these features and more is presented in Section 3.2.

3.1.4 Indirect Effects in ARMv7 Binaries

On architectures like the ARMv7 RISC architecture, the situation is quite different.[1] The same changes occur, e.g., with respect to the function prologues and epilogues, but there are three extra factors causing many more indirect effects.

Fixed-width instruction encoding. ARMv7 instructions are 16-bit or 32-bit wide. The immediate operands of ALU and LD/ST instructions can therefore only be quite narrow. Thus, when offsets grow bigger because of diversification, it can become impossible to encode them as immediate operands. Instead, the offsets then have to be stored in registers. This requires additional instructions and puts extra pressure on the register allocator, which can result in instructions becoming scheduled in different orders. In fact, we have observed that if the same offset has to be generated multiple times, the compiler sometimes applies common-subexpression-elimination [86], which can have a global impact on register allocation and instruction scheduling. Furthermore, we have observed that the compiler sometimes changes the base register used in LD/ST instructions, e.g., when the offsets of a location in the stack frame relative to the SP and/or the FP change.

Rotating immediate operands. The ARMv7 architecture has a peculiar way of encoding offsets as 8 consecutive bits that can be rotated over a 5-bit amount. It therefore also happens that offsets that could not be encoded as immediate operands in the original binary become perfectly fine ones after diversification. For example, whereas an original offset 0x3ff0 cannot be encoded in one immediate operand, it does work perfectly fine for the increased offset 0x4000 that can result from adding stack frame padding.

[1] The 32-bit part of the ARMv8 architecture, which is still omnipresent on mobile devices, is mostly identical to ARMv7.

The visible PC. ARMv7 code typically contains a sizable amount of PC-relative computations, both in position-independent and in position-dependent code. The reason is the visible PC. Constant values that cannot be encoded in individual immediate operands, such as vectors of numerical values to be used by vector instructions, and constants unknown at compile time, such as absolute addresses or inter-modular offsets, are often loaded from so-called literal pools: data chunks interspersed with the code, accessed through PC-relative load operations. As our diversification schemes can change the sizes of code fragments, and as only narrow offsets can be encoded, they also affect the location where the compiler injects the literal pools in between the code. Whereas the order of instructions and literal pools can remain the same when NOPs are inserted randomly in x86 code, it cannot remain the same in ARMv7 code.

In conclusion, when targeting an architecture like the ARMv7, we have to expect much further reaching changes to the code section, even if we only apply our three relatively simple offset diversification schemes. Moreover, on such an architecture it is impossible to replicate the changes to the corresponding symbol file completely without replicating part of the compiler infrastructure that was used during register allocation, instruction selection, and instruction scheduling. In other words, it cannot suffice to put a simple script on the crash collector server to replicate the impact of the diversification on the symbol file.

3.2 The ΔBreakpad Approach

To overcome this problem, ΔBreakpad combines three main concepts. The first concept is *imperfect replication* of the diversification process' impact on the symbol file. The second concept is *patching* of the imperfect replication result to make it perfect. The crash collector will not only receive the necessary seeds and keys to replicate the diversification decision process, but also a patch that will allow it to fix any imperfections in the performed replication. Next to the minidump, the seeds, and the keys, the ΔBreakpad client thus also has to send the patch to the crash collector. The third concept is Δ-*minimization*, with which we denote the adaptation of the compilation and diversification process to minimize the sizes of the patches that the client has to send to the crash collector.

Figure 3.4 presents an overview of the ΔBreakpad approach. It looks much more complicated than the Breakpad overview in Figure 2.1, but the main Breakpad components are still present, and are in fact reused

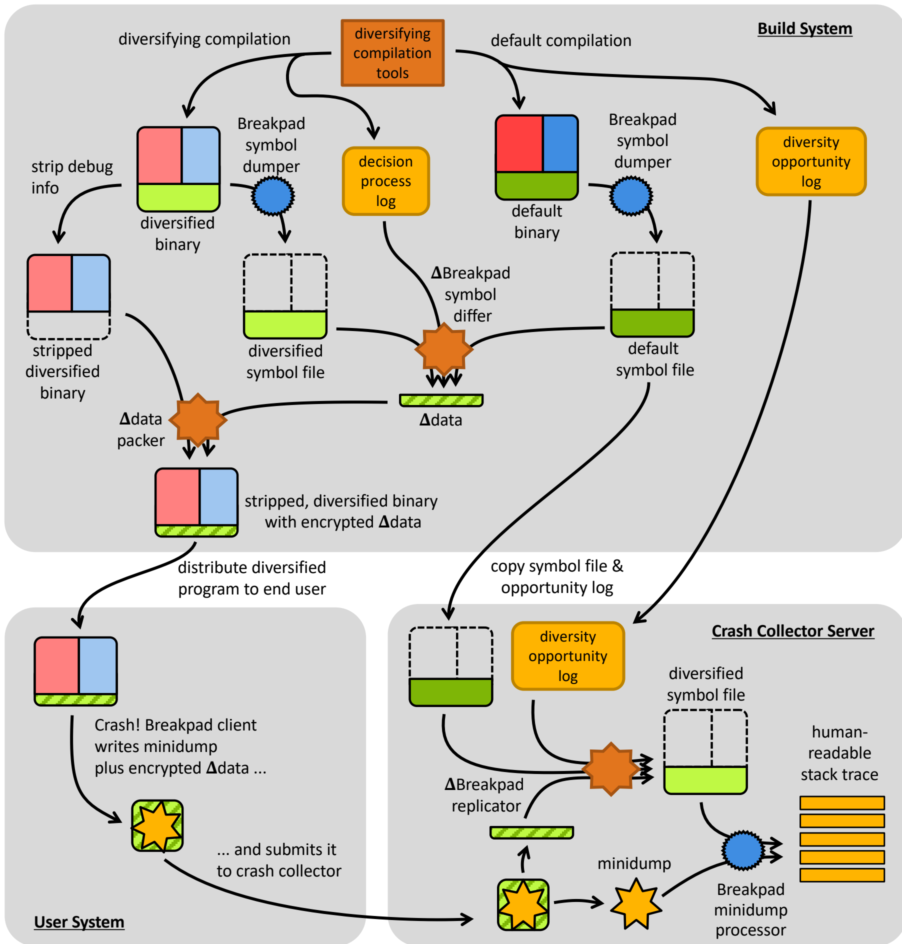


Figure 3.4: Overview of ΔBreakpad as an extension of Google Breakpad. The Breakpad symbol dumper and the Breakpad minidumper are reused as is from the standard Breakpad as shown in Figure 2.1.

as is: ΔBreakpad consists of scripts and unmodified existing Breakpad tools. As we will discuss in Section 3.4, it requires only minimal changes to the build system tools to generate the diversified binaries and Δdata.

3.2.1 Crash Handling & Stack Trace Generation

Importantly, the ΔBreakpad approach does not require any change to the minidump that is sent by the client to the server. The minidump file format as developed by Microsoft is similar to core dump files, but much smaller, better documented, and less OS-specific. A minidump contains:

- A list of the executable and all shared libraries loaded into the process when the dump was created.
- A list of the process threads, with their stacks and processor register contents. Complete stacks are included because the applications typically do not contain debug information to analyze the stack.
- Some more system information, including the processor and OS versions, as well as the reason for the crash.

We only need adapt the Breakpad client such that it sends the server a small chunk of Δ*data* along with the minidump (bottom right of Figure 3.4). This does not require any patch to the Breakpad library (`https://github.com/google/breakpad/`) that is to be linked into an application to enable Breakpad crash reporting. That library is only responsible for dumping the necessary information about a crash to disk. A separate process is then responsible for sending the data to the crash reporter. This isolation minimizes the risk that Breakpad's operation is corrupted by the trigger of the crash (e.g., buggy code being executed). The separate process needs to be implemented and customized for every OS and usage scenario. For ΔBreakpad, we only need to customize it some more to let it deliver the Δdata with the minidump. That Δdata contains the random seeds, keys, and other parameters that the server needs to perform the imperfect replication, as well as the aforementioned patch. If necessary, the Δdata can be encrypted and signed to guarantee authenticity, integrity, and confidentiality.

The crash collector server still persistently stores debug info in the form of a single symbol file of the *default binary*. No changes to its format are required, so the existing Breakpad symbol dumper utilities for the major OSs can be reused out of the box to extract the necessary information from the DWARF or STABS debug sections in ELF object

files or from stand-alone PDB (Microsoft's Program Database format) files.

In addition, the server persistently stores a *diversity opportunity log*. This log is generated during the *default compilation*, i.e., when the diversifying tool chain is invoked without applying any actual diversification to generate the default binary. It lists all the opportunities for diversification that occurred during the generation of that binary, but that were not exploited. For example, it lists all the program points where the diversification process considered but skipped inserting NOPs. An essential feature of the diversity opportunity log file is that it lists (i) all decision points where, during an actual diversifying run of the tools, random numbers are drawn from the PRNG; (ii) the necessary information for determining the diversification options from which one is selected with each drawn random number.

When a crash report arrives on the server, the *ΔBreakpad replicator* replicates the impact of the diversification process on the symbol file in a couple of steps. First, the replicator extracts, decompresses, and (optionally) decrypts the Δdata.

Next, the replicator extracts the seeds, keys and possible parameters from the Δdata, to replicate the impact of the diversification decision process on the *default symbol file* by means of the opportunity log. The replicator initializes a PRNG with the same parameters and random seeds that were already used on the build system for the actual diversification of the binary from which the crash report was achieved. The replicator then draws random numbers from that PRNG at each point where the original diversification process had already drawn numbers. For each drawn number, the replicator then adapts the content of the symbol file to replicate (approximately) the impact the original diversification step had caused on that file. The overall result is an approximation of the *diversified symbol file*, i.e., the symbol file that the original Breakpad symbol dumper tool had produced on the build system for the *diversified binary*. It is an approximation because the replicator only models direct effects of the diversification, such as increased region sizes resulting from inserted NOPs, but no secondary effects like the ones discussed in Section 3.1.4. So finally, the replicator extracts the patch from the Δdata and applies it to the approximation, thus reproducing an exact copy of the diversified symbol file.

As the contents of that diversified symbol file match the contents of the received minidump, the existing Breakpad minidump processor can then be used to produce the human-readable stack trace. Notice

that this stack trace only contains information at the abstraction level of the source code. Crashes occurring in corresponding regions in differently diversified versions of the binaries will hence produce exactly the same stack trace. As such, all existing manual or automatic tools and techniques to analyze and classify the stack traces, e.g., for triaging, still work out of the box.

3.2.2 Generating the $\triangle$data

The top part of Figure 3.4 shows the adapted build system. On the right, the standard Breakpad symbol dumper flow is shown to generate the default symbol file, to be stored persistently on the crash collector server. This symbol file is extracted from the default binary. On the left, the diversified binary is generated, along with the *diversified symbol file*, and the *decision process log*. The latter consists of the same info as the opportunity log, plus a description of the actual result from the applied diversification. Using this log and both the diversified and default symbol files, our $\triangle$*Breakpad symbol differ* then generates the $\triangle$data, in particular the patch part of it. Finally, the $\triangle$*data packer* compresses, and optionally encrypts and signs the data and injects it as an additional section into the stripped diversified executable. The resulting binary is then distributed to the end user, ready to be executed and crash.

3.2.3 Combining Multiple Diversification Processes

In order to make the described approach work, we need to ensure that the replication of the decision processes on the crash collector (on the basis of the opportunity log generated for the default binary) stays synchronized with the decision process as it was executed during the generation of the diversified binary. This is non-trivial when one wants to apply multiple forms of diversification one after the other. Because of the already discussed indirect effects of diversifications, the replication process does not know the exact outcome of an earlier diversification applied to some code fragment. The replication process hence does not know the exact form of the code fragment onto which the later diversification is applied.

For example, consider the design where randomized padding is injected into a function's stack frame first, and random NOPs are inserted in its code body afterwards, after instruction scheduling has been performed. Given the ordering of compilation phases in a compiler, this

is a reasonable design [86]. As discussed in Section 3.1.4, the injected padding can cause changes in the number of instructions in the function body. If this actually happens, and if the later NOP insertion process draws a random number for each instruction in the code to decide whether or not to insert a certain number of NOPs after that instruction, the replicator will draw more or less random numbers from the PRNG than were counted during the generation of the default binary.

Fundamentally, the problem is that the diversifying NOP insertion is then performed on code that differs from the code from which the opportunity log was constructed. So in that case, the replication of the decision process on the crash collector will at some point become desynchronized with how the actual diversification was decided. Unless special care is taken, this will result in completely diverging replication from that point on, which can only be compensated by including a huge patch in the Δdata.

We avoid this in two ways. First, the decision processes of the combined diversification schemes need to be carefully designed to become mostly, if not completely independent. In our diversifying tool chain, we achieve this by applying the later decision processes at a granularity of code fragments that is not likely to be impacted by earlier decision processes. Trivially, the order in which functions are shuffled is completely independent from the number of NOPs inserted in them, as well as from their stack padding size. We also observed that although random stack padding and NOP insertion often result in changes in the number of instructions in the function bodies, in particular when the ARMv7 architecture is targeted, they rarely impact the structure of the functions' CFGs. The few cases in which we did see changes to the CFGs are the following:

- When trampolines had to be inserted or could be removed as a result of changed code displacements.
- When basic blocks became so big or small that they (no longer) had to be split, e.g., to provide space for a literal pool.
- When heuristics used by the compiler consider the sizes of the involved fragments. For example, in the LLVM compiler, we observed that the *tail duplication* optimization considers code size (small blocks are duplicated more), as do *if-conversion* and *tail merging*.

Randomized stack padding and NOP insertion can hence impact the CFGs of functions. Importantly, the effects of the mentioned transformations do not escape functions, as the transformations are intra-procedural.

Whereas NOP insertion inherently changes the sizes of code fragments, stack padding changes them much less frequently. We build on this observation by performing the stack padding insertion first, followed by the NOP insertion, of which the decision process is performed basic block per basic block, with a re-initialization of the used PRNG before each block. So however the number of instructions in the basic blocks are impacted by the former two diversification steps, as long as the CFG of a function is not impacted, the replicator's decision process will remain synchronized automatically. Function shuffling is applied last.

Our second way deals with the above cases where a function's CFG is actually changed as a result of the first two diversifications. As function shuffling has no impact on the function bodies, such changes come only from stack padding. In such cases, we accept the desynchronization, but we contain it to the function whose CFGs is changed, i.e., to that function's part of the symbol file.

To avoid that the resulting desynchronization in the replication spills over into other functions, the tools that perform the diversification and the imperfect replication resynchronize the used PRNGs upon entry to a function. Such resynchronization per function can be implemented in several ways. Hierarchical PRNGs are one option, whereby the top-level PRNG is invoked on entry to each function. In our tools, we alternatively reset the PRNG with a new seed value that is computed by hashing a unique, immutable identifier of the function combined with the diversification seeds and keys. With cryptographically strong hash functions, the new seeds cannot be predicted by attackers unless they know the (global) diversification seeds and key. As a unique function identifier that will not be impacted by any diversification step, we use the concatenation of the (mangled) name of the function, the name of the object file from which the function originated, and the name of its section within that object file. By compiling code with the `-ffunction-sections` flag, these identifiers are guaranteed to be unique. Every function is then put into its own section in the generated object file, and that section name then includes the function name, even for functions that are themselves anonymous in the object file, such as C functions declared `static`).

3.2.4 Δ-Minimization

We want to demonstrate that crash reporting for diversified software is feasible with limited overhead. Therefore, we explored several ways of minimizing the Δdata. A first option to reduce the size of the Δdata is to compress it or to use more efficient encodings for the information that needs to be stored in the Δdata. Compression and coding are not the focus of our work, however, so we will simply rely on existing compression schemes to compress information encoded in a custom developed, but likely suboptimal coding scheme. A second option is to adapt the processes that perform the compilation and diversification. Those processes have an impact on the amount of imperfection in the replication, i.e., on the Δ between the diversified symbol files and the symbol files reconstructed through imperfect replication. Those processes can hence be tweaked to minimize that Δ, which will in turn lead to a reduction in the amount of patching information needed in the Δdata. Tweaking these processes is the option we explore in this section.

We opt not to achieve a smaller Δ at all cost, however. Apart from the restrictions discussed in Section 3.2.3, we do not want to impose strict limitations on the freedom with which to apply the diversification schemes. For example, when we let a compiler select a randomized amount of stack padding for some function, we do not want to restrict its selection to values that preserve the exact instruction schedules in the function body. Besides helping us to keep the diversification process decision logic (in the compiler as well as in the replicator) independent of compiler internals, this ensures that the entropy generated by means of the diversification does not depend more than strictly necessary on artifacts of the code being diversified. From the perspective of security, this is obviously an advantage. Furthermore, we want to limit the changes we need to make to existing compilers and related tools used for generating and/or diversifying the binaries. What remains then to reduce the Δ, is the selection of the default compilation strategy and a minimal set of adaptations to the compilation tools to enforce that strategy. For the three forms of offset diversification we deploy, we identified two tiny but very useful adaptations.

Adaptation 1: Default Stack Padding

The first adaptation is that 8 bytes of stack padding are added in every function in the default, non-diversified binary. During the diversification process itself, every function gets a randomized number of padding

bytes that is a strictly positive multiple of 8. This adaptation enforces the insertion of padding operations in all function versions, i.e., default ones and diversified ones. It therefore limits the number of cases where the code regions of the function prologues and epilogues as listed in the default symbol file need to be split to match the regions in the diversified symbol file (as discussed in Section 3.1.3).

The default padding enforces the inclusion of instructions to allocate and deallocate stack space in the function prologues and epilogues: the single prologue then contains one `add sp, sp, #const` instruction (or multiple ones, if the size of that stack space, i.e., the `const` value, cannot be encoded as a single immediate operand), and each copy of the epilogues contains one (or more) `sub sp, sp, #const` instructions, both in the default program version and in the diversified versions. Without the default padding, many functions in the default binary would not contain such SP incrementing/decrementing instructions. For those functions, the default padding minimizes the differences between default and diversified code and their corresponding regions in the symbol files.

For functions that already allocate and deallocate stack space in the default binary, adding default padding is useful as well. We observed quite some functions where the local area of a stack frame only holds relatively large arrays whose sizes are powers of two. In those functions, the aforementioned `const` operands are large values of which the least significant bits are all zeroes. Those values can hence be encoded as single immediate operands in the ARMv7 and similar architectures. By adding another 8 bytes of padding, a lower bit becomes set as well. The value can then no longer be encoded as a single immediate operand in the default binary, just as it will likely not be encoded as a single immediate operand in the diversified binaries, where a randomized (but still relatively small) amount of padding is added. The average difference between the default binary and the diversified binaries, and hence the average amount of information to be stored in the Δdata, is thus reduced. For other functions, such as those with small local areas, the added 8 bytes typically do not impact which offsets can be encoded as immediate operands. The added 8 bytes then do not offer any benefit, but they also do not hurt in any way.

Minimizing the differences that randomized stack padding introduces between default and diversified code fragments is particularly important for the function epilogues; not only to make the corresponding regions in the symbol files more similar to one another, but also to limit indirect effects on the generated code. As a result of the default

padding, the epilogues in a function typically have the same size in the default binary and in the diversified binaries. Maintaining the same size for epilogues throughout the stack frame diversification is important for Δ-minimization because the size of basic blocks, which is the form under which epilogues occur in the diversifying compiler's intermediate code representation, plays a significant role in the heuristics that steer some compiler optimizations, as discussed in Section 3.2.3. As a result, the insertion of extra instructions in the epilogues can result in altered CFGs. The introduction of default padding reduces the occurrence of such alterations. For the interested reader, Appendix A provides a quantitative analysis of this effect. In any case, reducing the number of alterations in the CFGs reduces the number of desynchronizations during the imperfect replication, thus minimizing the required Δdata.

The 8-byte padding in the default binaries has no impact whatsoever on the size or on the performance of binaries distributed to end users: The default padding only influences the default symbol files and the Δdata that will be used to reconstruct the diversified symbol file. With respect to security, there is only a small impact on distributed software versions. By excluding the possibility of adding zero bytes of stack padding to a function, keeping only the values 8, 16, ..., 256, we reduce the entropy in the stack frame layout of the diversified binaries from $\ln(33)$ to $\ln(32)$.

Note that this 8-byte padding in the default binary can be implemented trivially in a diversifying compiler that already injects randomized stack padding: Default stack padding simply comes down to executing the diversified stack padding code with a non-diversified amount.

With respect to correctness, we note that by making all diversifying padding multiples of 8 bytes, the padding does not affect the natural alignment of data in stack frames. Typically, that data needs 8-byte alignment or less. This is reflected in the ABIs we know of, and which impose at most 8-byte alignments. If data in a stack frame needs stricter alignment, e.g., because vector instructions will operate on wider data that needs 128-byte or 256-byte alignments, special constructs need to be used in the code that achieve such alignments independently of the address at which the stack frame starts. Such constructs include the use of `alloca` or the allocation of a bigger array than needed and then using only an aligned part in that array of which the starting address is computed at run time. As such constructs function correctly at whatever allowed stack frame address, i.e., at any 8-byte aligned stack frame

address according to the ABIs, those constructs survive the addition of randomized amounts of padding that are multiples of 8 bytes.

One can wonder whether the correctness of special programming constructs such as tail recursion can be affected by stack padding. We conjecture that this is not the case when the padding is implemented correctly. For example, we implement the stack padding insertion by simply asking the compiler to reserve space for more local variables on the stack as if more local variables were declared in the source code of the functions. The correctness of the padding then comes down to the correct implementation of the existing stack frame allocation in the compiler. As that allocation is a crucial aspect of any compiler, we can rely on its correctness.

Adaptation 2: SP/FP-Relative Access Optimization

The second adaptation consists of disabling a minor optimization in the (ARM-specific) compiler back end. When a function has a FP, the compiler back end can choose to access data in its stack frame via FP-relative LD/ST instructions or via SP-relative ones. The decision can take into account the offsets of the data relative to the FP and to the SP. By choosing the option by which the offset can be encoded in one immediate, rotating operand (as discussed in Section 3.1.4), the code can be optimized.

After disabling that optimization, the compiler alternates less between FP-relative and SP-relative addressing as a result of randomized padding. The diversified binaries therefore become more similar to the default binary, which ultimately results in smaller Δdata. Appendix A backs this up with quantitative data for the interested reader.

This adaptation is trivial to implement: In LLVM, a one-line edit (to a condition in an if-statement) suffices. However, unlike the default stack padding, this tweak does potentially impact performance. In the SPEC2006 C and C++ benchmarks in our benchmark suite compiled with `-O2`, we observed no significant average performance impact: The average execution times increased with the rather small amount of 0.34%. For individual benchmarks, disabling the optimization resulted into anything between a 0.86% speedup and a 2.70% slowdown. These effects are likely caused by accident, such as improved or worsened instruction cache behaviors that accidentally result from small code changes, i.e., unintentional and beyond the scope and awareness of the compiler's optimizations [87]. Still, these numbers indicate that there can be a small

effect, that the software developer in certain performance critical cases may want to trade-off against the potential benefits in terms of Δdata size. The latter is evaluated in Section 3.4.

With respect to security, this adaptation has no impact: The offsets in the stack frames do not change because of this optimization, and hence the entropy resulting from the offset randomization is not impacted. With respect to correctness, this adaptation has no impact either: We only let the compiler skip the exploitation of an optimization opportunity. In cases where the transformation implementing the optimization would be mandatory to generate correct code in the first place, it can of course still be applied as is. We know of no such cases, however.

3.2.5 Profile-Guided Diversification

Some diversification schemes can benefit from profile information to reduce the overhead. For example, the performance overhead of NOP insertion can be reduced by concentrating NOPs on infrequently executed program points [60]. ΔBreakpad supports such profile-guided diversification: As long as both the default compilation and the diversifying compilation runs are served the same profile information, the decision process logs and the diversity opportunity log will be consistent with each other, so the ΔBreakpad replicator will work just fine.

3.3 Prototype Diversification Tool Flow

As we want to demonstrate that our approach can work with small Δdata sizes even on architectures that are harder to target, we evaluated it on the more challenging ARMv7 architecture. In particular, our prototype tools support the 32-bit subset of the ARMv7-A architecture (i.e., excluding 16-bit Thumb and Thumb2 code).

Diversification processes can be applied at many stages during the SDLC [73]. In our prototype implementation, the three diversification schemes are applied when the binaries are built. The schemes are applied in the already discussed order using existing open-source compiler tools.

3.3.1 Stack Padding

First, we adapted LLVM 5.0 for randomized stack padding. All functions get a random stack padding between 8 and 256 bytes, but always a

multiple of 8 bytes, as discussed in Section 3.2.4. The amount of padding for each function is determined by hashing the function's (mangled) name. The diversification seed is the key to the hash function. In this stateless scheme, the amount of padding in each function is independent of the order in which functions are compiled. This further eases the replay on the crash server, for which all the necessary function names are already present in the default symbol file.

Our LLVM patch to implement the stack padding and related command-line options is 41 lines of code in total. The stack padding itself is implemented in the architecture-independent code of the LLVM compiler pass that inserts function prologues and epilogues. Amongst others, that pass determines the total size of each function's stack frame, including the space needed to implement calling conventions. Our patch extends that computation to insert randomized stack padding.

On top, a two-line patch sufficed to disable the SP/FP-relative stack access optimization discussed in Section 3.2.4.

3.3.2 NOP Insertion

We further adapted the LLVM 5.0 ARM back end to perform randomized NOP insertion and to generate an opportunity log, implementing a decision process as discussed in Section 3.2.3. It inserts a NOP in between every consecutive pair of instructions in a basic block with a user-controlled probability. For our experiments, we set this probability to 20%. More complex schemes, that introduce more entropy in the offsets between individual instructions in function bodies can easily be envisioned. Introducing many more NOPs will likely not be acceptable, however, as it obviously inflates the code size. As long as the more complex schemes have a decision process along the lines of the one discussed in Section 3.2.3, with a fixed number of random numbers drawn per basic block, we conjecture that the Δdata size will not be impacted significantly.

To minimize side effects that would lead to inflated Δ-data, NOP insertion is done as late as possible in the compiler back end. The new NOP insertion compiler pass is invoked after instruction selection, if-conversion, instruction scheduling, register allocation, peephole and other assembly-level optimizations, and code layout; and right before the very last LLVM ARM code generation pass that inserts literal address pools and the necessary trampolines. As already discussed in Section 3.2.3, that last pass can only be executed while all the basic

blocks sizes are being finalized: Trampoline insertion and literal address pool insertion leads to code size increases, which might necessitate additional insertions, so they are performed iteratively until a fix-point is reached. From then on, no extra insertions can be performed (without risking having to undo and redo the insertion of pools and trampolines).

To replay the NOP insertion on the server, the opportunity log lists the functions' code and data blocks, as well as their sizes. The data blocks include blobs of data that the compiler stores in the code section (for various reasons) as well as the literal address pools. Those blocks are marked as data, such that the NOP insertion replay knows to skip them, i.e., not to insert NOPs in them. The code blocks correspond to the basic blocks in the compiler's intermediate code representation. To enable the inclusion of all the necessary information, in particular with respect to literal address pools, the opportunity log is generated at the end of the trampoline and address pool insertion compiler pass.

Since the number of instructions per basic block can be different in a diversified binary as a result of stack padding, the data in the opportunity log allows for relatively accurate, but not perfect replay on the crash server. The difference is obviously covered by the patch in the Δdata.

Despite our careful design to obtain accurate opportunity logs, we observed that in some cases, the logs are not completely accurate. When source code contains inline assembly fragments, the LLVM code generator handles those mostly as strings, of which it estimates the maximal code sizes to insert trampolines and literal address pools as necessary. Most often, those estimates are correct. Occasionally, however, LLVM overestimates their actual size. This results in desynchronization during the NOP-replication, because the replication then inserts NOPs in later blocks at incorrect addresses, resulting in incorrect updates to the supposedly corresponding regions in the symbol file.

Fortunately, this form of desynchronization occurs infrequently. Most user-space application and library code (except for the standard system libraries) does not include inline assembly. In our experiments, only the injected Breakpad components contained inline assembly. For all but the smallest programs, those components make up only a tiny fraction of the whole binary. Moreover, the desynchronization ends at the function boundary, when global resynchronization is performed anyway. So the overall impact on the sizes of the Δdata is minimal.

We conjecture it is possible to eliminate this completely by engineering a way in which incorrect estimates in the opportunity log are patched

on the basis of an inspection of the actual assembler code generated during the default compilation. This engineering task is left for future work.

Another source of errors in the NOP insertion replay, and desynchronization, is the insertion of the NOPs themselves. These can cause the location of the data pools inside the function to change, or even cause the sizes of these pools to change. This form of desynchronization happens rather infrequently.

Our LLVM patch to implement the NOP insertion technique and related command-line options is 148 lines of code in total, 60 lines of which are used for outputting the opportunity log.

3.3.3 Function Shuffling

We use the standard GNU linker for shuffling functions. In preparation for this, we use the `-ffunction-sections` compiler flag to ensure that the compiler puts each function into a separate code section in the generated object files. To perform the actual shuffling, we simply generate a custom linker script that enforces a shuffled order of all the code sections, and hence of all functions. The order is determined with a pseudo-random number generator that is seeded with the diversification seed.

This process builds completely on existing linker functionality. No patch to the linker source code is needed to let it generate the diversified function orders. For generating the linker script, we extract all the linked-in functions from the linker map file. All linkers we know can produce such a file, which basically documents how the original (i.e., default) linker script was executed on the linked objects.

To replay the shuffling accurately on the crash server, the information extracted from the linker map file is needed, i.e., the names and sizes of linked-in functions, as well as their alignment requirements. These can be obtained from the linker map file and from the object files generated during the default compilation: The alignment requirements of functions correspond to those of their corresponding code sections in the object files. Those section alignment requirements are explicitly encoded in the object files to allow correct linking. We extract them to include them in the opportunity log. During the replay, they are useful to predict the amount of padding that needs to be inserted before each function in the diversified binary, such that that amount of padding does not need to be included in the Δdata.

3.3.4 Δdata

The uncompressed Δdata our tools generate contain human-readable ASCII text. With more engineering, smaller patch sizes can likely be obtained, so the (compressed) Δdata sizes we report in the next section only put an upper bound on what could be achieved with a more fine-tuned implementation. If authenticity, integrity and confidentiality are required for the Δdata it can also be encrypted and signed. This obviously adds some extra data. For example, when we experimented with GPG (GNU Privacy Guard, `https://www.gnupg.org/`) to encrypt with AES256 and sign using the SHA-1 hash and RSA, we observed that the Δdata grows with 354–356 bytes (depending on the needed padding).

3.4 Experimental Evaluation

3.4.1 Benchmarks and Correctness

For evaluating our approach and the correctness of our implementation, we use the C and C++ programs from the SPEC2006 benchmark suite. We evaluated the approach on dynamically linked binaries, all of which also include the Breakpad client next to the actual code. The dynamically linked, position-dependent binaries were compiled at optimization levels `-O1`, `-O2`, `-Os`, and `-O3`. For all four levels, we evaluated two versions: with and without the `-fomit-frame-pointer` option. So in total, we evaluated the benchmarks on eight compilation flag combinations.

For each of those eight combinations, we diversified the benchmarks using 30 tuples of three random seeds, one for each diversification scheme we implemented. All diversified versions compiled and executed correctly with our patches and three-step diversification. Consequently, our diversification implementation can be considered validated.

To validate the correctness of ΔBreakpad's crash reporting, we verified that the diversified symbol files generated with our server-side replicator on the basis of undiversified symbol files, the opportunity log, and Δdata are equivalent to symbol files obtained directly with the symbol dumper from the debug info in the diversified binaries.

3.4.2 Overhead[2]

We evaluated the overheads introduced by the diversification and the ΔBreakpad tools with the two Δ-minimization techniques from Section 3.2.4 enabled. Table 3.1 contains data for benchmarks compiled with `-O2 -fomit-frame-pointer`: The maximum and average sizes of the Δdata for our three techniques in isolation (A–C) and for all three combined (D). The listed Δdata sizes are the sizes of the bzipped data, or simply the size of the random seeds if there was no other Δdata to be compressed. As the Δdata sizes vary from one diversified version to another, we list their average size as well as the maximal sizes we observed during our experiments. These sizes are indicated with "(avg)" and "(max)" , respectively. The numbers (E) given for the opportunity logs for three techniques combined are also compressed using bzip2, as these files are quite large but very compressible. Also given are the sizes of the default (F) as well as the diversified symbol files (G), and the sizes of the corresponding stripped binaries (H and J). For the default binaries, we also report the average stack depth (I) observed over their execution on SPEC training inputs. This size corresponds to the amount of stack data that needs to be sent to a crash server in a minidump. As for the execution times, the table lists the time needed to compile and link the default binary (K); to generate the Δdata (L); to create a stack trace for a crash in the main function of the default binary, which requires no stack unwinding (M); and to produce the diversified symbol file on the crash server once Δdata is delivered with a minidump (N). The timing data was gathered using the Python *timeit* module on a machine with 16 GB of main memory and an Intel i7-4790 CPU. To put the absolute numbers in the table in perspective, four columns contain relative numbers on the right and aggregated numbers at the bottom of the table. The formulas to compute the relative numbers are detailed in the header rows.

We did not include execution times for generating the actual diversification, because the extra computation time needed to perform the diversification is negligible compared to the default compilation and linking times.

From the results in Table 3.1, we can draw several conclusions. First, the size of the Δdata is small. Even for the three techniques combined the extra Δdata to be stored in the binaries is roughly three orders of magnitude smaller than the binary size for each benchmark. Compared

[2] Between the publication of the paper and the publication of this dissertation we discovered a bug and subsequently resolved it. Because of this, the results in this section differ slightly from those in the published paper.

benchmark	file size (bytes)																execution time (seconds)						
	stack padding (A) data		function shuffling (B) data		NOP insertion (C) data		three diversifications combined										build system			crash server			
							(D) data				(E) opportunity log		(F) default symbol file	(G) diversified symbol file (avg)	(H) stripped default binary	(I) average stack depth default binary	(J) stripped diversified binary (avg)	(K) default compiling & linking	(L) generating data		(M) default stack trace creation	(N) replicating symbol file	
	(avg)	(max)	(avg)	(max)	(avg)	(max)	(avg)	(D/J)	(D/I)	(max)		(E/F)							(avg)	(L/K)		(avg)	(N/M)
perlbench (C)	67	118	4	4	2511	2917	2561	*0,2%*	*207%*	2960	106727	*5%*	2034359	2059642	1126652	1240	1321912	18,19	1,64	*9%*	0,058	0,940	*16,2*
bzip2 (C)	41	120	4	4	401	513	420	*0,2%*	*4%*	556	26126	*9%*	277774	280905	152336	9600	177382	1,93	0,28	*15%*	0,010	0,160	*16,0*
gcc (C)	268	547	4	4	5796	6161	6080	*0,2%*	*178%*	6581	248273	*4%*	6045927	6104283	3246200	3423	3779759	50,84	4,76	*9%*	0,163	2,640	*16,2*
mcf (C)	35	88	4	4	303	415	319	*0,3%*	*48%*	408	23675	*13%*	184516	186548	87816	659	103886	0,58	0,17	*29%*	0,008	0,110	*13,8*
milc (C)	35	77	4	4	495	582	513	*0,2%*	*54%*	617	38490	*10%*	404208	406969	190760	950	219997	2,89	0,30	*10%*	0,013	0,200	*15,4*
namd (C++)	105	161	4	4	709	793	786	*0,2%*	*23%*	897	30179	*5%*	567181	571399	304944	3402	363124	6,08	0,51	*8%*	0,020	0,290	*14,5*
gobmk (C)	391	483	4	4	1829	1998	2221	*0,1%*	*7%*	2489	98728	*5%*	1879631	1889407	3299016	32764	3444784	12,67	1,24	*10%*	0,048	0,810	*16,9*
soplex (C++)	78	125	4	4	1140	1322	1173	*0,2%*	*32%*	1350	78160	*7%*	1073919	1078691	399260	3668	470115	15,78	0,69	*4%*	0,031	0,450	*14,5*
povray (C++)	235	311	4	4	2890	3212	3080	*0,3%*	*66%*	3381	121151	*5%*	2343137	2352454	989480	4698	1157407	19,02	1,94	*10%*	0,066	0,980	*14,8*
hmmer (C)	71	99	4	4	906	1083	937	*0,2%*	*66%*	1101	48604	*7%*	744166	747934	336688	1418	395020	5,76	0,54	*9%*	0,023	0,340	*14,8*
sjeng (C)	74	186	4	4	569	722	613	*0,2%*	*0%*	799	33137	*9%*	383228	386575	215952	272457	249383	2,08	0,31	*15%*	0,013	0,200	*15,4*
libquantum (C)	48	79	4	4	352	454	369	*0,3%*	*48%*	478	26887	*12%*	232188	234676	112432	763	129644	0,84	0,19	*23%*	0,009	0,130	*14,4*
h264ref (C)	88	138	4	4	1230	1433	1315	*0,2%*	*53%*	1506	60244	*4%*	1345524	1351861	698916	2493	818383	10,48	1,41	*13%*	0,043	0,620	*14,4*
lbm (C)	35	75	4	4	287	377	302	*0,3%*	*59%*	391	22231	*12%*	183861	185738	87824	508	101284	0,29	0,17	*59%*	0,008	0,110	*13,8*
omnetpp (C++)	45	96	4	4	989	1121	1019	*0,1%*	*117%*	1149	122225	*9%*	1299185	1307457	685604	872	784841	17,31	0,94	*5%*	0,032	0,560	*17,5*
astar (C++)	41	87	4	4	380	471	397	*0,3%*	*8%*	493	26471	*10%*	255932	257895	112468	5183	130665	1,01	0,20	*20%*	0,009	0,140	*15,6*
sphinx3 (C)	86	117	41	59	598	722	638	*0,2%*	*1%*	804	38832	*8%*	476037	479281	236656	122182	274209	3,45	0,34	*10%*	0,015	0,230	*15,3*
xalan (C++)	136	186	4	4	6510	6857	6720	*0,2%*	*65%*	7069	866369	*8%*	10647877	10708617	3898260	10283	4396551	122,16	5,32	*4%*	0,204	3,520	*17,3*
							AVG:	*0,2%*	*58%*		AVG:	*8%*							AVG:	*15%*		AVG:	*15,4*
							MIN:	*0,1%*	*0%*		MIN:	*4%*							MIN:	*4%*		MIN:	*13,8*
							MAX:	*0,3%*	*207%*		MAX:	*13%*							MAX:	*59%*		MAX:	*17,5*
							client size & communication overhead				crash server space overhead								compile time overhead			crash server time overhead	

Table 3.1: Data sizes and execution times for ∆Breakpad use for benchmarks compiled at -O2 without FP

to the average stack size, which is a good indication of the average size of minidumps to be sent to a server, the Δdata can range from negligible for the sjeng benchmark to relatively large, such as for perlbench benchmark. Thus, the need to send Δdata can significantly increase the amount of data to be sent to the crash server, up to a factor 3 for perlbench. However, the increase is relatively high only for programs with shallow stacks. The absolute increase is, in each case, still limited to less than seven kilobytes.

Secondly, the symbol files barely increase as a result of diversification, and the opportunity logs are about an order of magnitude smaller than the symbol files. We can thus conclude that on the client as well as on the server, only a relatively small price is paid in terms of storage for allowing diversified symbol files to be recreated.

Thirdly, the computation times required to produce the Δdata on the build system and to produce the diversified symbol files on the crash collector server are significant. An important remark needs to be made, however. Both the generation of the Δdata on the build system and the reconstruction of the diversified symbol file on the crash collector are currently implemented in Python. Most of the execution time is spent in reading and parsing the default symbol file, and in allocating the internal data structures that represent it. These steps can be optimized significantly, by preprocessing the default symbol file such that it can be mapped into memory with one file open operation, by re-implementing the scripts in a performance-oriented programming language, and by redesigning the internal data structures for performance instead of research flexibility. The reported processing times are therefore only a large over-approximation of what more fine-tuned implementations will be able to achieve. We are hence confident that the computational overhead on both the build system and the crash collector server can be reduced to acceptable levels. With a reduction of one order of magnitude, which certainly seems within reach, the overhead on the crash server could be reduced to approximately a doubling of the computation time needed to produce a crash report.

Fourthly, the observations for C++ programs are in line with those for C programs.

Fifthly, from the individual results in columns A–C, we can make several interesting observations. Stack padding requires significant but relatively little Δdata. This results from the fact that with the default stack padding discussed in Section 3.2.4, relatively few changes to additional code regions result from stack offset changes. For almost all benchmarks, function shuffling only requires 4 bytes of Δdata, which are

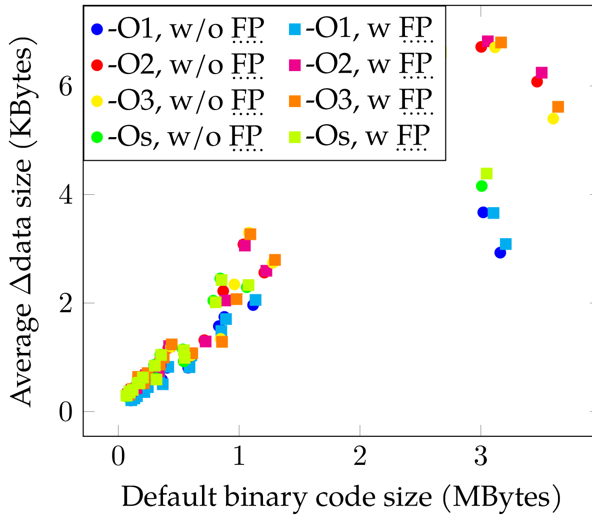


Figure 3.5: Correlation between binary code size and Δdata size

needed to store the key used for the diversification. For one benchmark, sphinx3, more Δdata is needed. This results from a small number of system functions being linked in from precompiled crt*.o files, that do not feature separate sections for each function. As a result, the alignment requirements of the functions are not replayed correctly, and patching is needed instead. Finally, the NOP insertion is responsible for the vast bulk of the Δdata. This is the case because NOP insertion affected the location of literal address pools in ways that the simple server-side replay cannot predict accurately.

Figure 3.5 charts the main result, i.e., the Δdata size, in function of the default binary code size for different compiler optimization levels (always with the Δ-minimization techniques enabled). The correlation between the two attributes of code size and Δdata sizes is clear, and it is also clear that the results are quite similar for the different optimization levels, with or without FP.

Finally, Figure 3.6 visualizes the effect on average Δdata sizes for each benchmark compiled with -O2—similar results are obtained at other optimization levels—of omitting FPs where possible, and of deploying the Δ-minimization technique discussed in Section 3.2.4. We did not include the effect of default padding (Section 3.2.4) because that does not involve any trade-off, as it does not affect the diversified binaries themselves. The blue, left bars indicate the effect on Δdata size of omitting the FP in functions where that is possible. On some benchmarks, this

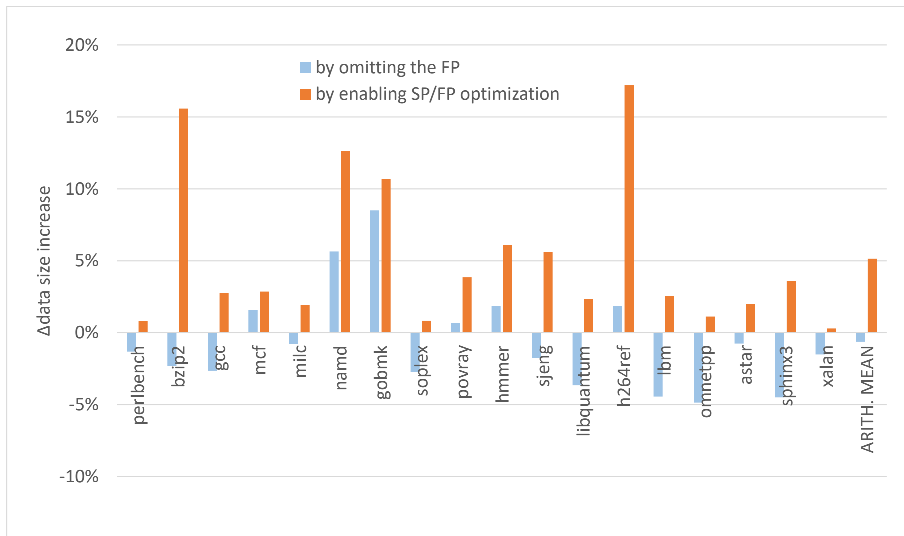


Figure 3.6: Impact on Δdata sizes from omitting the FP and from disabling the SP/FP optimization (on benchmarks compiled with FP) in LLVM (Section 3.2.4) for benchmarks compiled at -O2.

reduced the size; on others it increases the size. On average, the effect is negligible. The right, orange bars indicate the effect on Δdata size of enabling LLVM's SP/FP optimization when code with FP is generated for all functions. On average, enabling that optimization leads to 5% larger Δdata, without outliers up to 18%. We conclude that disabling the SP/FP optimization is a useful form of Δ-minimization for scenarios in which, for whatever reason, developers insist on letting their compilers generate code with FPs

Because the whole Δdata of a diversified benchmark version is more or less equal to a concatenation of Δdata chunks of the benchmark's functions, and because the effects of omitting the FP and of disabling the SP/FP optimization are also local to functions, the absolute effect of those compilation options on a benchmark's total Δdata size is also mostly a sum of their effects on a large amount of individual functions. If we assume that the large set of functions in our benchmark suite is partitioned randomly into the sets of functions of the individual benchmarks, we expect the results shown in Figure 3.6 to look more like Gaussian distributions than like uniform ones. And that is what we see. We conclude that if one's goal is to minimize the Δdata size even further than what we did, the compiler options should not be enabled or disabled per benchmark. Instead a choice should be made for each individual

function. With machine learning, or maybe even simple human analysis and engineering, we conjecture that it will be relatively straightforward to adapt a compiler for this goal. Still, it would be much more intrusive than the small patch we now deployed to let LLVM inject the randomized stack padding, the NOP insertion, and the Δ-minimization. So a trade-off needs to be made. Given the already small sizes of the Δdata achieved with our implementation, we considered it not interesting to investigate this any further as of yet.

3.5 Discussion

3.5.1 Alternative Designs

In an alternative design option of our approach, one could embed a unique ID in each diversified binary version, store all Δdata of all program versions persistently on the crash server instead of in the diversified binaries on the user systems, and include IDs in delivered crash reports to let the crash server look-up the corresponding Δdata. The IDs could then also serve as decryption and signature keys, such that the data on the crash server remains confidential until it is truly needed to build a crash report.

Despite the small sizes of the required Δdata, one problem of such a design might be the required storage for all that Δdata. In our design with the Δdata stored in the binary on the user system, the storage space occupied by old Δdata is automatically freed as soon as an old binary is discarded by the user, such as when an application is uninstalled or replaced by an updated version. No third party needs to be informed when such actions take place.

If the Δdata is stored on a server instead, the server either needs to hold on to multiple past and present versions of all Δdata, or it needs to be informed about the discarding of old binaries by users. In the former case, more storage space is needed. The latter case, depending on the application and usage context, involves the collection and communication of privacy-sensitive and security-sensitive information. Whether either of those options is feasible, is an open question.

In any case, a substantial amount of additional storage would be needed on the crash server. If a crash report service runs on a (small) farm of servers or in the cloud, it is also an open question as to what the cost might be of coupling all servers in the service to the necessary storage at sufficient throughputs and latencies. Whether or not existing

storage-computation solutions might still suffice is unclear; answering this question is considered out of scope.

In our design, where each contacted crash server receives the minidump and the Δdata over the Internet, only "centralized" access to the default symbol files and opportunity logs is needed. Our experiments indicated that accessing the opportunity logs on top of the symbol files (that a standard Breakpad setup needs to access anyway) on average requires only 8% more data to be accessed from the "centralized" storage. A 8% increase definitely is an extra cost, but it is not likely to void the feasibility of existing storage-computation solutions.

3.5.2 General Applicability

The top level of our ΔBreakpad implementation is both architecture- and compiler-independent. Lower-level components are designed to cooperate with standard Linux binutils tools such as objdump. On top of that, the design of the ΔBreakpad symbol differ, the ΔBreakpad replicator, and the Δdata format are architecture-independent and compiler-independent.

The implementation of the replicator and the opportunity log format are clearly architecture-dependent, however, and have been tuned specifically for the diversification schemes we deploy. Those schemes were also specifically chosen for the ease with which their effects could be replicated, resulting in small patches. In principle we can create patches for any diversification scheme, but there are some trade-offs. Unless replication is at least somewhat correct, patches will grow to a size where it would be preferable to simply replace them by the entire diversified symbol file. In other words, our approach then offers no benefit. Likewise, if the replication becomes too complex or time-consuming for a certain diversification scheme, the ΔBreakpad approach loses its appeal.

Consider, for example, the many diversification schemes discussed by Larsen et al. [74], which we mark in italics below. We implemented forms of three of these schemes: stack padding, which is a form of *Stack Layout Randomization*; function shuffling, which is referred to as *Function Reordering*; and NOP insertion, which is a form of *Garbage Code Insertion*. We conjecture that inserting other forms of garbage code will not result in larger Δdata as long as a similar amount of code is inserted. We furthermore conjecture that other forms can be supported with smaller Δdata, because only NOPs (having no side effects) can be inserted any-

where in code. Other instructions do have side effects when executed, though. These can only be inserted where they cannot be reached, which is definitely in fewer places. As for stack layout randomization, more heavy-weight schemes (such as those in which the locations of local variables and spilled data in a stack frame are permuted) will likely require larger Δdata, because in such schemes the offsets to the SP change. This is not, or at least rarely, the case in our scheme.

Other existing schemes would result in no changes to the debug information at all, and thus do not require any replication or patching. This will, e.g., be the case for some forms of *Register Allocation Randomization* if the randomization is limited to code-quality-maintaining randomization, i.e., if no allocations are chosen that lead to longer code schedules. *Instruction Reordering* and *Basic Block Reordering* mostly have local effects and we conjecture that with enough detail in the opportunity logs—which would thus become longer—these can be replicated sufficiently well. Schemes that have a larger impact on the CFG—such as *Inlining* and *Control Flow Flattening*—would require significantly more detailed opportunity logs and replication of compiler internals, and therefore most likely do not fit our approach.

Our current ΔBreakpad diversification schemes are applied at the compilation and linking stages of the SDLC. Schemes applied during later stages form no conceptual problem. When the diversification happens after the binary has been delivered to the user—as happens with diversification at installation time, load time, or even at run time—by nature the diversification can be performed without requiring the complexity of a full build system. In practice this typically requires that a form of opportunity log is included in the distributed binary to steer the diversification, and to allow it to be done both fast and conservatively, i.e., without altering the semantics of the diversified software. So by nature the diversification effect on the symbol file can then also be replayed on the crash server, assuming that it has access to the same opportunity log and all sources of randomness that were used during the diversification on the user system. Few such sources are needed, and as in the current implementation, they can be included in the Δdata.

We conjecture that in such cases small opportunity logs and Δdata will suffice. This conjecture is supported by the fact that currently proposed forms of diversification applied late in the SDLC are relatively simple and free of (more global) side effects as the ones we observed in, e.g., LLVM. The reason is of course that they need to be deployed very quickly to avoid downgrading the user experience, and hence without

heavy-weight compiler technology that can rewrite code to compensate for side effects.

Finally, we see no reason why our approach would be limited to specific compilation tool flows. In fact, before we implemented NOP insertion in LLVM, we already had an implementation in the post-link-time binary rewriter Diablo [104]. So the three schemes were implemented in three separate tools: the compiler, the linker, and a binary rewriter. While constructing its intermediate representation of the binary code, Diablo converts literal address pool entries into instructions. After implementing the NOP insertion, Diablo then recreates literal address pools. Whereas LLVM creates the pools per function, Diablo recreates them more globally, in effect combining pools from multiple functions into single pools. As a result, much fewer such pools end up in binaries rewritten (and diversified) by Diablo. The number of replay desynchronizations therefore was also much smaller in those Diablo-diversified binaries. As a result, the required Δdata for NOP insertion was on average 2/3 smaller. For some benchmarks, it was even 90% smaller. As Diablo does not output correct debug information, however, symbol files could not easily be generated for the binaries it created. We had to create these symbol files through an address translation, and verified their correctness by inducing crashes at randomly selected program locations in the diversified binaries. The crash reports thus generated were verified to be the same as generated for the undiversified binaries crashing at the equivalent program location.

Because of this added complexity, we eventually decided to switch to LLVM, however. LLVM is a mature, widely used tool, which makes our contributions readily available to everyone. This switch required us to adapt the generation of the opportunity log generation and the replication only slightly.

3.6 Related Work

In the past, both spatial and temporal software diversity has been proposed as a solution to a wide range of problems: Instruction set randomization can prevent, or at least delay, reverse engineering and tampering [112]. Multi-variant execution can be used to detect malware intrusions [107]. Limited, rather coarse-grained forms of run-time randomization, such as ASLR, are widely used and significantly raise the bar for memory corruption attacks [94]. In the academic literature, more fine-grained forms of diversification have been proposed to raise the

bar even further [19, 51], including for code dynamically generated with JIT compilers [59]. Dynamic temporal diversity has been proposed to mitigate timing side channel attacks [47]. Advanced software fingerprinting schemes can help in identifying the source of illegitimate software copies [28]. Diversification can prevent collusion attacks to identify software vulnerabilities based on patches [29]. Some software vendors diversify the code of their applications when major new versions are released, to hide the location of the new, valuable functionality in the new versions. Obfuscation tools and other software protection tools inherently rely on diversification to minimize the learning capabilities of attackers and to achieve stealthiness [88]. Microsoft diversifies the Window's system call numbering over time to prevent (malicious and benign) software targeting APIs they do not want to keep backwards compatible [67]. With the exception of the latter form of diversification, the other forms can only provide strong protection if code is diversified, i.e., if the diversification is not limited to changes in the embedded data.

3.7 Conclusions and Future Work

In this chapter we presented the ΔBreakpad approach to enable crash reporting on diversified software. We demonstrated an open-source implementation of co-designed compile-time software diversification, and provided crash-reporting server support for the diversified binaries. The source code of ΔBreakpad, as well as all scripts to reproduce the results presented, are available at `https://github.com/csl-ugent/delta-breakpad`.

We validated the ΔBreakpad approach for applications on which multiple fine-grained layout and offset diversifications are deployed. The tool and diversification techniques require only minimal adaptations to the build tool chain, and only a small price in storage space and communication bandwidth is paid to support the approach. It is our hopeful opinion that in providing a solution to an important issue faced by diversified software, we ease its large-scale adoption.

Further improvements to our approach can be made with respect to the employed diversification schemes. Currently these are rather simple, and it is worthwhile to investigate whether more complex techniques—such as techniques that can be deployed at install time, load time, or run time, and techniques that can stop non-control data exploits—can be supported and whether that will result in a larger overhead in terms of Δdata.

Chapter 4

Tightly-Coupled Self-Debugging

As debuggers are commonly used in MATE attacks, anti-debugging techniques such as self-debugging have become a popular protection method, as discussed in Section 2.3.3. All major OSs only allow a single debugger to attach to a process, and self-debugging works by occupying that single available debugger seat with a custom self-debugger. When the protected program is started it launches this self-debugger, which then attaches to the program. With this setup in place, attackers cannot attach their debugger to trace or tamper with the execution of the program anymore. Here, we follow the ASPIRE attack model in the assumption that it will be too much effort for attackers to introduce multiple-debugger-per-process functionality [115].

A major shortcoming of existing self-debugging schemes, however, is the simplicity of the self-debuggers. To ensure they cannot simply be replaced by an attacker's own debugger, they typically implement an obfuscated, but relatively simple inter-process control flow transfer mechanism, of which the implementation (typically based on simple address translation table look-ups) is completely predetermined by the developers of the protection tools. This feature, combined with the fact that only simple binary code patching is performed to convert individual control flow transfers to breakpoints, implies that it is relatively straightforward for knowledgeable hackers to inject a debugger-in-the-middle that iteratively resets each breakpoint to its original instruction, thus iteratively reconstructing and deobfuscating the unprotected program to re-enable standard tracing and live debugging techniques. Because the injection of breakpoints has left all other instructions in the program in their exact original location, the reconstruction only requires the attacker to flip some code bytes back to their original values, which can be

determined from the simple I/O relation of the self-debugger, or even be obtained statically from its address translation tables.

In this chapter, we present our improvements to the state of the art in self-debugging. Relying on advanced binary rewriting techniques, we propose to migrate whole chunks of functionality from the original software to the self-debugger. This offers several advantages. First, the I/O behavior of the self-debugger is no longer predetermined: Every time the self-debugger intervenes, it executes different functionality that is not predetermined, but that can instead vary as much as functionality in protected programs can vary. This makes the protection much more resilient against automated analysis, deobfuscation, and decompilation. Secondly, even if the attacker can figure out the control flow and the data flow equivalent of the original program, it becomes much harder for an attacker to undo the protection and to reconstruct that original program. Combined, we believe these two strengths make it much harder for an attacker to detach the self-debugger while maintaining a functioning program to be traced or live-debugged.

Our contributions are the following:

- We present the design of a self-debugger that executes part of the original program functionality to make it harder for an attacker to detach the self-debugger and to deobfuscate the overall control and data flow.
- We present an open-source prototype implementation and tool support for protecting stand-alone program executables as well as shared libraries.
- We discuss how to engineer the tool support to make it compatible with other software protections.
- We evaluate the tools and prototype on complex, real-life security-sensitive use cases, ranging from native libraries embedded in Android APKs and invoked with the JNI interface, to native plug-ins of the Android DRM server and the Android media server.
- We discuss the impact on attacker capabilities based on observations we made when professional penetration testers were hired to attack the protected use cases.

The remainder of this chapter is structured as follows: in Section 4.1, we discuss the overall design and applicability of our self-debugger

approach. Section 4.2 then discusses the necessary tool support. In Section 4.3, a number of implementation aspects are discussed in more detail, after which a prototype implementation is evaluated in Section 4.4. Additional practical considerations are discussed in Section 4.5, after which Section 4.6 draws conclusions and briefly discusses future work.

4.1 Design

Figure 4.1 illustrates the basic concepts of our self-debugging scheme. Our design, prototype implementation, and presentation target Linux (and derivatives such as Android). However, to the best of our knowledge, all aspects of the design are relevant and have direct counterparts on Windows, BSD variants, and OS X.

On the left of Figure 4.1, an original, unprotected application is depicted, including a small CFG fragment. The shown assembly code is (pseudo) ARMv7 code [1]. This unprotected application is converted into a protected application consisting of two parts: a debuggee that corresponds mostly to the original application as shown in the middle of the figure, and a debugger as shown on the right. Apart from some new components injected into the debuggee and the debugger, the main difference with the original application is that the CFG fragment has been migrated from the application to the debugger. Our design and current implementation support all single-entry, multiple-exit code fragments that contain no inter-procedural control flow such as function calls. The migration of such fragments is more than simple copying: Memory references such as the `LDR` instruction need to be transformed because the migrated code executing in the debugger's address space needs to access data that still resides in the debuggee's address space. All relevant components and transformations will be discussed in more detail in later sections.

At run time, the operation of this protected application is as follows. First, the debuggee is launched, as if it was the original application. A newly injected initialization routine then forks off a new process for the debugger, in which the debugger immediately attaches to the debuggee process. When later during the program's execution the entry point of the migrated code fragment is reached, *one possible flow of control* in the application follows the red arrows in Figure 4.1. In the application/debuggee, the exception-inducing instruction is executed and causes an exception (①). The debugger is notified of this exception and handles it in its debugger loop (②). Amongst others, the code in this loop is

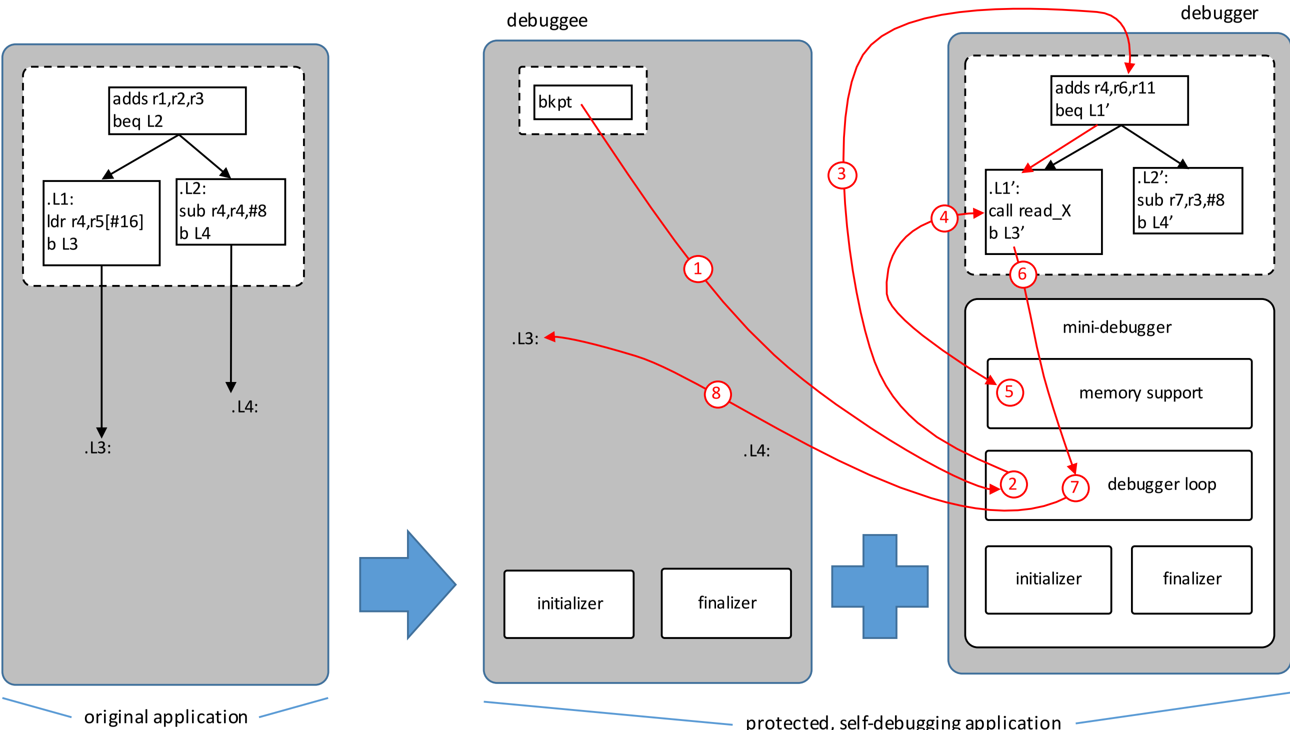


Figure 4.1: On the left, the original, unprotected application with a CFG fragment. In the middle, the protected application from which the fragment has been omitted, but with minimal functionality to launch and kill the debugger component. On the right, the debugger with the migrated fragment and mini-debugger functionality. Red, numbered edges and points show the control flow in the running, self-debugging application.

responsible for fetching the process state from the debuggee, looking up the corresponding, migrated code fragment, and transferring control (③) to the entry point of that fragment. As stated, in that fragment memory accesses cannot be performed as is. Therefore, they are replaced by invocations (④) of memory support functions (⑤) that access memory in the debuggee's address space. When an exit point (⑥) is eventually reached in the migrated code fragment, control is transferred to the corresponding point in the debugger loop (⑦), which updates the state of the debuggee with the data computed in the debugger, and (⑧) control is transferred back to the debuggee. For code fragments with multiple exits, such as the example in the figure, the control can be transferred back to multiple continuation points in the debuggee. In this regard, our debugger behaves in a more complex manner than existing self-debuggers, that implement a one-to-one mapping between forward and backward control flow transfers between debuggee and debugger. Eventually, when the application exits, the embedded finalization routine will perform the necessary detaching operations.

It is important to note that this scheme cannot only be deployed to protect executables (i.e., binaries with a `main` function and entry point), but also to protect shared libraries. Just like executables, libraries can contain initialization and finalization routines that are executed when they are loaded or unloaded by the OS loader. At that time, all of the necessary forking, attaching and detaching can be performed as well.

In the remainder of this chapter, we will write mostly about protecting applications, but implicitly, we denote applications and libraries. The only aspect specific to libraries is the need for proper initialization and finalization of the debugger. This is necessary because it is not uncommon for libraries to be loaded and unloaded multiple times within a single execution of a program. For example, repetitive loading and unloading happens frequently for plug-ins of media players and browsers. Furthermore, whereas main programs consist of only a single thread when they are launched, at the point in time where they load and unload libraries, they can easily consist of multiple threads. This complicates the attaching/detaching of debuggers to libraries.

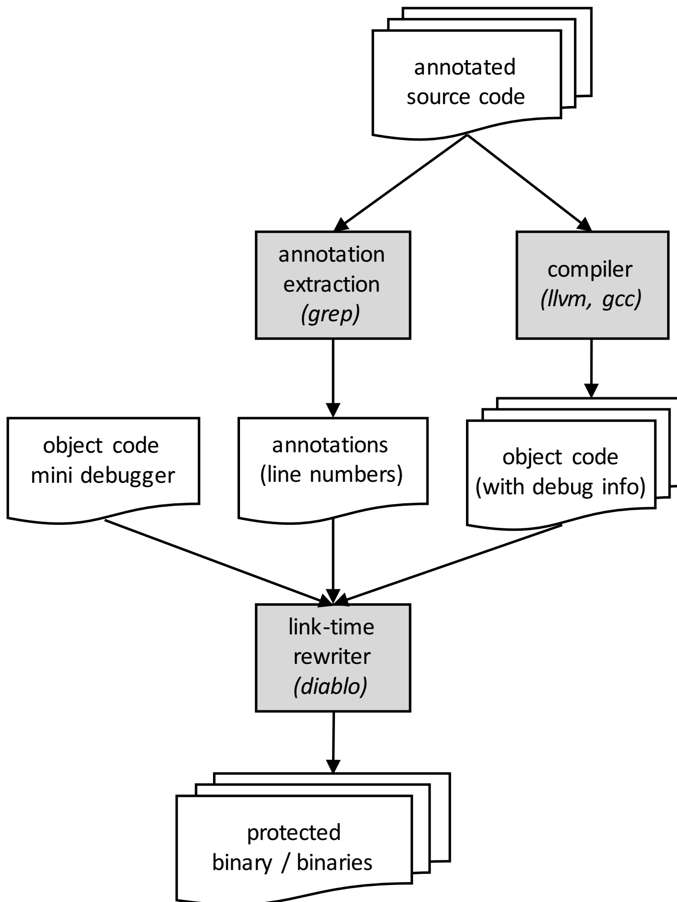


Figure 4.2: Tool flow of self-debugging support

4.2 Tool Support

Figure 4.2 depicts one possible conceptual tool flow. The main components are discussed in the following sections.

4.2.1 Source Code Annotations

A number of options exist for determining the code fragments to be migrated to the debugger. One, depicted in the figure—and also used in our implementation—is to annotate source code with pragmas, comments, or any other form of annotations that mark the beginning and end of the code regions to be migrated to the debugger process. A simple

grep suffices to extract annotations and their line numbers, and to store that information in an annotations file. Alternative options would be to list the procedures or source code files to be protected, or to collect traces or profiles to select interesting fragments semi-automatically.

In that regard, it is important to note that the fragments to be migrated to the debugger should not necessarily be very hot fragments. To achieve a strong attachment between the debuggee and the debugger, it suffices to raise exceptions relatively frequently, but this does not need to be on the hottest code paths. We will discuss good strategies to select fragments in more detail later. Obviously, every raised exception will introduce a significant amount of overhead (context switch, ptrace calls, ...). Consequently, it is important to minimize their number without compromising the level of protection.

4.2.2 Standard Compilers and Tools

To deploy our self-debugging approach, any “standard” compiler can be used: Our technique does not impose any restrictions on the code generated by the compiler. In our experimental evaluation, we used both GCC and LLVM, and we did not need to adapt or tune the code generation.

One requirement, however, is that the compiler and the binary utilities (the assembler and linker) provide the link-time rewriter with sufficiently accurate symbol and relocation information. This is required to enable reliable, conservative link-time code analyses and transformations to implement the whole self-debugging scheme, including the migration and transformation of the selected code fragments. Providing sufficiently accurate information is certainly within reach for commonly used tools. ARM’s proprietary compilers, e.g., have done so for a long time by default, and for the GNU binutils, GCC, and LLVM, very simple patches[1] suffice to prevent those tools from performing overly aggressive symbol relaxation and relocation simplification, and to force them to insert mapping symbols to mark data in code. These requirements have been documented before, and it has been shown that they suffice to perform reliable, conservative link-time rewriting of code as complex and unconventional as both CISC (x86) and RISC (ARMv7) versions of the Linux kernel and C libraries, which are full of manually written assembly code [104].

[1]Our patches are available at https://github.com/csl-ugent/toolchains.

A large, generic part of the debugger—the *mini-debugger*—can be precompiled with the standard compiler and then simply linked into the application to be protected. Other parts, such as the debug loop's *prologues* and *epilogues* for each of the migrated fragments, are generated by the link-time rewriter, as they are customized for each fragment.

To allow the link-time rewriter to identify the fragments that were annotated in the source code, it suffices to pass it the line number information extracted from the source code files, and to let the compilers generate object files with debug information. That debug information then maps all addresses in the binary code to source line numbers, which the rewriter can link to the line numbers from the annotations. To the best of our knowledge, all compilers and binary utilities support the generation of debug information.

4.2.3 Binaries, Libraries, and Processes

The link-time rewriter has two options to generate a protected application. A first option is to generate two binaries, one for the application/debuggee, and one for the debugger. From a security perspective, this might be preferable, because the application semantics and its implementation are then distributed over multiple binaries, which likely makes it even harder for an attacker to undo the protection, i.e., to patch the debuggee into the original application. This option does introduce additional run-time overhead, however, as the launching of the debugger then also requires loading the second binary.

The alternative option—that we use in our implementation—is to embed all debuggee code and all debugger code into one binary. In that case, simple forking will suffice to launch the debugger. Whether or not, and to what extent, this eases attacks on the protection provided by self-debugging is an open research question.

4.3 Implementation

4.3.1 Initialization & Finalization

We add an extra initialization routine to protected binaries. This routine is invoked as soon as the binary has been loaded (because it is assigned a high priority), after which all the other routines listed in the `.init` section of the binary are executed. This initialization routine invokes `fork()`, thus creating two processes called the parent and child [80].

Once this routine is finished, the parent process will continue execution, typically by invoking the next initialization routine.

Two options exist for assigning the debugger and debuggee roles: After the fork, either the child process attaches to the parent process, or vice versa. In the former case, the child becomes the debugger and the parent becomes the debuggee, in the latter case the roles are obviously reversed. The former option is highly preferred. The parent process remains the main application process, and it keeps the same PID (Process ID). This facilitates the continuing execution or use of all external applications and inter-process communication channels that rely on the original PID, e.g., because they were set up before loading a protected library.

This scheme does come with its own problems, however. With `dlopen()` and `dlclose()` [79], shared libraries can be loaded and unloaded at any moment during program execution. A potential problem, thus, is that a protected shared library can be unloaded and *loaded again* while the originally loaded and forked off debugger has not yet finished its initialization. This can result in the simultaneous existence of two debugger processes, both attempting (and one failing) to attach to the debuggee. In order to avoid this situation, we block the execution of the thread that called `dlopen()`. Thus, until has been allowed to continue, that thread cannot invoke `dlclose()` using the handle it got with `dlopen()`, nor can it pass this handle to another thread. Blocking this execution is implemented through an infinite loop in the debuggee's initialization routine, preventing the loading thread from exiting the initialization routine before the debugger allows it to proceed.

The initialization routine also installs a finalization routine in the debuggee. This routine does not do much. At program exit (or when the shared library is unloaded) it simply informs the mini-debugger of this fact by raising a `SIGUSR1` signal, causing the mini-debugger to detach from all the debuggee's threads and to shut down the debugger process.

4.3.2 Multithreading Support

Attaching the debugger is not trivial, in particular in the case of protected shared libraries. When a library is loaded, the application might consist of several threads. Only one of them will execute the debuggee's initialization routine during its call to `dlopen()`. This is good, as only one fork will be executed, but it also means that only one thread will enter the routine's infinite loop. The other threads in the debuggee process will

continue running, and might create new threads at any point during the execution of the debuggee's initialization routine or of the debugger's initialization routine.

To ensure proper protection, the debugger should attach to every thread in the debuggee process as part of its initialization. To ensure that the debugger does not miss any threads created in the debuggee in the meantime, we use the `/proc/[pid]/task` directory, which contains an entry for every thread in a process [81]. The debugger process attaches to all the threads by iterating over the entries in this directory, and by continuing to iterate and attach until no new entries are found. Upon attachment to a thread, which happens by means of a `PTRACE_ATTACH` request, the thread is also stopped (and the debugger is notified of this event by the OS), meaning that from then on it can no longer spawn new threads. For any program that spawns a finite number of threads, the iterative procedure to attach to all threads is thus guaranteed to terminate. Once all threads have been attached to, the infinite loop in the debuggee is ended and its stopped threads are allowed to continue.

When additional threads are created later during program execution, the debugger is automatically attached to them by the OS, and it gets a signal such that all the necessary bookkeeping can be performed.

4.3.3 Control Flow

Transforming the control flow in and out of the migrated code fragments consists of several parts. We discuss the raising of exceptions to notify the debugger, the transferring of the ID that informs the debugger about the fragment it needs to execute, and the customized prologues and epilogues that are added to every migrated code fragment.

Raising Exceptions

The notification of the debugger can happen through any instruction that causes an exception to be raised. In our implementation, we use a software breakpoint (i.e., a `BKPT` instruction on ARMv7) for simplicity. Other, less conspicuous exceptions can also be used, such as those caused by illegal or undefined instructions. When such instructions are reachable via direct control flow (direct branch or fall-through path), they can of course be easily detected statically. However, when indirect control flow transfers are used to jump to data in the code sections, and the data bits correspond to an illegal or undefined instruction, static detection can

be made much harder. Likewise, legal instructions that throw exceptions only when their operands are "invalid" can be used to conceal the goal of these instructions. Such instructions include division by zero, invalid memory accesses (i.e., a segmentation fault), or the dereferencing of an invalid pointer (resulting in a bus error).

Transferring IDs

We call the thread in the debuggee that raises an exception the *requesting thread*, as it is essentially asking the debugger to execute some code fragment.

The debugger, after being notified about the request by the OS, needs to figure out which fragment to execute. To enable this, the debuggee can pass an *ID* of the fragment in a number of ways. One option is to simply use the address of the exception-inducing instruction as an ID. Another option is to pass the ID by placing it in a fixed register right before raising the exception, or in a fixed memory location. In our implementation, we used the latter option. As multiple threads in the debuggee can request a different fragment concurrently, the memory location cannot be a global location. Instead, it needs to be thread-local. As each thread has its own stack, we opted to pass the fragment's ID via the top of the stack of the requesting thread.

Depending on the type of instruction used to raise the exception, other methods can be envisioned as well. For example, the dividend operand of a division (by zero) instruction could be used to pass the ID as well.

Finally, many data obfuscation techniques [88] can be used to hide the values of the passed IDs, thus complicating the reverse engineering of the control flow in the original application.

Prologues and Epilogues

The debugger loop in the mini-debugger is responsible for fetching the program state of the debuggee before a fragment is executed, and for transferring it back after its execution. Standard `ptrace` functionality is used to do this: With one API call, the status of all registers in the debuggee can be retrieved in a struct in the debugger. Likewise, one API call suffices to update all registers in the debuggee with the values in a struct.

For every migrated code fragment, the debug loop also contains a custom prologue and epilogue to be executed before and after the code fragment, respectively. The prologue loads the necessary values from the struct into the debugger's registers, the epilogue writes the necessary values back into the struct. The prologue is customized in the sense that it only loads the registers that are actually used in the fragment (the so-called live-in registers). The epilogue only stores the values that are live-out (i.e., that will be consumed in the debuggee) and that can have been updated by the executed code fragment.

4.3.4 Memory Accesses

For every load or store operation in a migrated code fragment, an access to the debuggee's memory is needed. There exist multiple options to implement such accesses.

The first is to simply use `ptrace` functionality: The debugger can perform `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` requests to read and write in the debuggee's address space. In this case, per word[2] to be read or written, a ptrace system call is needed, which results in a significant overhead. Some recent Linux versions support wider accesses, but those are not yet available everywhere, such as on Android.

The second option is to open the `/proc/[pid]/mem` file of the debuggee in the debugger, and then simply read or write in this file. This is easier to implement, and wider data can be read or written with a single system call, so often this method is faster. Writing to another process's `/proc/[pid]/mem` is not supported on every version of the Linux/Android kernels, however, so in our prototype write requests are still implemented with the first option.

A third option builds on the second one: If the binary rewriter can determine which memory pages will be accessed in a migrated code fragment, the debug loop can actually copy those pages into the debugger address space using the second option. The fragment in the debugger then simply executes regular load and store operations to access the copied pages, and after the fragment has executed, the updated pages are copied back to the debuggee. This option can be faster if, e.g., the code fragment contains a loop to access a buffer on the stack. Experiments we conducted to compare the third option with the previous two options revealed that this technique might be worthwhile for as few as 8 memory accesses. We did not implement reliable support for it in our prototype,

[2]The ptrace word size depends on the processor architecture.

however: A conservative link-time analysis for determining which pages will be accessed by a code fragment remains future work at this point.

A fourth potential option is to adapt the debuggee, e.g., by providing a custom heap memory management library (malloc, free, ...) such that all allocated memory (or at least the heap) is allocated as shared memory between the debuggee and the debugger processes. Then the code fragments in the debugger can access the data directly. Of course, the fragments still need to be rewritten to include a translation of addresses between the two address spaces, but likely the overhead of this option can be much lower than the overhead of the other options. Implementing this option and evaluating it remains future work at this point.

Security-wise, the different options will likely also have a different impact, in the sense that they will impact the difficulty for an attacker to reverse-engineer the original semantics of the program and to deconstruct the self-debugging version into an equivalent of the original program. Without penetration tests, we are not in a position yet to make strong statements in any one direction, however.

4.3.5 Combining Self-Debugging With Other Protections

To provide strong software protection against MATE attacks, one protection technique does typically not suffice. For example, on top of self-debugging, a good protection also requires obfuscation to prevent static analysis, and anti-tampering techniques to prevent all kinds of attacks. The binary rewriter that implements our self-debugging approach also applies a number of other protections, including:

- Control flow obfuscations: the well-known obfuscations of opaque predicates, control flow flattening, and branch functions [25, 78, 109].
- Code layout randomization: code from all functions is mingled and the order and layout are randomized.
- Code mobility: a technique in which code fragments are removed from the static binary and only downloaded, as so-called mobile code, into the application at run time (see Chapter 5).
- Code guards: online and offline implementations of techniques in which hashes are computed over the code in the process address space to check that the code has not been altered.

- Control flow integrity: a lightweight technique in which return addresses are checked to prevent the invocation of internal functions from external code.
- Instruction set virtualization: a technique with which native code is translated to bytecode that is interpreted by an embedded virtual machine instead of executed natively.

Combining the self-debugging technique with all those protections poses no problem in practice. In the link-time rewriter, it is not difficult to determine a good order to perform the transformations for all of the protections, and to prevent the deployment of multiple techniques on the same code fragments when those techniques do not compose.

For example, in our prototype implementation we do not yet support mobile code in the debugger. Furthermore, the debugger needs to know the exact continuation points to transfer control to in the debuggee. However, mobile code is relocated to randomized locations. Consequently, at least for the time being, our prototype implementation of self-debugging does not compose, on the same fragment, with code mobility. Handling all protections correctly requires some bookkeeping, but nothing complex.

As for the run-time behavior, the techniques compose as well. Multiple techniques require initialization and finalization routines, but in the debugger process we do not want to execute these routines for the other protections. After all, that process should only be a debugger, and not another client for code mobility or any other technique. To prevent the other initialization routines from executing, the self-debugger routine is given the highest priority. It is executed first when a binary is loaded, and the debugger initialization routine implements in fact both the real initialization, as well as the debug loop. The routine therefore never ends (that is, as long as the finalization routine is not invoked), and hence control is never transferred to the other routines that might be present in the binary.

Finally, we should point out one limitation of our current design and tool support. As presented, it can only be deployed once in a running process. In other words, with the design and implementation details presented in the remainder of this chapter, either the main application or one of the shared libraries can be protected, but not more. This limitation stems from the fact that in order to protect multiple libraries (including possibly the main program), one debugger needs to contain, or have at least have access to, the migrated code fragments and all auxiliary

code and data of all protected libraries. The extensions required to our scheme to support this are future work at this point.

4.4 Evaluation

4.4.1 Evaluation Platform

Our prototype implementation targets ARMv7 platforms. Concretely, we targeted and extensively evaluated the implementation on Linux 3.15 and (unrooted) Android 4.3/4.4. We also checked whether the techniques still work on the latest versions of Linux (4.7) and Android (7.0), which is indeed the case.

Our testing hardware consist of several developer boards. For Linux, we used a Panda Board featuring a single-core Texas Instruments OMAP4 processor, an Arndale Board featuring a double-core Samsung Exynos processor, and a Boundary Devices Nitrogen6X/SABRE Lite Board featuring a single-core Freescale i.MX6 processor. The latter board was also used for the Android versions.

In our tool chain, we used GCC 4.8.1, LLVM 3.4, and GNU binutils 2.23. We compiled code with the following flags: `-Os -march=armv7-a -marm -mfloat-abi=softfp -mfpu=neon -msoft-float`.

4.4.2 Use Cases

To evaluate the real-world potential of our self-debugging scheme, we deployed it on three use cases that were developed independently in three market leader companies. The three use cases were hence developed using different development approaches, different software architectures, and even different build systems. Each use case consists of one or more shared libraries to provide us with software of sufficient complexity to be representative for real software products and with embedded, security-sensitive assets representative of the assets (and corresponding security requirements) in the companies' real products. We chose the fragments to migrate from the application into the debugger together with the application architects and developers, and with security architects from the companies.

Table 4.1 lists a number of features of the three use cases as an indication of their representativeness of real-world software. The number of source code lines includes all the mentioned third-party libraries that are compiled and statically linked into the shared libraries to be protected.

use case	developer	src lines	third-party libraries	assets
DRM	Nagravision	306.247	OpenSSL	keys
software license manager	SafeNet Germany	55.487	libtomcrypt, libtommath	keys
OTP generator	Gemalto	360.446	OpenSSL, libcurl	seed, counter

Table 4.1: Feature matrix of the self-debugging use cases

This static linking is a security requirement, because dynamic linking leaks too much symbolic information to attackers. Whereas the linked-in libraries do not contain any assets, they operate on assets such as keys, and the flow of control into them needs to be protected against reverse engineering as well.

Digital Rights Management

The first use case consists of two plug-ins, written in C and C++ at Nagravision S.A., for the Android media framework and the Android DRM framework. These libraries are necessary to obtain access to encrypted movies and subsequently decrypt them. A video app programmed in Java is used as a GUI to access the videos. This app communicates with the mediaserver and DRM frameworks of Android, informing the frameworks of the vendor whose plug-ins it requires. On demand, these frameworks then load the plug-ins. Concretely, these servers are the `mediaserver` and `drmserver` processes running on Android.

During our experiments and development, we observed several features that make this use case a perfect stress test for our protection. First, `mediaserver` is multi-threaded, and creates and kills new threads all the time. Secondly, the plug-in libraries are loaded and unloaded frequently. Sometimes the unloading is initiated even before the initialization of the library is finished. Thirdly, as soon as the process crashes, a new instance is launched. Sometimes this allows the Java video player to continue functioning undisrupted, sometimes it does not. This makes debugging the implementation of our technique even more complex than it already is for simple applications. Fourthly, `mediaserver` and `drmserver` are involved in frequent inter-process communications.

Software License Manager

The second use case is a software license manager that stores credentials and controls access to licensed content and functionality, e.g., through time-limited licenses, key-enabled licenses. This license manager is programmed in C at SafeNet Germany GmbH (which has since been

Transformation	Overhead
Control Flow Switch	1.7 ms
Memory Read	3.4 μs
Memory Write	2.3 μs

Table 4.2: Overhead of self-debugging transformations

acquired by Gemalto S.A.). The library includes the JNI interface, and is embedded in an Android app. This native library thus functions as a license manager for a Java application. In this case, the Java application is relatively simple: It is a riddle game of which the solutions are protected by the license manager.

To test our self-debugger technique, this use case is also interesting. In particular, the library is loaded into Android's Dalvik execution environment, which features multiple threads (such as for the JIT compiler, garbage collector, ...), and over which we have absolutely no control [9].

Fortunately, a command-line version of the riddle game is also available, programmed in C. It uses the same library (except the JNI wrapper). On top of providing an easier target to debug on our Android developer boards, this command-line version can also be compiled for Linux. This way, we could also test our implementation on Linux.

One-Time Password Generator

The third use case is a one-time password generator developed in C and C++ at Gemalto S.A. In this case, the native library is responsible for storing and accessing counters and seed values necessary to generate one-time passwords, as well as for provisioning the initial counter and seed values. This library is again embedded in a Java Android application, which in this case simply provides GUI functionality on top of the functionality in the library.

4.4.3 Correctness

We evaluated the technique for correctness through extensive testing, first on toy examples and then on the three use cases. For the use cases, we tested self-debugging combined with many different combinations of the protections listed in Section 4.3.5. After a considerable amount of engineering effort, we reached the status of reliable correct execution, even in complex situations such as native code libraries loaded into

Google's Dalvik environment and plug-ins loaded into the Android DRM and media servers. The latter server proved to be particularly testing, as described in Section 4.4.2.

4.4.4 Execution Overhead

On the use cases, self-debugging did not introduce significant overhead. This is due to the nature of the assets and code fragments protected with the technique, which are not located on hot code paths.

To get an idea of the actual overhead, we also performed measurements on micro-benchmarks. Our aim was to measure the overhead introduced by our transformations, both on switching the control flow to the debugger and back, and memory accesses from the debugger to the debuggee. Each micro-benchmark migrated a little code fragment to the debugger. For memory accesses this was a load from memory or a store to memory that was executed in a loop (the entire loop being migrated). For control flow the migrated code fragment consisted of a single instruction (an `ADD` instruction), which was contained in a loop that was not migrated. As these transformations only replace a single instruction, the original execution time of the micro-benchmarks is simply that of the respective instructions (which is on the order of nanoseconds).

We tested these micro-benchmarks on our Linux board (see Section 4.4.1), and made sure the loops had sufficiently high trip counts to make the execution time measurable and to ensure the execution time of the micro-benchmark was completely dominated by the transformation we wanted to benchmark. Table 4.2 lists the results.

4.4.5 Security Analysis

We describe four categories of possible vectors of attack against our technique: circumventing or avoiding the migrated control flow fragments, reverting the binary transformations, attacking the mini-debugger directly, and full system emulation.

Circumvention & Avoidance

One possible method of attack is to prevent the mini-debugger from ever being invoked. This requires the attackers finding a path between an entry point of the protected binary and the area they are interested in, that does *not* contain any migrated control flow fragment. Attackers can

then disable the mini-debugger, attach their own debugger, and debug the found path without any consequences. Finding such a path will be easier in shared libraries. Whereas an executable possesses a single entry point, shared libraries usually have multiple.

Even if no unprotected path to an area of interest exists, attackers can debug this area if they manage to disable the mini-debugger at the right moment. That is, after the last migrated fragment but before the area is entered. We will not go into the question of exactly how one would deduce from the application's execution that the right moment for intervention has arrived. A plethora of side channels might be used for this.

Reverting the Transformations

If the attacker simply reverts all the transformations that were applied to migrate control flow fragments, the mini-debugger can be disabled without problem. We differentiate between memory access transformations and control flow transformations. Both classes of transformations need to be reverted for this attack to succeed, but reverting a control flow transformation requires determining the address to which control should be transferred. In the current implementation this can easily be done through static analysis. For example, when a migrated fragment is invoked one can simply find the destination address by looking up the fragment ID in a data structure (see Section 4.3.3).

Attacking the Mini-Debugger

The mini-debugger itself obviously also forms an attack surface through which the application can be compromised. As the child process containing the mini-debugger is not protected, an attacker can attach a debugger to it and attempt to observe and manipulate the application through it. Through observation of the mini-debugger the attacker can learn more about the control and data flow of migrated fragments, which can be used in the other attacks discussed previously.

Instead of attaching their own debugger to the application, attackers can also subvert the mini-debugger for their own purposes. They can attach their own debugger to the mini-debugger process and subvert its `ptrace` privileges over the target application for their own purposes. Using this indirection, `ptrace` requests can be inserted into the application to one's heart desire.

Another possibility would be for attackers to develop their own debugger that incorporates the mini-debugger's functionality and that augments it with real debugging functionality. The mini-debugger could then be safely disabled and replaced with this new debugger.

Full System Emulation

Obviously, full system emulation could also be used to trace and debug our self-debugging applications. To the best of our knowledge, however, and as confirmed to some extent by the penetration tests described below, no such emulators are available for our targeted platform. On top of that, analyzing and understanding the interaction between two processes, with all kernel interactions in between, will be harder than debugging a single program in isolation.

4.4.6 Penetration Tests

For each of the three Android use cases, professional penetration testers were hired for several weeks to attack the assets in the code protected with many techniques, as discussed in Section 4.3.5. Whereas all of them tried to use tracing and live-debugging techniques, within the time frame of the penetration tests none of them succeeded in collecting full traces or attaching debuggers for live-debugging of the most interesting code fragments being executed in situ. The latter of course followed from the manual selection of migrated code fragments by the use case developers, which ensured that migrated code fragments occur on all execution paths to the relevant code fragments.

The evaluated tools that break on self-debugging libraries include: gdb, valgrind used both as a stand-alone tracer tool and as a gdbserver for gdb and IDA Pro, and QEMU [7, 36, 50, 89]. The fact that gdb breaks is not surprising, as it depends on the ptrace interface to which access is blocked by the self-debugging technique. Valgrind currently does not support the `BKPT` instruction correctly, and it cannot emulate ptrace calls, so it cannot emulate a self-debugging program correctly. The QEMU version corresponding to our Android targets also does not support the `BKPT` instruction correctly.

Other tracing tools, such as dtrace [55] and systemtap [38] were not evaluated because they do not support our Android platform and because they do not support interactive debugging anyway. Their tracing and debug actions need to be scripted beforehand.

Some penetration testers did succeed, however, in tracing and live-debugging code out of context. They then loaded a library into a specially crafted main program that directly invokes some of the library functions in isolation. In essence, they were able to create an execution path leading to some of the interesting internal functions without first executing a migrated fragment.

4.5 Practical Considerations

4.5.1 Fragment Selection

As explained in Section 4.1 the decision of what fragments are to be migrated to the debugger rests with the programmer. This is an important decision, as selecting the wrong fragments will result in a weakened protection, as was discussed in the previous section. Therefore, the location of the selected fragments in the control flow should be right before and inside all of the code regions an attacker might be interested in.

The fact that this selection is not straightforward was clear when the experts chose the fragments to be migrated. For example, at some point they mistakenly chose to migrate variable initialization code of which the initial values later proved to be dead. While their choice still resulted in control flow being transferred to the debugger, and control flow hence being obfuscated, an attacker could relatively easily undo the protection by rewriting the exception-inducing instruction, thus circumventing many of the challenges that general code rewriting exhibits.

To hinder the transformations being reverted, a selected fragment should contain sufficient code, and code that is sufficiently complex. Some examples are control flow internal to the fragment, memory accesses, and complex computations.

4.5.2 Impact on Multithreading

While we did not observe this in our use cases and test programs, a potential issue with our technique might be that the debugger process is single-threaded, while the debuggee process is multi-threaded. Handling all requests to execute migrated fragments in the debugger might therefore become a bottleneck. Clearly, the solution to this problem is engineering a more complex, multi-threaded mini-debugger.

4.5.3 OS Limitations

In the current implementation, the debugger forks from the protected application and attaches to it using `ptrace`. However, the `ptrace` interface is quite powerful, and over the past years a number of protections placing restrictions on its use have been introduced and adopted by some Linux distributions. When enabled, these protections can hinder our technique or even make it impossible to use.

One of the protections introduced is `ptrace_scope`, which places restrictions on attaching to another process [118]. In Ubuntu, e.g., the enabled restriction level allows a process to attach only to its children [103]. In our case this can still be overcome however, as we have the ability to execute code in the protected application: During initialization the application can explicitly allow the debugger process to attach (using `prctl(PR_SET_PTRACER, debugger, ...)`).

Still, it is possible for Linux distributions to choose higher restriction levels of `ptrace_scope`. In that case, our self-debugger will not work.

4.6 Conclusions and Future Work

We proposed to migrate code fragments from an application to a debugger that serves as an anti-debugger. This way, we can make attacks on self-debuggers significantly harder: The semantics of the code in the debugger is not predetermined, and multiple control flow paths are possible per invocation of the self-debugger.

Our open-source prototype implementation works on complex, real-world use cases, as demonstrated by protecting complex shared libraries for Android. We also discussed multiple implementation issues and options. The source code can be found together with the entire ASPIRE framework at `https://github.com/aspire-fp7/framework`. Specifically, the source code of the Diablo link-time rewriter is available at `https://github.com/csl-ugent/diablo`, and the self-debugger source code and associated Diablo support code is available at `https://github.com/csl-ugent/anti-debugging`.

As for future work, a number of issues and avenues for further research remain. First, an open issue is the development of support for protecting multiple libraries that are loaded within the same application. Secondly, the idea of reciprocal debugging shows promise. Here, not only would the main application be debugged by the self-debugger, but

it would also serve as a debugger to the self-debugger. We believe that there are no technical issues that make reciprocal debugging impossible, but a significant engineering effort is still required. Reciprocal debugging would certainly make certain attack paths harder, e.g., by requiring the inclusion of a full emulation step on the attack paths to simply observe the control flow implemented by the debugger. Thirdly, in the near future we wish to investigate the impact of different implementation aspects (such as ways to transfer and obfuscate fragment IDs) on the effort required by attackers.

Chapter 5

Native Code Mobility

One way of defending against MATE attackers is through the code mobility protection technique, as discussed in Section 2.3.1. Code mobility is an online protection technique where parts of the program are downloaded at run time from a trusted server. Through code mobility, Collberg et al. [26, 27] and Falcarin et al. [40] proposed the continuous replacement of Java and binary code, respectively. Here, the trusted server periodically sends a set of new code blocks to the user's untrusted machine.

Existing implementations of code mobility either do not operate on binary code, or do not allow selectively making only certain parts of the program's functionality mobile, hindering composability with other protections at the binary level.

In this chapter, we present our code mobility framework. Through source code annotations, developers can specify which parts of the program are to be made mobile. The framework then splits off the pieces of native code associated with these annotations, and forms mobile blocks out of them. These blocks are then placed on a trusted server, in charge of providing mobile code blocks to the untrusted client. To protect against code analysis, the code mobility framework delivers native code to the client at run time; the client application modifies its own code layout to install the downloaded code blocks, in order to thwart static analysis and increase the difficulty of dynamic analysis.

The code mobility framework has been developed within the ASPIRE project [102], and it is compliant to the software protection reference architecture designed in the project and documented in deliverable D1.04 [116], which is available on the project website.

Our main novel contributions are:

- A design and open-source prototype implementation with demonstration on standard Android code.
- Integration with compilers commonly used for native code development, including in the Android NDK.
- Integration in a whole tool-flow (of the ASPIRE project) to ensure as much composability as possible with other protections.
- Very fine-grained code blocks (albeit with a performance overhead).
- A convenient way to specify and control deployment via source code annotations.
- An evaluation on real networks, ranging from local networks, to 3G mobile networks.

The remainder of this chapter is structured as follows: in Section 5.1 we introduce the code mobility architecture and all its components, then in Section 5.2 we describe how to create offset-independent mobile code. In Section 5.3 we introduce the automated tool support to instrument and split binary code in code blocks before run time; then, in Section 5.4, we describe the performance analysis of our framework on different network settings. In Section 5.5 we discuss some related work, while Section 5.6 draws the conclusions and discusses future work.

5.1 Architecture

In the code mobility architecture we designed and the prototype tool support we developed, a client application (which may also be a dynamically linked library) is stored on the user device as an incomplete binary that does not contain all the application's code. Two components, Downloader and Binder, are introduced for this technique: They are able, respectively, to fetch binary code blocks from a trusted server at run time, and to patch these into the running process' memory, in a dynamically allocated memory area. These components are not part of the original application and they have to be injected into the protected version. This approach aims to mitigate reverse engineering: Instead of preventing code analysis by making the code more complex, we make

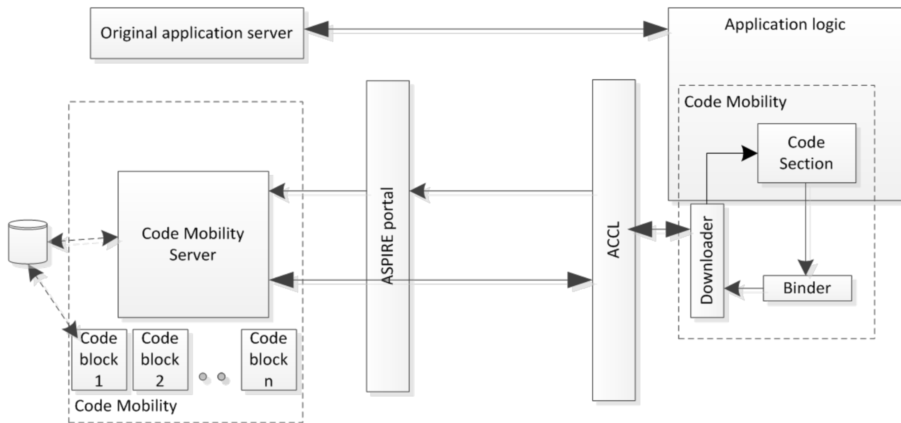


Figure 5.1: Code mobility high-level architecture

sure that the code is not available for static analysis on the client side as long as possible, and deliver the necessary code only when it is actually needed by the control flow. The code mobility framework's architecture is depicted in Figure 5.1: It can be seen as a dynamic binary obfuscation approach, based on the deployment of an incomplete application. The missing application code arrives from a trusted network entity (the code mobility server) as a flow of mobile code blocks. Such blocks are fetched by the Downloader component and arranged in memory by the Binder component at run time, with an unpredictable memory layout. The code mobility framework is compliant to the ASPIRE project reference architecture [116], defining the ASPIRE portal—which acts as a common entry point for all online protections developed in the ASPIRE project—and the ACCL (ASPIRE Communication Control Logic) library in the client host, which provides native socket support to Android apps.

Mobile code blocks coming from the code mobility server will not be placed in a statically known location in the binary code section, but will instead be placed in dynamically allocated memory. Consequently, the location of the code blocks will not be fixed. This implies that the mobile code needs to be PIC (Position-Independent Code) that can be dynamically relocated, and independently so from the non-mobile part of the client's binary code. Thus, indirections need to be inserted in the transformed code to deal with these variable code locations, both in the static, non-mobile parts of the client application and in the mobile code. Fortunately, only local code transformations are required to implement

this: Instructions will be replaced with small code snippets that can deal with the a priori unknown addresses at which the code has been loaded.

Our current design only supports mobile code blocks with a single entry point. These can be entire functions or parts of their CFGs. This significantly simplifies the implementation of the Binder and its bookkeeping data structures. We should also point out that we only make code mobile. Data allocated statically in the binary's data sections is left static, including statically allocated data that is accessed by the mobile code.

As an alternative to downloading entire mobile blocks—as done in our current design—we could also store the mobile blocks in an encrypted manner in the binary, and only store a decryption key on the server. When a mobile block is then needed, only the decryption key needs to be downloaded from the server, instead of the complete mobile block. The advantage of this approach is a smaller network overhead. Security-wise there is no real difference to this approach: The mobile block can be decrypted upon intercepting the decryption key from the server, but then again intercepting the mobile block (or dumping it from memory) is already a possibility. A disadvantage of this alternative approach is that storing the mobile blocks in the binary itself makes them somewhat fixed. They can not be "updated" on the server if they do not reside on it. The functionality to updating/overriding such fixed blocks could be added to the Downloader, however, at the cost of increased complexity and undoing the decrease in network overhead. Another hybrid strategy would be to store only some blocks locally in the binary, and for others to still require the full download from the server.

5.1.1 Binder

The client-side Binder component is in charge of invoking the Downloader when required. The Binder is invoked by the application when the control flow reaches a mobile code block. If that block has not been downloaded from the server yet, the Binder asks the Downloader to retrieve the requested missing code block. Through the ACCL communication library—implementing a socket API in native code—the Downloader queries the code mobility server to download the mobile block in question. After the block has been downloaded, the Binder places it in memory and makes sure that it will not be downloaded again, reducing the overhead effort introduced by the protection tech-

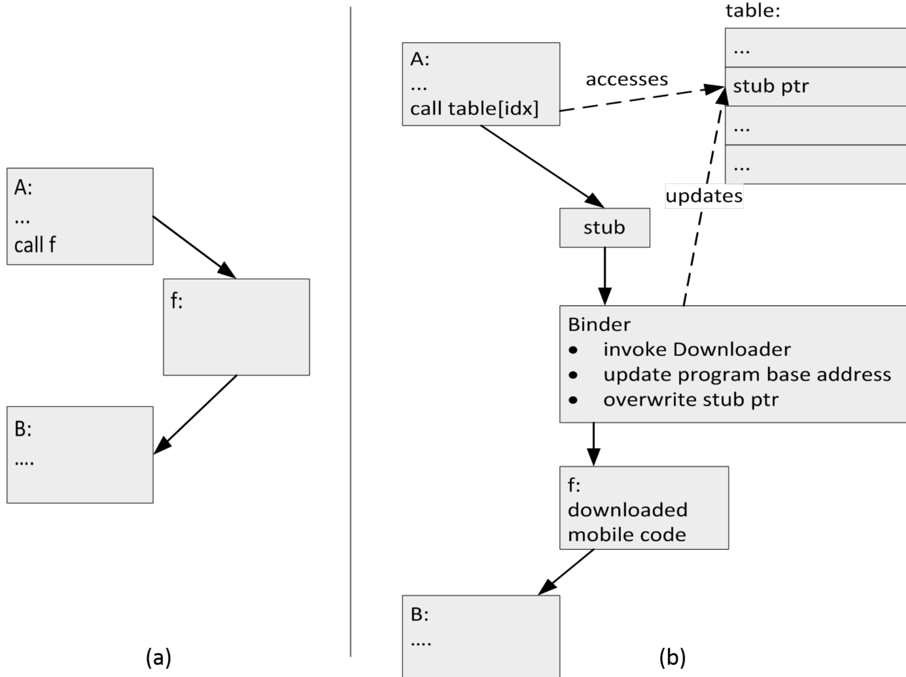


Figure 5.2: Function call: before (a) and after (b) code mobility transformations

nique. Eventually the Binder redirects the control to the entry point of the downloaded code, and program execution continues normally.

In the original client application, control flow transfers (such as function calls) to mobile regions need to be transformed such that:

1. Upon the first execution of a call to a mobile region, the Binder and Downloader components are properly invoked in order to obtain the code from the server.

2. Upon subsequent calls to the same mobile region, the control is immediately transferred to the already downloaded mobile code.

By avoiding going through the Binder again, the performance overhead of mobile code can be limited. Figure 5.2(a) shows the original control flow without mobile code: function f() is selected to become mobile. In the transformed program, shown in Figure 5.2(b), our tools inserted a look-up table with function pointers. Look-up table accesses are depicted with dashed arrows whereas control flow transfers are depicted with regular arrows. The pointers in the look-up table either point

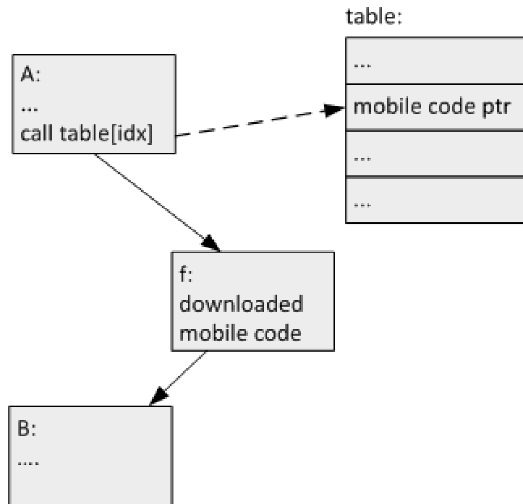


Figure 5.3: Calling already downloaded mobile function f()

to stubs that invoke the Binder to start the mobile code downloading process, or they point directly to the already downloaded code. All calls to mobile functions are transformed into a code snippet consisting of a table look-up and an execution control redirection to the address loaded from the table.

Initially, when the called mobile function f() has not yet been downloaded and bound, the address in the look-up table is that of a stub that invokes the Binder. This stub calls into the Binder, providing as argument the index at which this stub is installed in the look-up table. This index is then used as the identifier of the mobile function to be downloaded. Subsequently, the Downloader component is invoked to retrieve the mobile (PIC) version of the function—the mobile block—from the code mobility server, and stores this block in a dynamically allocated buffer. Finally, the Binder updates the entry in the pointer look-up table by overwriting the address of the stub with the address of the downloaded code, after which it redirects the control to this code, and normal code continues.

Subsequent calls to the already downloaded function f() then proceed as indicated in Figure 5.3. Since the Binder has already updated the pointer in the look-up table at the used index to let it point to the downloaded code, the inserted code snippet (in block A in Figure 5.3) now loads this function pointer and thus transfers control immediately to the previously downloaded mobile code. So for subsequent calls, the

overhead is limited to the table look-up, and the necessary spilling and restoring of registers.

The Binder contains three tables: the GMRT (Global Mobile Redirection table), a mutex table, and a table that stores whether a certain mobile block is present or not (if it is not, the entry is zero). At program startup, for a certain mobile block its GMRT entry contains the address of the associated stub, the mutex entry is initialized, and the entry in the last table is zero. When control is transferred to the stub through the GMRT, it will itself invoke the Binder with the index for the mobile block as an argument. The Binder locks the corresponding mutex and checks whether the block is present. This is very unlikely to happen, unless another thread just downloaded it.

If the block is not present, the Binder instructs the Downloader to download the block. It then writes the base address of the protected binary onto the first four bytes of the mobile block, maps all the pages the block resides on as executable, backs up the current GMRT entry (which is the address of the stub) to the last table, replaces the GMRT entry with an address in the mobile block, and unlocks the mutex. As a small aside, the locking and subsequent unlocking of a mutex is not actually done in single-threaded applications, avoiding unnecessary cost.

5.1.2 Downloader

The Downloader is invoked by the Binder to request a specific mobile code block (identified by an index) when needed by the client application. After a mobile code block is correctly received a suitable heap-allocated memory area is prepared, filled with mobile code, and passed back to the Binder. The returned memory area must be allocated with respect to a few constraints:

- It must be memory-page-aligned so that the Binder can apply the proper access rights (execution) later.
- Every mobile block must be allocated in one or more dedicated memory pages so that there are no access right conflicts: after a page is declared as execution-only it should not be accessed in write mode to avoid segmentation faults.

The first constraint is respected by using the posix_memalign system call which allocates page-aligned memory. The latter is respected by simply allocating the minimum number of memory pages able contain

to the full mobile code block. These constraints result in an additional overhead (in terms of time and memory consumption) because, after receiving the buffer containing the mobile block, the Downloader must copy it into a new memory-aligned one. This overhead could be avoided introducing a new parameterization that instructs the ACCL API to allocate page aligned buffers natively. Furthermore, reserving full memory pages for single mobile blocks leads to an additional overhead in memory allocation. This overhead can be computed as:

$$\sum_{i=1}^{N} ps - mbs_i$$

where N is the total number of mobile code blocks transferred over time, ps is the single memory size, mbs_i is the ith mobile block size. In a scenario where one hundred blocks are extracted from the original application the additional overhead is upper limited by the page size times one hundred. As an example if the page size is 4KB the "wasted memory" would be less than 400 KB. Tuning the amount of original binary code made mobile can mitigate this.

5.1.3 Server-Side Components

This component is reachable by the client via a network link and is assumed to be trustworthy. The code mobility server is the back end invoked by the Downloader component on the client side. It is in charge of delivering requested mobile code blocks by accessing a repository using a given index.

5.2 Offset-Independent Mobile Code

When a mobile code block is mapped into the address space of the binary, this is done on a randomized address on the heap because of ASLR. The statically allocated, non-mobile code and data of the binary is randomized as well. This implies that the offset between the mobile code block and the non-mobile code and data is unknown at compile time. This differs from standard position-independent code, where the offsets between elements in a statically allocated segment are still fixed. Position-dependent code or PIC in the original binary therefore needs to be rewritten into so-called offset-independent code.

```
        .text
        ...
.Lins0: or r4, r5, #1
.Lins1: ldr r1, [pc,#20]
.Lins2: ldr r1, [pc,r1]
.Lins3: add r1, r1, #3
.Lins4: mul r1, r1, r4
        ...
        .word: .Ldata-(.Lins2+8)

        .data
.Ldata: .word 0x12345678
```

(a) original static position-independent code accessing statically allocated data

```
        .text
.Ltext: ...

.Lins0: or r4, r5, #1
        ldr r6, [pc,#20]
.Ltmp:  add r6,pc,r6
        ldr r6, [r6,#16]
        bx r6
        ...
.Lins4: mul r1, r1, #r4
        ...
        .word   .Lgmrt-(.Ltmp+8)

        .data
.Ldata: .word 0xcafebabe
```

(b) remaining static position-independent code and statically allocated data

```
        .text
.Lbase: .word 0x00000000
        ... #code for restoring
        ... #registers
.Lins1: ldr r1, [pc,#-40]
.Lins2: ldr r2, [pc,#20]
        ldr r1, [r1,r2]
.Lins3: add r1, r1, #3
        ldr r2, [pc,#-32]
        ldr r3, [pc,#8]
        add r2,r3,r2
        bx r2
        .word: .Ldata-.Ltext
        .word: .Lins4-.Ltext
```

(c) offset-independent mobile code block accessing statically allocated data

Figure 5.4: Example of offset-independent code

On architectures like x86, this rewriting is straightforward, as one of the registers is used (by convention) as a so-called GP (Global Pointer) to the GOT (Global Offset Table) that contains pointers to all code and data fragments of which the absolute address might be needed at some point. On architectures like ARMv7, however, position-independent code makes heavy use of the visible PC register and of PC-relative addressing. So there is no fixed register holding a GP, and PIC code is full of PC-relative offsets.

Figure 5.4(a) shows an ARMv7 assembly PIC fragment. To load the value at label .Ldata into memory with the instruction at .Lins2, a PC-relative address stored in a so-called literal pool in the .text section is first loaded into a register at .Lins1, and then used in the PC-relative memory access at .Lins.[1] All edges in the code fragments of Figure 5.4 correspond to offsets that are known at compile time. For that reason, they can be computed by the linker or protection tool, and stored as entries in the literal pools, or they can be encoded as immediate operands of instructions.

Suppose that the three instructions in red become mobile. Figure 5.4(b) shows the transformed static PIC. In this example, we assume that enough registers are available (like r6 in this fragment) to store temporary values. If not, additional spill code would be needed. Instead of the original code, the first two inserted instructions in red produce the address of the GMRT. The next instruction loads the address of the mobile block from its (fixed) index in the GMRT, and then control is transferred to that address. When the mobile code block is not yet present, control will be transferred to a stub that invokes the Binder with the requested block index instead. The Binder then invokes the Downloader and overwrites the address of the stub in the GMRT with that of the downloaded block.

Please notice that in the remaining static code of this example, there is absolutely no need to place the instruction at .Lins4 right after the inserted instructions, since the control transfer from the mobile code to that instruction will happen indirectly. Besides hiding the mobile code, this also opens up opportunities to obfuscate the control flow in the code that remains static. When code mobility is combined with code layout randomization in which independent code fragments (i.e., fragments that do not need to be allocated consecutively because there are no fall-through execution paths between the fragments) are reordered and

[1]The +8 in the PC-relative address is due to the ARM specification that a used PC equals the PC of the instruction that uses it plus eight.

spread throughout the whole text section, the fact that .Lins0 and .Lins4 belonged to the same basic block will no longer be apparent in the static code.

Figure 5.4(c) shows the offset-independent mobile code block that replaces the three instructions extracted from the static code. The single entry point of this code block (i.e., the address that will be stored in the GMRT by the Binder) is actually the third word in this block (marked by the .Lins1 label). The second word is an instruction that restores some registers and the first word is a kind of GP. In our current implementation, it points to the start of the statically allocated code and data of the binary in memory, i.e., to the .Ltext label that marks the start of the .text section. As this address is randomized by ASLR, it is unknown at compile time. Therefore it is the Binder's job to fill in this address in the blocks first word at run time, i.e., when the mobile code block is placed in the process' memory space.

Rather than relying on the PC and a PC-relative address loaded from a literal pool to access statically allocated data as the instruction at .Lins2 did in the original code fragment, the rewritten code in Figure 5.4(c) uses the .text GP stored in the first word of the block, and an .Ltext-relative address loaded from the literal pool. Likewise, to facilitate the jump from the end of the mobile code back to .Lins4 in the static code, that address of .Lins4 is computed using an .Ltext-relative address.

By combining the different redirection mechanisms discussed above, it is possible to rewrite all direct references, be it in direct memory accesses or in direct control flow transfers from mobile to static code or data, from static code to mobile code and even from mobile to mobile code.

To handle indirect references from static data to mobile code, however, we require another mechanism. This occurs when pointers to mobile code are stored inside static data sections or when they are computed on the fly, to be used in indirect control flow transfers to mobile code. Fundamentally, the problem with such references is that while the origin of the reference can accurately be identified (in source code or in binary code, as we will discuss in the next section), the points of use of those references cannot easily be identified accurately: Once some function pointer has been computed and stored in memory, it is very hard if not impossible in most programs (due to aliasing) to decide exactly where that pointer will be used in an indirect transfer. Run-time solutions to rewrite all potential indirect transfers where code pointers are used have been proposed in the SecondWrite binary code rewriting system and

in other designs [92], but all of them introduce a significant amount of code and data bloat, which we consider unacceptable in many usage scenarios.

Thus, rather than rewriting the code fragments that indirectly use references to mobile code, we propose to limit code mobility to regions that can only be reached through direct control flow transfers. In practice, this is straightforward: When we detect that a region we want to make mobile is accessed indirectly, an indirection pre-header is generated for this region. This pre-header consists simply of a direct branch to the original entry point of the region, which will later on be converted into an indirect branch. It then suffices to replace all indirect references to the region's original entry point (i.e., statically allocated code pointers or code pointer computations) by references to the indirection pre-header instead. This pre-header then remains static, thus avoiding the problem completely, while the entire region itself can still become mobile.

With the discussed transformation, the code mobility protection can be applied widely. It is clear that rewriting mobile code references to static code or data into offset-independent code can introduce significant overhead, in particular when additional registers have to be freed. We will evaluate this overhead in the evaluation section.

5.3 Automated Tool Support

It is not trivial to make the described form of code mobility generally applicable and usable for developers. They may not have the time to invest in complex tools, and may have to operate in industrial environments that put a lot of restrictions on the used compilers and development tools.

In the ASPIRE project, we therefore designed a plugin-based tool flow that allows a developer to annotate the source code that he wants to make mobile, and that can be used in combination with open-source compilers like LLVM and GCC, as well as with proprietary compilers such as ARM RVDS. In this tool flow, we make use of three sets of tools, which corresponds to the three phases depicted in Figure 5.5.

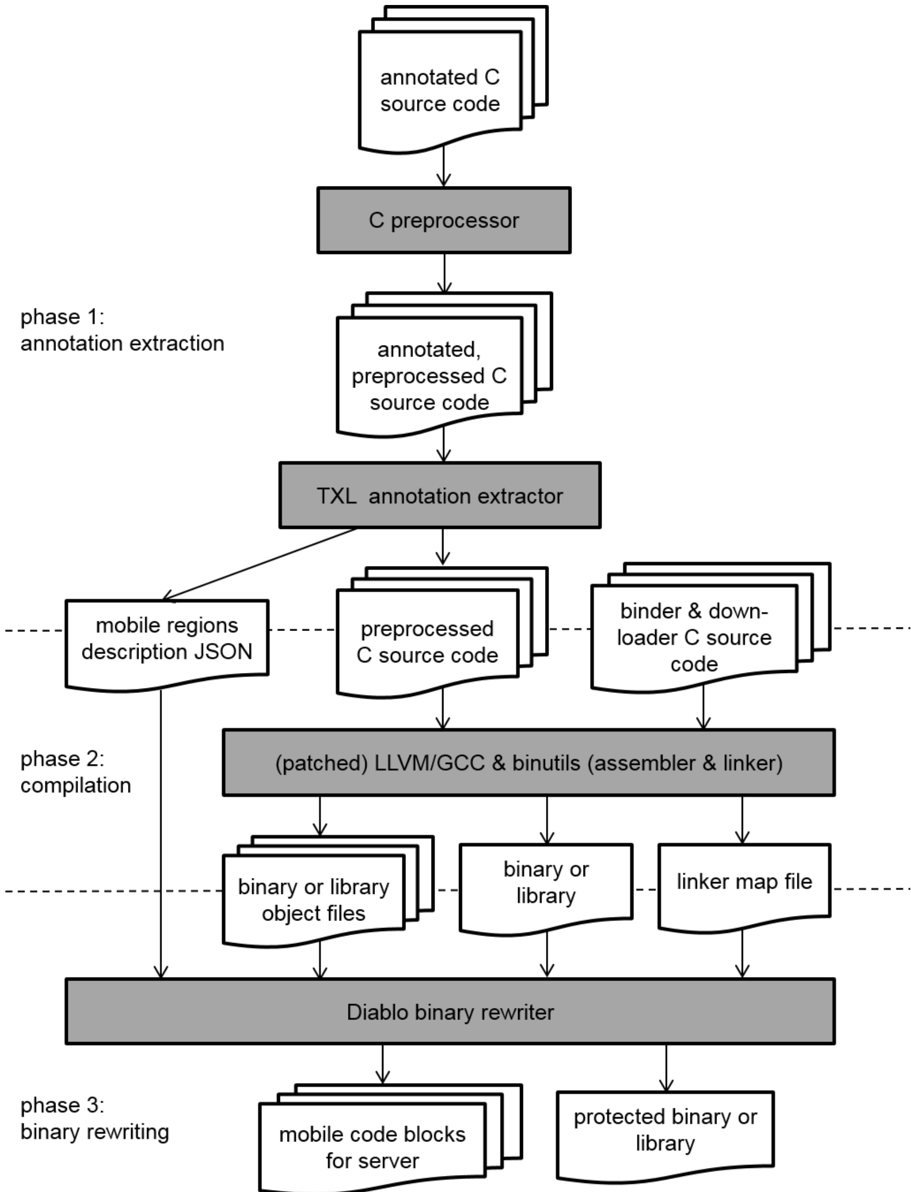


Figure 5.5: Code mobility tool flow

5.3.1 Specifying Mobile Regions

First, we use source code analysis tools based on TXL [31] to extract annotations from the C source code.[2] The annotations are inserted by the programmer in the form of _Pragma directives as defined in the C standard since C99. Listing 5.1 depicts an example. The ASPIRE begin and ASPIRE end pragmas denote a code region to be protected, in this case with the code mobility protection. Many other protections are also supported by the full ASPIRE tool chain, but are out of scope. The regions mark by the pragmas have to follow the scoping rules of { ... } blocks in C. This is no problem, however, since C programmers are of course very familiar with this scoping.

```
1  int f(x) {
2    int y,z,i;
3    y = 2 * x;
4    z = 0;
5
6    _Pragma("Aspire␣begin␣protection(mobility)");
7    for (i=0; i < y; i++)
8      z += x << i;
9    _Pragma("ASPIRE␣end");
10
11   z /= 2;
12 }
```

Listing 5.1: Annotation code example

The analysis tool extracts the annotations from preprocessed source code, and produces a JSON file that identifies the regions by means of their path and file names, their line numbers, the functions in which the regions were found, as well as the protections that were specified for each region. The tool also removes the ASPIRE pragmas, such that compilers will not complain about unknown pragmas.

In addition, the user can edit the JSON file, for example to mark additional functions that need to be made mobile. Wildcards can be used to denote multiple functions and multiple files. This eases experimenting with regions, for example to find a good balance between overhead and protection. Moreover, it also allows the user to specify functions that need to be made mobile that are not part of the original application.

[2]For the time being, we only support C code because the TXL grammar we use is limited to C. C++ grammars exist as well, however, so this is no fundamental limitation.

Such functions can be injected into the application to implement other protections, such as code guards, by other plug-in components in a protection tool flow. A range of such components is documented in some of the public ASPIRE deliverables available on the ASPIRE website [102].

5.3.2 Compilation With Standard Compilers

In the second phase, the preprocessed code without the pragma is compiled, assembled, and linked into a binary. The compiler, assembler, and linker are instructed to generate debug information in the produced object files and final binary, as well as a linker map file. All compilers and linkers we know can do so. The linker map and the debug information, as well as sufficient relocation and symbol information, need to be available in support of the third phase: a link-time rewriting process.

This requirement of sufficient relocation and symbol information serves to allow the link-time rewriter to rewrite the generated code conservatively, i.e., without breaking the original program behavior. For example, so-called mapping symbols are needed that identify data present in the code sections. As another example, relocations should not be relaxed because important information is lost during the relaxation process. A standard linker does not suffer from that loss, but an advanced link-time rewriter does. Some compilers and binary utilities already produce sufficient information, such as ARM's proprietary compilers. Others, like GCC, LLVM, and GNU binutils do not produce it out of the box. However, about 10 small patches—touching only few lines of code in total—suffice to make them produce it.

5.3.3 Binary Code Rewriting

The third phase then consists of the actual extraction of mobile code blocks and the rewriting of all code to insert the necessary indirections. For this, we rely on the Diablo link-time rewriter from Ghent University (http://diablo.elis.ugent.be) [104]. This rewriter has already been used for many different applications, including fault injection mitigation; obfuscation; kernel customization; memory safety; software diversity; and program compaction, optimization, and instrumentation. In the ASPIRE project and tool chain, it applies many protections besides code mobility, including control flow obfuscation, code guards, ISA randomization, and anti-debugging techniques.

The internal program representation in Diablo is a WPCFG (Whole-Program Control Flow Graph). This WPCFG includes the CFGs of all functions in the program, as well as call and return edges, and additional so-called hell nodes and hell edges that can conservatively model unknown code—such as library code—and unknown (or at least not precisely known) control flow, such as calls through function pointers.

Diablo first builds the WPCFG of the original executable or library by disassembling it with the help of the linker map file and the original object files (and the relocation and symbol information contained in them). After this, it annotates the nodes in the WPCFG with line number information that it extracts from the debug information.

In the WPCFG, Diablo then identifies the regions specified in the JSON annotations file. If a region has multiple entry points, it is split in multiple single-entry regions. Moreover, if a region is reachable through indirect control flow transfers such as calls through function pointers, the already mentioned form of pre-headers is inserted in the code. At that point, all regions are single-entry regions that are only entered through direct control flow transfers. Diablo then rewrites all those direct transfers into indirect ones that go through the Binder's redirection tables.

Next, the code inside each region is rewritten to replace all transfers and references to other mobile code regions or to static code and data by indirect, offset-independent references. Typically, the offset-independent references require more instructions, and often they need to store temporary (relative and absolute) addresses in registers. Diablo relies on its bi-directional, inter-procedural, context-sensitive liveness analysis to maximally find available registers in the code. If none are available at some point, the necessary number of registers is freed by inserting register spills to the stack.

The rewritten regions are then extracted from the WPCFG, and migrated to separate WPCFGs, one per region. Entries and exits to and from these separate WPCFGs are modeled conservatively with hell edges, as if each region corresponds to a library that can be called by unknown application code. Once the original WPCFG has been split in multiple ones this way, each of them can still be transformed independently: The hell edges ensure that dependencies between the blocks are respected automatically.

For each extracted region, multiple WPCFGs can actually be translated, which are then diversified with the stochastic diversification techniques previously documented in literature [29, 30], including opaque

predicates, branch functions, flattening, and code layout randomization. Obviously, those protections can also be applied to the application code that remains static, including the Binder and the Downloader.

5.3.4 Current Status and Limitations

Most Diablo transformations—including the aforementioned diversification transformations—can handle both the fixed-width 32-bit ARM code and mixed-width Thumb2 instruction sets of the ARMv7 architecture, as well as combinations of these two sets. The current tool support for producing offset-independent code, however, only handles the 32-bit ARM subset. This is not a fundamental limitation, only a matter of engineering effort.

Diablo in general can handle position-dependent as well as position-independent code, and so can the mobile code support we implemented on top of Diablo. There is one exception, however: The current tool cannot yet convert position-dependent switch tables (a.k.a. branch tables) into position-independent or offset-independent ones. WPCFG fragments containing such tables are therefore excluded. This is also a matter of engineering effort, not a fundamental issue.

The whole tool flow, including the code mobility support, has already been extensively tested with LLVM 3.3 and 3.4, as well as with GCC 4.8.1 and 4.6.4, and binutils 2.23.2 for ARMv7 software executed on Linaro Linux, as well as with the Android NDK API level 18 (including the already mentioned compilers and binutils) for software running on Android JellyBean (4.3). Both stand-alone executables (from the SPEC2006 benchmark suite, as well as system utilities) as well as shared libraries have been tested, including security-sensitive plug-ins for the Android DRM Framework. In terms of structure and other requirements, such as the use of GNU_STACK and GNU_RELRO segments, the generated binaries conform to the strict security requirements of SELinux.

For the moment, mobile blocks can not share pages yet. This is because when a new mobile block has to be loaded into memory, the page(s) it would be placed on would have to be mapped first to non-executable and then back to executable; in Android systems this would require a rooted device. Next to that, in case the code from another mobile block present on one of these pages is being executed in another thread at the same moment, this thread would generate a segmentation fault. A future solution for this problem would be to install a signal handler for segmentation faults in the binary that suspends this thread

Config		Latency	Block download	Libquantum 50% mobile
	Average	0.12	9.36	369.37
Localhost	Std Dev	0.03	6.63	66.28
	Overhead			+1.97%
	Average	0.32	6.98	370.45
LAN	Std Dev	0.02	1.46	65.74
	Overhead			+2.27%
	Average	3.43	29.64	401.56
WiFi	Std Dev	2.81	24.49	68.36
	Overhead			+10.86%
	Average	134.27	228.87	659.54
3G	Std Dev	119.58	154.44	173.42
	Overhead			+82.08%

Table 5.1: Summary of performance overhead (in *ms*)

and resumes it when the page is executable again. For this same reason there is also no support yet for removing mobile blocks from memory, but this feature can eventually be added with minimal effort.

5.3.5 Testing

To ensure rewriting binaries with Diablo and splitting off mobile blocks didn't introduce any bugs, we verified whether rewritten applications still work correctly. For this purpose, a stub Downloader without an actual network connection was used, which simply maps the requested mobile block from the disk. This testing was done for both ARM Linux and Android, using Position Independent Executables. The applications used are those from the SPEC CPU 2006 benchmark. The testing was done by simply making mobile every named function present in the binary (if that was possible). As an example, more than 3000 functions were made mobile for the 403.gcc benchmark.

5.4 Performance Analysis

Our performance analysis was carried out for our code mobility framework on three case studies written in the C and C++ languages. These three, taken from the SPEC CPU 2006 benchmarks, were libquantum, namd, and milc. Tests were performed on a SABRE Lite i.MX6 board with a Quad-Core ARM Cortex A9 processor at 1 GHz clock speed, with 1 GByte of 64-bit wide DDR3 at 532 MHz.

To evaluate the steady-state overhead of the mobile code transformations, i.e., the performance overhead on an application in which all executed mobile code blocks have already been downloaded, we used a customized version of Diablo. It transforms the applications by applying the GMRT indirection and by making all mobile code offset-independent as described in Section 5.2, but it leaves the mobile code blocks in the binary's static code sections, thus avoiding the dumping of the mobile blocks.

To evaluate the latency that downloading the blocks might incur, we tested four different network scenarios: Localhost, LAN, WiFi, and 3G. In the localhost scenario, all components were configured such that both the mobility server and client reside on the same test virtual machine: All communications took place locally, in order to exclude the influence of transmission delays from collected data, and in order to have a reference for the other configurations.

In the LAN configuration, we tested the code on a 100 Mbps wired network; in the WiFi configuration we tested the code on a 54 Mbps wireless network, while in the 3G scenario we tested it on a HSDPA mobile network.

We measured the *latency*, i.e. the time required to establish a new TCP connection, whenever a new code block has to be downloaded; then we calculated the *block download time* to measure the time needed to download a mobile block on different network configurations. For the block download we made an arbitrary function mobile and measured the time needed to transfer it from the server to the client. The chosen function has a code size of 412 bytes.

Each experiment was repeated 500 times to collect data and we calculated the average and standard deviation for both the latency and block download time (see Table 5.1); for the latency experiments we only ran the code 100 times. The last column of Table 5.1 represents the total execution time of a mobile version of the libquantum application. For this experiment, we selected a hot function that by itself represents circa 50% of the executed operations, and made this function mobile.

Since most of the overhead comes from downloading blocks, which happens only once per mobile code block in our current implementation, and because our Android boards are relatively slow, we used the test SPEC inputs in our experiments. As expected, the worst overhead (82%) is found in case of mobile network connection while in a LAN scenario the overhead is as low as 2%.

Execution time	Average	Std Dev	Overhead
libquantum			
original	362.23	63.11	
20%	363.18	67.93	+0.26%
50%	355.73	67.14	-1.80%
100%	394.80	62.06	+8.99%
milc			
original	85,697.45	29.98	
20%	85,417.24	46.73	-0.33%
50%	85,985.24	46.73	+0.34%
100%	88,557.82	133.17	+3.34%
namd			
original	92,729.70	107.89	
20%	93,403.56	124.05	+0.73%
50%	94,383.00	115.48	+1.78%
100%	95,503.73	119.98	+2.99%

Table 5.2: Summary of computational overhead (in *ms*)

Table 5.2 shows the performance once all mobile code blocks have been downloaded, i.e., when the redirection via the Binder's GMRT is applied to all the fragments of an application.

For each benchmark application scenario the average total execution time and its standard deviation are provided, overhead is computed as the increment of execution time with respect to the original application, where no functions have been instrumented to become mobile. Each row indicates a different experiment with a significant percentage (20%, 50%, and 100%) of indirection/mobility, evaluated as the number of instructions executed in mobile functions over total number of executed instructions.

In both the 20% and 50% coverage example we can see that the overhead is very low and sometimes even less than zero. This is due to the optimizations made to the code by Diablo. Only when 100% of the application's functions are made "mobile", forcing the indirection, do we see a significant overhead occurring.

5.5 Related Work

Different online protections use dynamic code replacement to periodically replace the copy of the program running on the untrusted machine with the goal of limiting the amount of time that the attacker has to reverse engineer the application. The replacement may be implemented for the functional part of the program, and/or for the protection techniques used to protect it [64]. Collberg et al. [26, 27] and Falcarin et al. [40] proposed the continuous replacement of Java and binary code respectively, in which the remote trusted entity frequently sends a set of new code fragments to the untrusted machine. The technique of Collberg et al. has some limitations as it relies on CIL (Common Intermediate Language), however. This restricts the scenarios in which the technique is usable (e.g., not with dynamically linked libraries), their composability with other protections, and the granularity of the code blocks. Although the technique of Falcarin et al. operates on binary code, the code blocks are generated in a rather arbitrary manner: The binary's `.text` section is simply chopped into a set of blocks. This means one function might cover multiple blocks, and one block might contain pieces of multiple functions. Consequently, the technique does not allow selectively making only certain parts of the program's functionality mobile, hindering composability with other protections at the binary level.

Previous work in Java implemented dynamic replacement of protection code implementing code mobility features on top of dynamic aspect-oriented platforms [39] or by ad-hoc JVM extensions [96].

5.6 Conclusions and Future Work

The main contribution of our work is the definition of a new software protection relying on code mobility and the full automation of mobile code block generation. Our solution shows that splitting a program into code blocks transmitted via network by a trusted server is a suitable and low-cost software protection that can be useful in defending software programs from reverse engineering. Our protection creates problems for common reverse engineering tools and makes the code comprehension task more difficult for the attacker. The source code can be found together with the entire ASPIRE framework at `https://github.com/aspire-fp7/framework`.

The proposed solution provides stronger protection than the one described in previous work. First of all, the addresses at which the

mobile code is downloaded will differ from one run of the program to another. This makes all kinds of dynamic attacks more difficult. Secondly, almost all the necessary support is already available to free the allocated memory of mobile code blocks, and to restore the addresses in the look-up table to their original values, i.e., the stub addresses. Once this is implemented, it will allow us to make sure that not all mobile code is present at once, and to let multiple different mobile code blocks occupy the same memory addresses during a single run of a program. The fact that addresses in the program's address space then no longer map onto instructions in a one-to-one mapping also complicates many dynamic and hybrid attacks. This is because many tools such as IDA Pro are engineered around the central notion that every code byte and address corresponds to at most one instruction.

Further research will be devoted to integrate code mobility with remote attestation in order to integrate tamper-detection techniques, improving the level of protection. Another line of research we want to explore is the combination of code mobility and software diversity. Software diversity creates many different copies from an initial version of a program: Each copy of the protected program is different in its binary shape, but is functionally equivalent to other copies [73]. Thus, attacks designed to work with one version might not work with other customized versions. Along with parameterizing the binary layout (diversity in space) we will explore how to extend it with diversity in time, by making code mobility even more configurable, by randomizing the binary structure [110] and parameterizing the number and size of code blocks and their duration in the client code before expiring and being replaced by a new version.

Chapter 6

Native Code Renewability

MATE attackers use debuggers, emulators, custom OSs, analysis tools, etc. to reverse engineer or tamper with software distributed by providers of software, service, and content (as discussed in Section 2.2.3). Software protection techniques aim at protecting the integrity and confidentiality of the provider's assets in the software by making it harder to reverse engineer and tamper with [41, 88]. Each protection technique only affects a small set of attack vectors, and applying only a few will merely divert the attacker's attention to the remaining unprotected attack vectors. Thus, multiple techniques need to be combined to ensure all these possible paths-of-least-resistance are hardened.

Overall, protections aim for (i) increasing the effort needed to identify successful attack vectors; (ii) increasing the effort needed to manually exploit these attack vectors (iii) increasing the effort needed to automate and scale-up their exploitation; (iv) minimizing the number of instances on which automated attacks can be deployed; (v) reducing the window of opportunity for generating income from an attack.

Protections hence need to be diversified (as we discussed in Section 2.3.4), such that they maintain a level of resilience and different versions can be generated of the same functionality. Defenders need a mechanism to renew (i.e., update) assets and protections in the field such that the attack vector identification has to be re-done frequently, the value of assets decreases rapidly, and the protections' behavior varies over time. If temporal variation is unpredictable, attackers always need to take into account all protections to remain undetected and successful. This furthermore means that not all protections need to be active at the same time, which can allow the run-time overhead to remain acceptable.

This chapter presents the ASPIRE renewability framework for delivering renewability to the native executables and libraries that often implement the security-sensitive functionality of mobile applications. Our framework leverages existing diversity techniques [73] and exploits renewability opportunities for protection techniques to generate the required variation. The protection techniques used were developed in the context of the ASPIRE project, by various partners and separately from this framework. The renewability framework builds on the code mobility technique (presented in Chapter 5). Our main contributions are:

- The framework architecture to support many forms of software renewability.
- The supporting tool flow.
- Concrete deployments of the framework that mitigate specific attacks by making existing protections renewable.
- The evaluation of a prototype implementation.

Section 6.1 discusses the MATE attack model. Section 6.2 presents the overall framework design and architecture, after which Sections 6.3 and 6.4 discuss specific features. Section 6.5 presents the tool flow to support automated deployment of the framework. Section 6.6 discusses concrete uses of the framework to mitigate a variety of concrete MATE attack steps. Section 6.7 evaluates the proposed renewability framework and the prototype implementation in terms of robustness, overhead, and scalability. After related work is discussed in Section 6.8, conclusions are drawn and future work is discussed in Section 6.9.

6.1 Attack Model

We aim to protect software against MATE attacks. In their labs, MATE attackers have full access to—and full control over—the software under attack, as well as over the system on which the software runs. They can use static analysis tools, emulators, debuggers, custom OSs and all kinds of other hacking tools. The attacks are looking to break the integrity and confidentiality requirements of assets, e.g., to steal keys or intellectual property, and to break license checks. They do so by means of reverse engineering and by tampering with the code and its execution.

We focus on mobile applications distributed by providers of content, software, and services. Often, their GUI parts are implemented in

managed languages such as Java. Because of the ease with which, e.g., Java bytecode can be reverse-engineered, and because of performance concerns, the security-sensitive assets are typically still implemented in dynamically linked, native libraries that are packaged with, e.g., the Java apps. The software under attack therefore consists of native binary files (this includes both dynamically linked libraries as well as stand-alone executables). Because of the economic value of the assets, we assume software protection techniques are deployed in and on the native code [88].

We only target always-online applications, such as video streaming apps or edge apps that connect to cloud servers. While this is a limitation, the omnipresence of wireless networks (4G, 5G, WiFi) has resulted in a big enough market to develop protections that exploit the always-online feature.

Our protections target economically driven attackers. We aim at increasing their attack investment cost, at lowering their profit, and at tilting the balance between the two. The protection is effective when the attackers expect a negative return on investment before they even start the attack or while they are still pursuing it, as well as when they expect a higher return on investment from attacking other providers' software. The protections then stopped the attackers before they had a chance to succeed. Even if the attackers succeeded, though, the protections can have delayed them enough for the provider to make a healthy profit of the assets. In that case as well, the protections can be considered successful.

In their lab, MATE attackers execute an attack strategy and a series of attack steps. The strategy is adapted on the fly, based on: the results of previous attack steps; hypotheses that the attackers formulate and test regarding assets, deployed protections, other relevant features of the software under attack (such as the locations of relevant code and data); and the perceived path of least resistance. With the perceived path of least resistance, we mean the sequence of future attack steps that the MATE attackers consider the most efficient and effective to pursue given their expertise, skills, and tool availability. We refer to existing literature for more information and models of the attack processes as obtained through empirical experiments with various kinds of attackers on various kinds of assets [16]. In the context of this work, one important aspect to point out is that in the eyes of MATE attackers, many seemingly uninteresting artifacts of software (system calls, control flow

structures, ...) are in fact interesting, because they can serve as hooks for the attackers to guide their search to the really interesting code.

To be effective, protections should cover as many attack paths as possible that might be paths-of-least-resistance for certain attackers. The protections can achieve this by making the individual attack steps on the paths more expensive or time-consuming, by requiring extra attack steps, or by preventing certain attack steps and the automation thereof. Section 6.6 will discuss several concrete attack steps against which protections exist that can be made more effective by making them renewable with the presented framework and architecture. In general, these steps are attack vector identification and attack vector exploitation steps that require a certain amount of repeatability, such as the iterative development and later use of customized scripts that work well as long as the software they operate on remains the same.

It is commonly accepted that sufficient protection can only be achieved by combining many protections in a layered fashion. The deployed protections then become assets themselves, protecting the original assets, the artifacts that attackers can hook onto, and each other. The value of the proposed renewability framework and architecture hence cannot be judged in isolation. The supported forms of renewability are supposed to be combined with other protections that protect against additional attack vectors, and that protect the components of the renewability implementation. The communication to a secure server to download renewed assets and protections, for instance, is supposed to be protected by sufficiently strong cryptography, of which the keys are protected through white-box cryptography, of which the code is obfuscated to prevent static reverse engineering, and anti-debugging techniques to protect against dynamic reverse engineering. Similarly, remote attestation is supposed to be used for hampering replay attacks, i.e., for checking that a client application actually executes freshly downloaded code rather than old copies stored on disk by an attacker.

In the ASPIRE project, we reached the necessary composability of renewability with other protections in an open-sourced protection tool chain [5]. The renewability framework and architecture presented here are only one of several novel aspects of that tool chain. Composability of all kinds of protections in the tool chain is out of scope of our work.

Our MATE attack model neglects hardware-based protections. Off-the-shelf processors offer limited protection against MATE attacks. SGX enclaves can leak information in contexts similar to MATE attacks [10, 76]. Furthermore, they are restricted in their interaction with outside

components, so they cannot protect all code. TrustZone [2] is only effective in well-configured systems. In a lab, a MATE attacker can easily disable the protection. Furthermore, many lower-end devices lack hardware protection. For those, software-only protection is the only available option. Moreover, hardware-based protection is considered a risk by some, because it is expensive and at the same time not renewable [56]. The reason for this is that when a hardware defense mechanism is broken at some point, e.g., because implementation bugs are discovered, it is typically very hard—if not impossible—to fix it, so all systems relying on that hardware are vulnerable from then on. Software renewability offers a complementary solution for such scenarios.

6.2 Architecture

Figure 6.1 visualizes the ASPIRE renewability architecture. It is based on code and data mobility, which builds on the existing concept of code mobility (as presented in Chapter 5). From the binary file of a client app or library that needs to be protected, parts of the statically allocated code and data sections are extracted. These parts correspond to code that will need to be renewed dynamically, i.e., when the app or library is actually running. By simply removing this code and data, it is already protected against purely static MATE attacks. The code and static data (Client Application Code and Data in Figure 6.1) that remains in the binary is extended with support code: Communication Logic, a Downloader, and a Binder.

The Communication Logic implements protection-agnostic communication and protocols to the Server Portal. Its prototype implementation offers a simple request protocol and a WebSockets protocol [63] for protections that need occasional connections and/or server-initiated communications and a persistent connection. (Replacing WebSockets with a more secure implementation is orthogonal to this work.)

The Downloader implements the communication. Upon requests from the Binder, it connects to the Mobility Server to download mobile code and data blocks. The downloaded blocks are then mapped at randomized locations on the heap of the running client. In basic code mobility, these blocks correspond to individual code fragments extracted from the statically allocated code of the protected app.

The Binder initiates the download requests on demand and ensures that all control transfers and accesses to and from downloaded code and

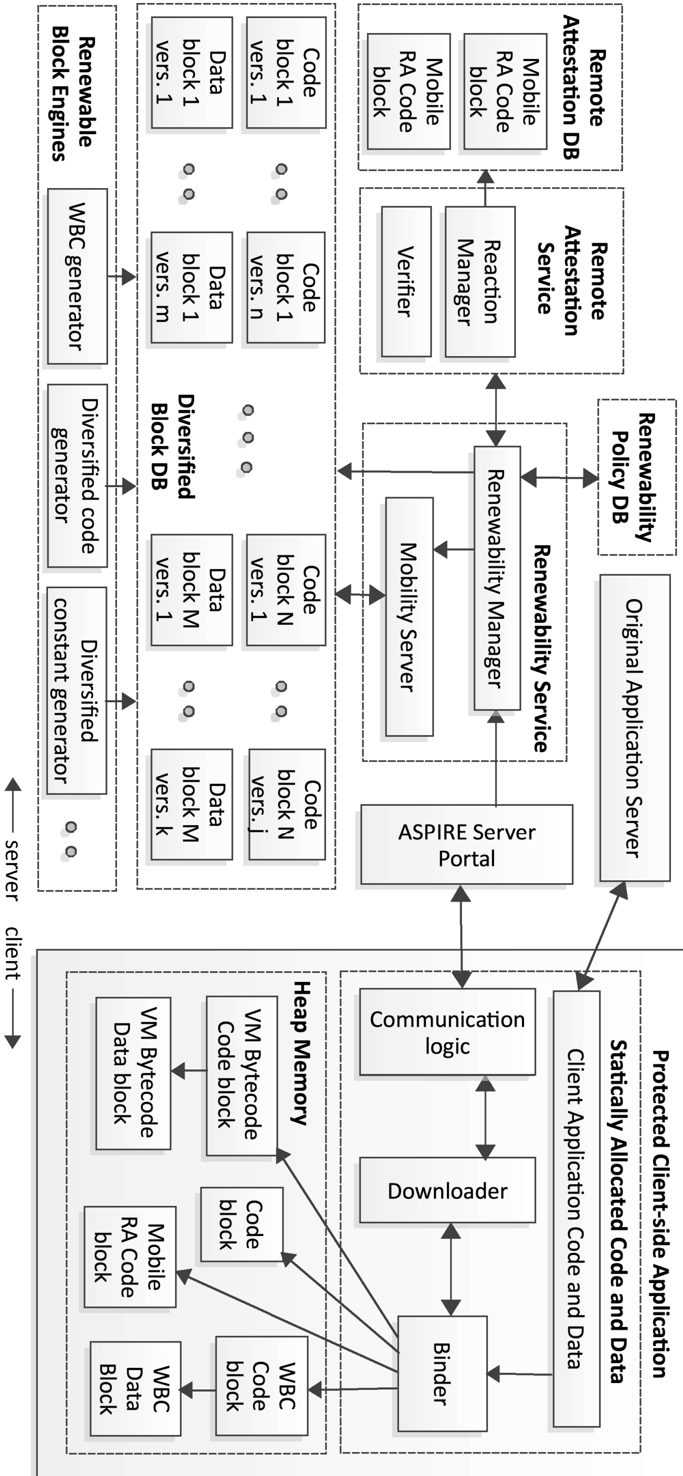


Figure 6.1: ASPIRE Renewability Architecture. On the right, the native application from which static code and data has been extracted and to which code mobility components have been added. On the left, the original application server in case the application was an online application, and all server-side components that implement the many forms of renewability.

data execute correctly. Each transfer *into* a mobile block is redirected via a stub that transfers control to the block's address, which it finds in a GMRT. Until a block has been downloaded, the found address is that of another stub that invokes the Downloader with the correct input and that performs the necessary allocation and bookkeeping. This includes replacing the stub's address with the block's address in the GMRT, and then continuing execution at the entry of the block. All transfers *out of* a block are transformed into *offset-independent code* by adding a level of indirection (as discussed in Section 5.2). single-entry code regions.

In basic code mobility, a downloaded mobile block remains mapped on the heap of the running process until it halts. To support advanced forms of renewability, we extended the Binder to support the flushing of mapped blocks and subsequent re-downloading of renewed, different versions of those blocks. We also extended it to support mobile data, which is necessary to support several interesting forms of renewability that will be discussed later.

The protection-agnostic ASPIRE Server Portal forwards communications between clients protected with (multiple) online protections and the corresponding services. In our prototype, it also supports client-server code splitting [15] and remote attestation techniques [105].

The Renewability Manager selects which mobile code and data blocks need to be delivered to a running client. By varying the mobile block versions that are delivered to different clients and at different times, the assets and protections implemented by that mobile code and data can be renewed. The Mobility Server takes care of the actual delivery and interaction with the Downloader in stateless communication: The Server does not keep track of existing sessions with clients for the sake of scalability; it just serves the right block whenever a request arrives from one of the clients, based on the policy implemented by the Manager.

The mobile code and data blocks are stored in a database (DB). That Diversified Block DB can hold multiple, diversified versions of each block. For most forms of renewability, the different mobile blocks and the different versions thereof, are independent of each other. This is the case because either all versions of a block implement exactly the same semantics, or because one block's semantics is independent of the other blocks' semantics. For some forms of renewability, however, there may exist dependencies between the blocks. Some interesting cases are discussed later.

Different server-side code generators (Renewable Block Engines) produce diversified mobile blocks. Depending on the renewability poli-

cies, these generators can generate blocks a priori or on demand. For example, in case a policy only aims at delivering different versions of a block with the exact same semantics, the DB can be populated a priori. If specific versions need to be generated to react to events, they can instead be generated on demand. Obviously, if the event to react to is an actual request from a running app, the on-demand generation will result in a higher response time.

The renewable code engines are application-dependent, as they generate mobile blocks matching the code and data fragments that were extracted from the static binary of the client software. Section 6.5 discusses how these engines are generated. Obviously, but not drawn in Figure 6.1 to keep it simple, the Renewability Manager also has to interact with the code engines to know what code is in the DB, and to trigger on-demand generation of blocks.

Furthermore, the Renewability Manager can interact with other protection servers. Figure 6.1 includes an example Remote Attestation Service. That interaction can be exploited in two ways. First, the renewability policy itself can interact with the other protection server. In the case of remote attestation, the interactions can involve notifications of failed attestations, and communication about the mobile blocks that were delivered to the client such that the remote attestation can attest them. Secondly, the other online protection might also have client-side protection components, such as specific hash functions used in remote attestation code guards that need to be delivered as mobile blocks via the Renewability Manager. Figure 6.1 shows this for mobile remote attestation blocks. Note that the difference between the mobile remote attestation Code blocks and the Code and Data blocks in the Diversified Block DB is that the former are application-independent components of a deployed protection, while the latter implement original client-side functionality.

6.3 Integrating Renewability into Existing Applications

In order to integrate renewability into existing applications, there are two choices that need to be made: (1) where and how decisions will be made to renew blocks, and (2) decide how these decisions will be communicated with the client application. We call the former the renewability policies, and the latter the renewability communication design. We will

now discuss the spectrum of options and trade-offs that can be made for both kinds of policy.

6.3.1 Renewability Policies

Renewability policies define when a client needs to discard and replace downloaded mobile blocks with renewed ones. The decision to renew a block can either be made server-side, or it can be made in the client itself. When considering client-side decisions, these can be made either without external inputs (the logic is set in stone), or it can be that the application implements a policy that has been dictated to it by the Renewability Server. Either way, a MATE attacker might be able to learn or even subvert the renewability policies. A major advantage of server-side decisions, is that the client then only learns about the concrete decisions taken by a policy (i.e., the flushing commands), and not about the rules that lead to these decisions. On top of that, a persistent, server-initiated connection to pass policy decisions to the client enables dynamic policies that can be adapted on the fly. A compromise between these two approaches would be to send a policy description to the client with each served block. A policy would then be immutable in between the delivery of blocks. In the rest of this section we will consider server-based renewability policies.

When the renewability policy is implemented on the server, we can either make this an application-agnostic *decoupled policy*, or a *coupled policy* that is tightly integrated with the Original Application Server.

In the case of decoupled policies, no changes need to be made to the application source code when compared to the original, non-renewable ASPIRE architecture: It suffices to add annotations in the source code. The decision of whether and when to renew certain blocks is made completely independently from the application, by the server-side Renewability Manager. When this component decides to renew a certain block in a certain application instance, it sends a flushing command to that application instance.

Implementing policies in a decoupled manner on the renewability server somewhat restricts the manner in which the renewability policies can react to events occurring in the protected application. We would however argue that there is still quite a lot of leeway left to react: We can compose different (application-agnostic) protection techniques and change the policies based on the state and observations of these other protection techniques. For example, in our prototype implementation,

the integrity violations that are observed by the remote attestation component are passed on to the renewability server, which changes its policy based on these observations. Several reactions are possible: the Mobility Server can stop serving blocks, the client can be notified in the next communication through the ASPIRE Server Portal, or the Original Application Server can be informed that it should stop delivering content [105]. Furthermore, it has already been demonstrated in the ASPIRE project that other protections, such as client-server code splitting, can be used to let a protection server keep track of different events in the client [15, 105]. Decoupled policies thus lead to a clear separation of concerns.

Alternatively, in a *coupled* policy, the (server-side) application logic is tightly integrated with the renewability policy. The decisions of which (specific instances of) blocks to flush can be based explicitly on the state in which a specific client happens to be, and can be made to coincide with other actions that are taken in the application server. For example, in the case of a streaming video application, the streaming server can be integrated in the renewability policy so as to force the client to download a different decoder function after a specified number of video frames have been sent. The application server can thereafter send differently encrypted or encoded frames, which the old decoder function is not able to decode. This option offers the vendor much more control over the renewability policy. The price, however, is a sharp dent in the separation of concerns, as the protection is now to a large degree hard-coded in the application source code.

6.3.2 Renewability Communication Design

After a decision has been made by the renewability policy, it has to be communicated to the protected application. We elaborate on two possible designs for this communication: an application-agnostic, decoupled communication design, and a tightly-integrated, coupled communication design.

In the case of a decoupled communication design, we can build on existing the ASPIRE components: The client-side Binder component handles flushing commands received from the server, while the server-side Renewability Manager sets up a bi-directional connection with the application for future, server-initiated flush requests. Flushing consists of the deallocation of individually specified—or even all—mobile blocks, the resetting of addresses in the GMRT, and informing the server of its completion. In this manner, the server can be aware that flushing is not

happening, and suspect the client is being tampered with. When the client fails to confirm the flush request within a given time, an appropriate reaction can be activated.

Conversely, in a coupled communication design, both the protected application logic and server logic (including the existing protocols) are augmented in order to support all communication logic that is required for renewability (e.g., receiving new blocks, receiving and confirming flush requests, etc.). The application server needs to communicate directly with the Renewability Service to obtain mobile blocks, and will need to embed those blocks—together with descriptions of renewability actions—in the packets sent to the client application. This is practical, e.g., for streaming video applications, where mobile blocks can be sent together with the video frame data. The client application is then also adapted by adding the necessary functionality—in the client's source code—to handle the extra content of packets coming in from the application server, and, if necessary, to respond by inserting responses in outgoing packets.

6.4 Mobile Data Blocks

When code fragments are made mobile, it suffices to replace all call sites with stubs, and use a simple indirection step to either download the code fragment, or to execute it immediately. In contrast, data blocks can be accessed from any location in the program that can dereference a pointer to the block. Due to the problem of aliasing [35], precisely identifying all those locations for all potentially useful mobile data blocks is impossible. Even if it would be possible, adapting all code to ensure that a data block is downloaded before it is accessed would introduce an unacceptably high overhead.

The solution is not to adapt the program locations where pointers are potentially consumed, but instead to adapt the locations where pointers to the data blocks are produced.

To produce an address of a statically allocated data section during the execution of a program, three options exist. First, the address of some section can be available in the statically allocated data of the program, i.e., in another data section. Such cases are trivial to identify in object files, as they are marked in the relocation information that linkers consume to relocate such addresses. The second option is that the address of some data or data section is computed in a code fragment of the same binary.

Those cases can also be identified through the relocation information. The third option is that the address of some section is produced or statically stored in another binary (e.g., a library) that is loaded into the same process. That case can only occur when at least one symbol in the section at hand is exported from the binary containing the section. If no such symbol is exported, it is impossible for the dynamic loader to let another binary relocate a symbolic reference to the section.

In short, data sections linked into a binary become accessible if and only if (i) a symbol residing in the section is exported from the linked binary, or (ii) a relocatable address residing in the section is stored in another section that is accessible, or (iii) a relocatable address residing in the section is produced in code being executed.

These conditions for being accessible are already used by linkers. The GNU linker option `-gc-sections` [52] lets it garbage collect all inaccessible sections. Link-time program compaction techniques have pushed this further by combining inaccessible section analysis with whole-program unreachable code analysis [33]. Our support for data mobility relies on the same principle: We limit mobility to data blocks that (i) correspond to full data sections in the object files and that (ii) become accessible only because their addresses are computed in code that is marked to become mobile and possibly renewable. We exclude data sections that become accessible because their addresses are stored in other data sections or because they are exported.

The limitation to full data sections poses no problems for the forms of renewability that will be discussed in Section 6.6. Most compilers offer a compilation flag `-fdata-sections` to store statically allocated variables in a separate data section each. So the granularity for making data mobile is that of individual global variables. This suits our purpose.

The second limitation poses no problems for the forms of renewability we currently support either, because we only make data mobile in connection with mobile code. When a source code fragment is annotated with code mobility pragmas, and the option of data mobility is enabled in the pragma, the link-time rewriter automatically identifies all data sections that become accessible only through addresses produced in that code fragment. Those data sections are then made mobile together with the code fragment. Our data mobility can hence be seen as code mobility where statically allocated data "owned" by a mobile code fragment becomes part of its mobile block. In Figure 6.1, this is visualized with arrows from mobile code blocks to mobile data blocks in the heap memory region of the client-side application. Remember, those arrows do not

indicate that only the mobile code blocks can access the data. They only indicate that the mobile code blocks contain the code fragments that generate pointers to the mobile data blocks in the program state as the mobile block is executed, thus making the mobile data blocks accessible to other code fragments.

Because the Binder and the injected stubs ensure that each mobile code fragment is downloaded before it is executed, and because they download the mobile data together with the code that can produce the data's address, they also ensure that mobile data is downloaded before any pointer to it is generated or used to access the data.

6.5 Tool Flow Support

Figure 6.2 depicts the tool flow that integrates the renewability framework with protections. Full black arrows denote the compiler and protection tool flow of code and data that includes basic code mobility (as discussed in Section 5.3) but without renewability. Dashed black arrows denote the generation of renewable code generators. This code generator generation process was added to the existing tool chain for supporting renewability. Dashed red arrows visualize the flow of code and data when renewed mobile blocks are generated, either a priori or on demand.

6.5.1 Existing Static Protections and Mobility Tool Flow

The existing tool flow supports the insertion of software protections in three phases. First our tool chain contains a number of source-to-source protection plug-ins. These take seeds, keys, and other configuration parameters as input, together with the application source code to be protected. In step ⓐ, the plug-ins produce transformed, (partially) protected application source code, as well as protection source code that implements additional protection functionality to be injected into the application. The operation of the plug-ins is based on source code annotations such as pragmas and attributes. These annotations allow one to mark the code fragments that need to be protected and to specify the protections to be deployed, their parameters and configuration. Figure 6.2 only depicts one source-to-source plug-in, but any number of them can be chained in practice [5].

Both sets of source code produced by the source-level plug-ins chain are then fed to a compiler to produce object files in step ⓑ. The compiler can optionally inject additional protections. In our prototype, this is not

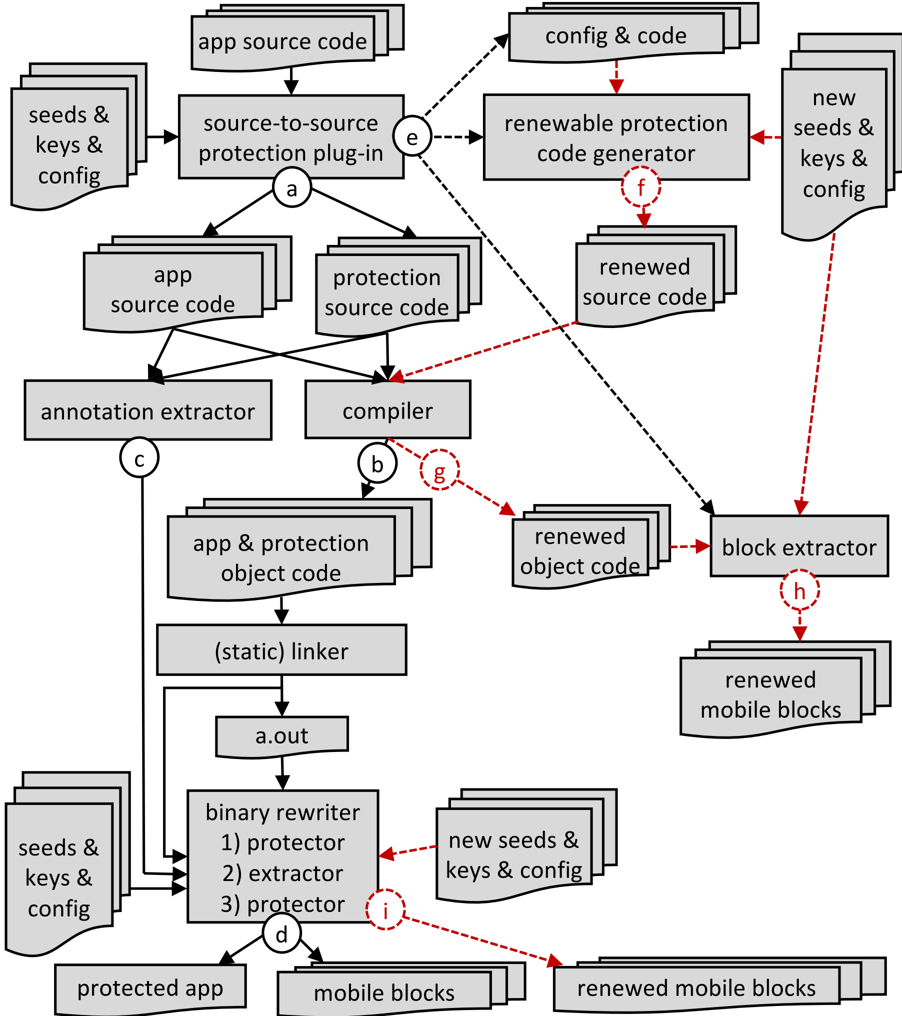


Figure 6.2: Tool flow for generation of renewable blocks

the case, as we use a standard LLVM to compile Linux and Android binaries.

From the source code files fed to the compiler, the remaining annotations and their line numbers are extracted by an annotation extractor in step ⓒ. After the object files have been linked, both the object files, the linked binary `a.out`, and the extracted annotations are fed to a binary rewriter, together with additional seeds, keys, and configuration info. The binary rewriter deploys binary-code-level protections, extracts blocks to make them mobile, and applies further protections, both on the extracted code and on the remaining, static code. In step ⓓ the rewriter produces the fully protected application as well as an initial set of mobile blocks as specified by the code mobility annotations extracted from the source code. The binary rewriter maps source code annotations onto binary code fragments by means of the extracted source line numbers and the line number information present in the object files.

6.5.2 Renewable Code Generator Generation

The existing static tool flow is extended in several ways to enable renewability for both protection code and original application code. First, the spec of the source code annotations is extended to support renewability. The tool chain documentation provides a full spec of the annotations [5].

Secondly, source-to-source plug-ins are extended to produce not only the initial code version, but also the necessary code and data for generating additional code versions later on. See ⓔ in Figure 6.2. Three components are made available per form of renewable protection. First, a *renewable protection code generator* is needed, and its *code and configuration* inputs need to be stored persistently. The generator is a version of the plug-in that can be invoked separately, with new seeds, keys, and other configuration parameters to generate different code versions. Its code and configuration input contains a partial copy of the original source code and annotation input of the plug-in. This generator can be application-specific, in which case it is produced or at least customized on the fly by the plug-in during the source-to-source protection, but it can also be a pre-installed tool.

To inject the renewed source code generated by the generator and compiled by the existing compiler into actual mobile blocks, a *block extractor* is needed. This can also be application-specific or pre-installed. It knows, for the form of protection supported by the plug-in, how to extract binary code fragments and data sections from object files, which

can trivially be done with standard GNU binutils tools, and how to create new mobile block versions out of them to be stored in the Diversified Block database.

6.5.3 Renewable Code Generation

With the presented tool flow, renewed versions of code and data blocks can be generated. For source-level protections, the generators are invoked on their input codes and configurations, albeit of course with new, different seeds, keys, and parameters. The result of this step (f) is renewed source code, either of the original application or of some protection. This renewed source code is then compiled to produce renewed object files in step (g), after which the block extractor extracts and assembles renewed mobile blocks in step (h).

For binary-level protections, our prototype of the renewal process re-runs the binary rewriter on its inputs with new seeds, keys, and configuration parameters. The binary rewriter then produces renewed mobile blocks in step (i). This is not very efficient, as each invocation of the rewriter re-executes all the binary-level processing passes, including passes on code fragments that do not become mobile. With some engineering, this can definitely be made more efficient.

6.5.4 Discussion

Neither the framework architecture nor the tool flow are limited to application executables. As is, they can also be deployed to protect dynamically linked libraries. In Figure 6.2, both a.out and the protected app in that flow can in fact be libraries such as libmylib.a.

In our proof-of-concept implementation resulting from the ASPIRE project, all the necessary client-side components (Communication Logic, Downloader, Binder, ...) are linked statically into either an executable or into a dynamically linked library, and non-exported symbols are stripped. This means that, e.g., the Binder cannot be identified by means of symbol information, and also that its code is mingled and protected with the original application or library code. This design choice does imply that when multiple dynamically linked libraries protected with renewable protections are loaded into the same application process, those components will be loaded and executed multiple times, possibly even in parallel. To avoid this overhead, one could opt to put the client-side

components in a separate dynamically linked library, of which only one copy then needs to be loaded into a process.

That would lower the level of protection, however, as all the interfaces to those components are then exposed in the libraries' exported and imported symbols. Furthermore, in that case a MATE attacker would only have to attack one version of those components to defeat all renewability in the process. When there is one copy per library, all of which can be protected with different, independent anti-tampering and anti-reverse-engineering protections, such as different forms of obfuscations, remote attestation, and renewability, an attacker will have to invest much more work.

Secondly, putting the components in an external library would imply that the single version of each component that is then loaded into the running process has to perform the renewability bookkeeping of multiple libraries that were possibly compiled and protected completely independently from each other. This would make those components much more complex, and it would significantly impact important aspects of the software development life cycle. For example, it would imply that only libraries protected with compatible forms of the renewability support can be loaded together into a process. This would make it practically infeasible to load protected libraries from independent vendors into the same process, which would result in a DLL Hell as existed on Windows in the past. In the current design, by contrast, every loaded library and the renewability components in it are oblivious to the fact that other protected libraries with renewability components are running in the same process. They can even connect to different servers. This is obviously useful: It is not unimaginable that vendors of different libraries (e.g., libraries that implement vendor-specific DRM plug-ins for Android's media and DRM frameworks) only trust their own servers.

Also on the server, each of the running libraries are treated in isolation. Even if multiple libraries running in some client process connect to the same server, that server does not know that its incoming requests are originating from the same running process. This greatly eases the design and development of the server functionality.

Of course, this design choice limits the flexibility of the server decision processes. Currently, there is no global coordination between the renewability services serving the multiple libraries that may be running in the same process, and that hence may be undergoing the same attack. As future work, we plan to investigate whether such coordination can be supported effectively and efficiently.

6.6 Mitigations Against Concrete Attacks

The renewability framework supports a range of renewable software protections that mitigate MATE attack steps.

6.6.1 Syntactically Diversified Mobile Code

Dynamic analysis can be done manually, e.g., with a single-step debugger, or it can be automated, e.g., by collecting trace information with an emulator. It can also be semi-automated, e.g., by writing small debugger scripts that steer the program execution up to the specific point of interest by means of breakpoints and watches, at which point manual single stepping can start to collect additional information. Such scripts are often developed iteratively: Each time more information is obtained, the scripts are adapted to zoom in on the next piece of useful information on the attackers' path. All of these approaches commonly involve multiple runs of the same program. This also happens in, e.g., delta-debugging-like attacks, in which the difference in program execution behavior on different inputs is analyzed [3], and it obviously also happens in fuzzing attacks [100]. Such attacks require repeatability, and become harder if the code fragments that are revisited differ from one run to the other.

This can be achieved by syntactically diversifying the code in renewed mobile blocks. Rather than creating one version of a mobile block, multiple semantically equivalent but syntactically different versions can be created and delivered.

Our prototype tool flow creates versions by stochastically applying obfuscations (opaque predicates, branch functions, and control flow flattening) and code layout randomization on the extracted code fragments. By initializing a pseudo-random number generator with varying seeds, versions can be generated that feature varying CFGs and code layouts [29]. This makes it significantly harder, e.g., for attackers to automate the setting of breakpoints in their scripts. It also makes it harder to compare multiple traces in collusion attacks.

The applied obfuscations have previously demonstrated their effectiveness in the context of collusion attacks that rely on program diffing [29], where they prevent diffing tools to automate the identification of the corresponding code fragments in two syntactically different versions of the same software. We therefore conjecture that it will be non-trivial for an attacker to automatically overcome the protection provided by syntactically diversified mobile code.

6.6.2 Semantically Diversified Mobile Code

Syntactical diversification does not hamper all attack tools. For example, pointer chaining tools (e.g., Cheat Engine - `https://www.cheatengine.org/`) can still find relevant data in randomized memory layouts during repeated executions. In a first run of a program, the attacker then identifies the relevant data in the process memory space manually. The tool then collects the pointer chains to the identified data. These chains are lists of offsets. For example, the transparency value of a wall in a shooter game might be located at the end of the chain `*(*(*(frame_pointer_main+24)+4)+8)`, which does not depend on the code syntax or layout, or on the data layout as affected by address space layout randomization. Cheaters might want to make the walls transparent to see their adversaries through them. For such chains to become invalid for repeated executions, the order of fields in C/C++ structs and classes needs to be diversified, the location where data is stored in stack frames needs to be diversified, the order in which parameters are passed to functions needs to be diversified, etc.

We hence need diversification that also changes the semantics of individual fragments, i.e., the relation between the process state before and after their execution. For example, when the fields in a C struct are reordered, fragments writing to the fields will write to different offsets in allocated memory blocks, thus implementing different semantics.

Deploying such diversifications is more difficult, however, because they have a more global impact on the generated code. If the order of fields in a struct is altered, all code in the binary that accesses the struct will change as well, in a consistent manner: The same change in offset will occur all over the program. Likewise, if the signature of a procedure is altered, e.g., by reordering its parameters, the procedure's code body will change, but so will the code of all its callers. When aggressive compiler optimizations are used, those initial changes can result in ripple effects throughout the binary code of all directly or indirectly affected functions. In general, almost all data and data flow obfuscations or diversification techniques [88] have more global effects on the generated code. To support such forms of diversification, a client that is served multiple diversified code fragments by a renewability server can only execute correctly if all of the fragments served during a single run implement and assume consistent semantics.

Our approach supports this, because the server can partition the diversified mobile blocks into consistent groups: The Renewability Man-

ager can be informed which versions of the mobile blocks in the database feature consistent semantics, which do not, and which ones are independent. Simple server-side bookkeeping can then ensure that whenever some block is requested, it only delivers blocks that are independent of or consistent with previously delivered blocks.

Of course, the use of this form of semantic fragment-level diversity restricts the freedom of the renewability policies to replace fragments within a single execution of a program: Once data values or the layout of data in the client program's address space have been produced by a certain first code version, all code executed later during that execution has to be consistent with that first code version.

Still, the use of attack tools and reuse of attack scripts over multiple runs of an application can be significantly hampered by this form of protection. In particular, it will decrease the effectiveness of pointer chaining tools.

To support semantic renewability, we extended the basic tool flow of Figure 6.2 somewhat. For a prototype implementation that changes the layout and order of fields in structs and the parameter order of functions, we rely on a source-to-source protection plug-in to generate the diversified code. To enable the identification of all binary code fragments that undergo relevant changes as a direct result of the source-level diversification or as a result of ripple effects through compiler optimizations, multiple approaches can be envisioned.

In our approach, we do not want to restrict or alter the used compiler. Instead, in line with common industrial software development life cycle requirements, we want to keep treating the used compiler as a black box. Then two options remain. The first, more conservative option, is to track references to diversified function signatures and structs in the rewritten source code, to mark any function that directly or indirectly touches (directly or indirectly) upon such references as dependent on the deployed diversification, and to enforce separate compilation of each function in the compiler such that compiler optimization ripple effects are bound to individual functions. This strategy will conservatively overapproximate all functions or code regions that might be affected by the deployed diversification, which allows us to make all of those mobile and renewable.

A more accurate identification of the altered binary code fragments, i.e., the ones that need to be made mobile and renewable as a result of source-level diversification and potential ripple effects, can be achieved through binary diffing. To enable this diffing in our tool flow, we generate

all diversified versions upfront. We also compile all of them upfront. We then run a binary differ that compares the compiled binaries, and identifies the precise differences between the compiled versions. The binary code regions embodying those differences are then marked to be made mobile and renewable in the binary rewriter. Only after all versions are compared and all necessary regions are identified do we run the binary rewriter to extract the necessary blocks from all program versions. The remaining rewritten binary in which these blocks are removed, is then identical across all versions, as it consists of the binary code fragments that did not differ at all across the different versions, whereas the mobile blocks contain all the differing code.

This black-box approach offers the advantage of not needing any change to the used third-party compiler and linker, or to the internal operation of the binary rewriter. The developer of the source-to-source protection plug-in that implements this semantic diversity hence does not need to invest any effort in learning all the ripple effects that those three complex tools might induce as a consequence of his source code transformations. The diffing tool automatically exposes all code impacted by the ripple effects. We implemented our own Clang-based source-to-source rewriter, but our approach easily allows for other (already existing) source-to-source rewriters.

It is important to note that the semantic diversification does not need to be limited to individual code fragments. If appropriate, it can easily be extended to externally visible changes to the semantics of the whole program as well. For example, in some cases it might be useful to renew the semantics of code fragments that prepare a payload to be sent to the original application server (see Figure 6.1) or that consume a payload received from the application server. Formally, this changes the semantics of the whole client program, but if this is coordinated with the semantics implemented on the application server, this can be perfectly fine, and happen transparently to the end user of the software.

6.6.3 Diversified Static-To-Procedural Conversion

Static data such as strings can serve as hooks in many attacks. To protect these against static inspection, static-to-procedural conversion [88] replaces the static data by invocations to injected procedures that compute the data on the fly. If dynamic attacks are then also made harder, e.g., by combining this protection with anti-debugging (discussed in Chapter 4), strong protection can be obtained. With renewability, the level of protection can be increased even further: If the code that computes the data changes between every run of a program, the attacker will have to adapt and re-execute his attack script to extract the data he is after whenever a new version is downloaded.

This form of renewability is readily supported: It suffices to let the source-to-source protector generate randomized procedures to replace static data, and to annotate these to have them extracted by the binary rewriter.

6.6.4 Dynamic and Time-Limited WBC

WBC (White-Box Cryptography) is a technique for protecting the confidentiality of cryptographic keys in software [20, 113]. The literature mostly focuses on fixed-key implementations, where the key is hard-coded in the software. Rather than including a key as a constant input to a standard implementation of a cryptographic primitive, which is trivial to attack in a MATE scenario, a custom version of the primitive is included in the software, which hard-codes the key in a way that it cannot be extracted (easily), e.g., by encoding it in large randomized tables or code structures.

Fixed-key implementations are acceptable for some use cases such as hard-coding global bootstrap keys. However, for many industrial use cases keys need to be updatable. For example, for personalizing software with application-unique keys or for installing service-dependent keys, cryptographic implementations can ideally be instantiated with keys at run time [114]. While there is almost no literature on this, several companies are selling such *dynamic-key white-box* implementations; there is no publicly available information on they are built. One possible approach would be to build special-purpose white-box implementations which receive a protected version of the key as input. The protection of the keys then needs to be integrated in the application design, and additional routines such as preprocessing the protected key and key schedule algorithms need to be integrated: This introduces a lot of addi-

tional complexity and can have a considerable impact on performance and code size. Another approach is to update existing fixed-key implementations at run time. In the most common white-box implementations such as that of Chow et al. [20], key material is embedded in look-up tables. It suffices to update these tables in order to change the key, so the technique of mobile data blocks can be applied to achieve dynamic white-box implementations. Other white-box techniques do not solely depend on look-up tables [8, 11], but also encode the key in complex code structures. Updating the key then implies updating the code. This is also supported out-of-the-box with our renewability framework. In summary, our framework offers all that is required to evolve from static key WBC to dynamic key WBC.

Still, designing secure WBC implementations, whether static or dynamic, remains a challenge. All currently proposed designs have been broken, and recent proposals that are submitted to the ECRYPT White-Box Cryptography Competition (the WhibOx Contest) [37] are challenged in a matter of hours or days. Therefore, rather than focusing on designing implementations that give long-term security guarantees (and will probably be very slow and large) an alternative approach is to focus on more efficient but less secure implementations that are renewed at high frequencies. We denote these as *time-limited white-box* implementations. Such WBC implementations can protect short-lived session keys or temporary access tokens with acceptable performance. With those implementations, it are then not the keys that need to be rotated frequently, but the WBC implementations that embed the keys. This rotation is readily supported by our renewability framework.

Combining this form of renewability with the already discussed forms of diversification can then help to achieve longer-term protection as well, namely by ensuring that the rotated implementations differ in more respects than simply embedding different keys, thus hampering attackers in reusing simple attack scripts.

6.6.5 Diversified Instruction Set Randomization

A popular form of obfuscation is to translate an application or part thereof is to some virtual, randomized bytecode ISA. At run time, the bytecode is emulated. Popular tools that implement this form of emulation-based obfuscation are Code Virtualizer [91], EXECryptor [99], Themida [90], and VMProtect [106]. Unlike native code formats—which are well-documented by processor manufacturers—the randomized ISA

is not documented. It is also diversified for each protected program to reduce the learnability for attackers.

Custom bytecode can be made mobile when it is read-only data where the set of code locations that refer to it is clear and limited. In the ASPIRE tool chain, security-sensitive chunks of native code are translated into bytecode chunks [14, 116]. The original chunks are replaced by stubs that invoke the emulator, passing it a pointer to the data representing the bytecode. This scheme fits our mobile data block support perfectly: The stub becomes mobile code, and the bytecode—to which only the stub produces a pointer—becomes a mobile data block attached to that mobile code.

Bytecode renewability can then be achieved by combining diversified mobile bytecode with semantic diversification of the emulator. Both the semantics and the syntax of the bytecode can then vary over time; for each execution a corresponding interpreter and bytecodes are delivered.

6.6.6 Evolving Protections

The proposed tool flow and architecture pose no limits on the sizes of the mobile blocks. In particular, renewed blocks don't need to have the same sizes. This helps in supporting gradually evolving renewability, e.g., where over time more complex forms of protections are delivered to client applications as those protections become available in response to detected attacks. For example, when more advanced attacks on WBC crypto schemes become available over time that reduce the search space for brute-force attacks, or when faster brute-force methods become available, more complex versions of the white-box algorithms can be delivered to the client applications to catch up with the attacker's capabilities. In many WBC schemes, this can be achieved with bigger tables that embed the secret keys.

To support such evolving protections with our framework, the only aspect that needs to remain constant from one mobile block to another is the binary-level interface of each renewed code block, i.e., the way data is passed to and from the mobile blocks to static code, and in between mobile blocks: the registers used, the stack frame layout, ...

For source-level forms of diversification and renewability, this requirement of constant binary-level interfaces can in practice only be achieved when the mobile code blocks correspond to units of which the compiler cannot alter the binary-level interface at will. This is the case for functions or methods, because compilers are bound to calling

conventions. Functions and methods are often also the "units" in which developers implement functionality, be it protection, library, or application functionality. So in practice, the limitation to renew only whole functions does not impose overly strict restrictions on the ability to let the deployed protection components vary over time.

Besides the potential to respond to advances in the attacker's toolbox, this ability to vary the deployed protection offers two major advantages. First, it can help in reducing the time to market. Selecting the optimal combination of software protections is a cumbersome, difficult, time-consuming task. The proposed renewability framework—and the capability to vary protections over time—allows vendors to release weaker protected versions early, and to upgrade the protection seamlessly (without the user being disturbed) after the initial release. Second, the ability to vary the deployed protections over time can be used to find a better balance between their strength and overhead. A good example is code integrity verification by means of remote attestation based on code guards. Code guards are basically hashing functions that compute hashes over the code being executed. With remote attestation, a server requests such a code guard to be executed on some code region, and checks whether the received hash value is the expected one. If not, this is a signal that the code has been tampered with. Different remote attestation and code guard designs come with different degrees of overhead. To keep the overhead acceptable, all schemes leave some freedom to the attacker to tamper and remain undetected. When the deployed scheme varies over time, however, as supported by our approach, the attacker has to take into account all possible schemes to remain undetected for a longer period of time. As already discussed before, attacks often involve multiple executions of a program, so this period typically spans multiple executions. During any (tampered) run then, the attacker has to be cautious and assume no freedom, as if all anti-tampering schemes were being deployed together. At any point in time, however, only one or a couple of schemes are actually deployed. A regular user thus only experiences the overhead of a limited number of them. In the ASPIRE project, we experimented with renewing code guard implementations that, e.g., vary the pseudo-random walk over the code fragments they hash. By making it unpredictable for an attacker which instructions will be visited, it suffices to hash only a limited number of instructions during any invocation of a guard.

6.7 Experimental Evaluation

6.7.1 Target Platform of Prototype Implementation

Our prototype targets ARMv7 client platforms. Our client hardware consists of several developer boards, on which we ran Linux 3.15 and Android 4.3+4.4. For Linux, we used a Panda Board featuring a single-core Texas Instruments OMAP4 processor, an Arndale Board featuring a double-core Samsung Exynos processor, and a Boundary Devices Nitrogen6X/SABRE Lite Board featuring a 1GHz quad-core ARM Cortex A9 with 1 GByte of DRAM. The latter was also used for running all Android benchmark versions, and for running the measurement experiments reported below. On the server side we set up a VirtualBox VM (Virtual Machine) running a 64-bit Debian Linux, 2 GBytes of RAM and a Gbit NIC adapter. This VM ran on an Intel Xeon E3-1270 CPU 3.50GHz with 16 GBytes of RAM. We used GCC 4.8.1, LLVM 3.4, and GNU binutils 2.23 for the client, for which we compiled code with `-Os -march=armv7-a -marm -mfloat-abi=softfp -mfpu=neon -msoft-float`. On the server we used GCC 4.8.1 and binutils 2.23 to build our components. The Mobility Server and Renewability Manager were compiled with `-O3` and `-Os -fpic` respectively.

6.7.2 Validation on Use Cases

The robustness and applicability of our framework were tested by deploying various forms of renewability on two industrial use cases that were developed independently by two market leader companies using different development approaches, software architectures, and build systems. Each use case consists of a shared library of sufficient complexity to represent real software products and with embedded, security-sensitive assets representative of the assets in the companies' real products. We chose the code and data fragments to make mobile and renewable together with the application architects and developers, and with security architects from the companies.

The first use case consists of two plug-ins, written in C and C++ at Nagravision S.A., for the Android media framework and the Android DRM framework. These plug-ins, in the form of dynamically linked libraries, are necessary to access encrypted movies. A video app programmed in Java is used as a GUI to watch the videos. This app communicates with the mediaserver and DRM frameworks of Android, informing the frameworks which vendor's plug-ins they require. On demand, these

use case	developer	SLoC	3rd-party libraries	assets	forms of mobility and renewability
DRM library	Nagravision	306.2k	OpenSSL	crypto keys	mobile code, diversified & dynamic (time-limited) WBC (Section 6.6.4)
software license manager library	SafeNet Germany	55.4k	tomcrypt, tommath	keys, code IP	syntactically diversified mobile code (Section 6.6.1), diversified instruction set randomization (Section 6.6.5)
bzip2 app	Julian Seward (open source)	5.8k	-	-	syntactically diversified mobile code (Section 6.6.1) semantically diversified mobile code (Section 6.6.2)
WBC crypto app	Dušan Klinec (open source)	6.3k	-	crypto keys	diversified WBC (Section 6.6.4)

Table 6.1: Feature matrix of the renewability use cases

frameworks then load the library plug-ins. In our research, we observed several features that make this use case a perfect stress test. The multi-threaded mediaserver launches and kills threads all the time. The plug-in libraries are loaded and unloaded frequently, sometimes the unloading being initiated even before the initialization of the library is finished. As soon as the process crashes, a new instance is launched. Sometimes this allows the Java video player to continue functioning undisrupted, sometimes it does not. These forms of behavior stress all client and server components.

The second use case is a software license manager that stores credentials, and controls access to licensed content and functionality, e.g., through time-limited and key-enabled licenses. This manager is programmed in C at SafeNet Germany GmbH. It is a dynamically linked library that includes the JNI interface, and is embedded in an Android app. This native library thus functions as a license manager for a Java application. In this case, the Java application is relatively simple: It is a riddle game of which the solutions are protected by the license manager. To test our renewability support, this use case is also interesting. In particular, the library is loaded into Android's Dalvik execution environment, which features multiple threads (such as for the JIT compiler, garbage collector, ...), and over which we have absolutely no control [9]. A command-line version of the riddle game, programmed in C, is also available. It uses the same library (except the JNI wrapper). On top of providing an easier target to debug on our Android developer boards, this command-line version can also be compiled for Linux. This way, we could also test our implementation on Linux.

Table 6.1 lists a number of features of the two use cases as an indication of their representativeness of real-world software. The number of source code lines includes all the mentioned third-party libraries that are compiled and statically linked into the shared libraries to be protected. Whereas the linked-in libraries do not contain any assets, they operate on assets such as keys, and their control flow needs to be protected against reverse engineering as well.

Even though no additional protections are listed in the table, we did actually combine many additional non-renewed protections with the listed forms of renewability on the industrial use cases. This includes anti-debugging (discussed in Chapter 4), remote attestation [105], and code and data obfuscation techniques [88].

The third row of Table 6.1 lists bzip2, the popular compression tool. While this open source program does not contain any security-sensitive

assets, we did deploy it to evaluate the correctness of our tool support for both syntactically and semantically diversified mobile code on a real program. We evaluated two semantic source-to-source diversifications: struct field reordering and function parameter reordering. The correctness of the semantic code diversification transformations was evaluated by compiling and testing the diversified code as it was diversified with the source-to-source plug-in. The correctness of the whole semantic code diversification setup was evaluated by using the deploying the full extended tool flow, including the binary diffing and mobile block extraction during binary rewriting, on multiple diversified versions. For all of them, the exact same static binary with mobile blocks extracted was obtained, and that binary was tested to execute correctly with any compatible version of renewable blocks delivered to it.

Next, we investigated how the degree of semantic diversification (of Section 6.6.2) influences the generated binaries and mobile blocks. In our flow, a set of blocks that is mutually compatible originates from the same diversified instance of the program. Any function that is diversified in any specific instance needs to be made mobile in all instances in order for them to be compatible with the same binary. Thus, increasing the number of diversified program instances—from which the renewable sets of blocks originate—will have an effect on the number of blocks that need to be made mobile, and on the size of the remaining static binary. To gain some insights into these effects, we experimented with bzip2 and function parameter reordering. Our tool flow has a configuration parameter to specify the number of different versions that need to be generated by diversifying the code of selected software components. We varied this parameter value from 2 to 100. For each evaluated value, we did not select any specific subset of functions for diversification, but instead allowed the tools to randomly select any subset of functions in the whole program to reorder their parameters. For each of the selected parameter values, we ran a total of 20 differently random-seeded runs of our framework, and averaged the measurements. We also measured the size of the .text section of the undiversified bzip2 binary both with and without the extra support code that is linked into the binary to support the code mobility functionality. This 'base' .text section consists of 94.9KB without support code, and 1116.6KB with; it is thus clear that for this specific use case the mobility support code exceeds the original application code by an order of magnitude.

Table 6.2 shows some results. For every number of compatible programs, we measured the averages of: the number of functions found to

versions	mobile functions	mobile to base .text	base .text remaining
2	3	8.0%	93.6%
5	8	24.4%	79.8%
10	15	37.5%	68.8%
20	22	53.9%	54.7%
50	31	77.7%	33.1%
100	32	79.5%	31.4%

Table 6.2: Effects of increasing the number of diversified versions for function parameter reordering

differ and thus made mobile, the total size of mobile blocks proportionate to the base .text section, and the percentage of base .text still present in the binary. It can be seen that the portion of the binary being made mobile increases with the requested number of diversified versions, but that there is a limit to this increase. There might be code that will never be impacted by the specific diversifications used, and thus need never be made mobile (as a simple example, leaf functions without parameters can never need to be made mobile with this specific diversification transformation). Next to that, the transformations used for code mobility both increase the size of the mobile code, and the size of any code still left in the binary invoking the mobile code. Note that the two fractions add up to more than 100% because making fragments mobile involves the injection of stubs and other small code snippets in the static binary, and because the code in mobile blocks is enlarged as it is transformed to make it offset-independent as discussed in Section 6.2.

Finally, we tested diversified WBC on a small stand-alone WBC crypto app. While we did so mainly to perform overhead measurements—on which we report later—they also contribute to the validation of the prototype implementation and hence the practicality of the proposed approach.

In summary, the forms of renewability listed in Table 6.1 have been validated on four use cases. Combined, our evaluation hence successfully covers four applications from Section 6.6. Most importantly, it covers both mobile and renewed code, and mobile and renewed data—thus covering all client functionality—as well as most (and definitely all core) server functionality. Finally, the evaluation successfully covers Android and Linux platforms, and application executables as well as dynamically linked libraries.

Almost all prototypes of framework components we discussed and evaluated are available on-line in the ASPIRE project code repository at `https://github.com/aspire-fp7`; the only exception are the proprietary generators of white-box cryptographic primitives.

6.7.3 Performance Overhead

In our previous work, we already analyzed the overhead of basic code mobility when it is deployed over various wired and wireless networks with different throughputs and latencies (as discussed in Section 5.4). The difference between basic code mobility and renewability is the flushing and re-downloading of code after the initial download. The impact thereof on performance obviously depends on the frequency with which code needs to be flushed, as well as on the frequency with which it needs to be downloaded. The flushing frequency is determined by the enforced renewability policy. This hence varies from one usage scenario to another, and even from one asset to another. The re-download frequency depends on the flushing frequency, but also on the frequency with which the mobile code and data is executed and accessed. As an extreme example, a code fragment that is only executed when a new movie is launched in a media player, will need to be downloaded at most once per movie, however fast it is flushed after that execution. By contrast, a code fragment that is executed once or more per frame in the movie will need to be reloaded at essentially the flushing frequency.

The performance overhead of the proposed renewability protection will hence vary wildly from one scenario to another. We therefore aim for providing the reader a feeling for the range of overhead to expect, rather than for trying to argue that the overhead is low enough. What is acceptable and what is not, depends on the usage scenario at hand.

We did not measure the timing of the interactive industrial use cases. We can confirm, however, that the overhead of the renewability did not significantly impact the overall user experience of those apps. In the case of the DRM library, downloading mobile code produces a slight additional delay when a movie is started, but this delay is negligible compared to the delay caused by having to download enough frames to fill the video buffer. The video playback frame rate was not impacted by the renewable protections. The renewable functionality of the license manager is downloaded when the software is launched, and whenever functionality with custom licenses is accessed for the first time. On those

occasions, the downloading of code introduces a (barely noticeable) delay that is deemed acceptable.

Our first quantitative performance analysis was carried out on the CPU-intensive bzip2 program (`www.bzip2.org`). The experiment consisted of measuring different properties of multiple runs of bzip2 over the controlled, standard input consisting of the SPEC2006 training data (`www.spec.org`). Experiments were carried out on three program versions, in which different sets of functions were made renewable. For the first two versions, we collected profile information with the GNU gprof tool [53], and selected hot functions of which the total execution time approximated respectively 20% and 50% of the total execution time of the program. The second set is not a superset of the first one, but there is some partial overlap. In the third version, all functions in the bzip2 program are made renewable. This corresponds to 100% of the total program execution time. It is hence clear that this experiment is not meant to measure realistic overheads. Instead, the experiment serves the purpose of a sensitivity analysis, demonstrating that the performance overhead can be impacted by tuning the protection deployment, and that there is a need to do so, because not doing so will often result in unacceptable amounts of overhead.

With each version, we first set up a baseline by collecting the execution time of a non-protected, vanilla application. For each of the three renewability percentages, we then ran the program for different renewability flushing time-outs of 1000, 2000, 3000, and 5000ms. For each mobile block, 600 different versions were generated a priori, using syntactic code diversification techniques [29]. On each download request, the Renewability Server picks one of them randomly.

For each run we sampled the wall-clock execution time, the number of transferred blocks, their total size in bytes, and the CPU time consumed by the Renewability Manager on the server side. Each experiment was repeated 20 times to collect data, in the remainder of this section, we discuss and present averages over those 20 runs.

Table 6.3 reports the average wall-clock times and the overhead in that regard, as well as the network overhead in terms of numbers of downloaded blocks and the network throughput. Table 6.4 reports the CPU time consumption on the server. For reference and comparison, Table 6.5 presents the overhead when the different amounts of code in bzip2 are made mobile, but never flushed and renewed, i.e., when they are downloaded only once. The tables confirm that the overhead is

mobility	refresh time (s)	execution time (s)			transferred blocks		
		Mean	StDev	overhead	Mean	per sec.	kb/s
0%	-	279	0.3	-	-	-	-
20%	1	324	1.6	16%	753	2.32	18.38
	2	321	1.9	15%	401	1.25	9.94
	3	319	1.0	14%	276	0.86	6.93
	5	317	1.0	14%	171	0.54	4.34
50%	1	487	1.9	74%	3,885	7.97	12.77
	2	475	3.7	70%	1,953	4.11	6.83
	3	459	1.1	64%	1,267	2.76	4.63
	5	456	2.9	63%	793	1.74	2.90
100%	1	647	4.9	132%	9,818	15.17	30.47
	2	591	10.4	112%	5,236	8.86	18.99
	3	565	3.3	102%	3,498	6.19	13.29
	5	552	4.1	98%	2,127	3.85	8.23

Table 6.3: Client wall-clock execution times and network throughput of renewability on bzip2

mobility	renewability refresh time (s)			
	1	**2**	**3**	**5**
20%	405	363	334	300
50%	923	621	544	439
100%	1,006	860	669	565

Table 6.4: Server CPU consumption for bzip2

		mobility		
		20%	**50%**	**100%**
Client exec time (s)	**Mean**	282	299	313
	StDev	211	135	136
	overhead	1.1%	7.1%	12.0%
transferred blocks		4	22	55
blocks/s		0.01	0.07	0.18
network throughput (kb/s)		0.08	0.06	0.18

Table 6.5: Baseline overhead of code mobility on bzip2

directly related to both the renewal refresh rate and the hotness of the code fragments being renewed.

Comparing the server overhead to the client execution times, we observe that for this program and hardware, the server CPU load varies between 0.1% and 0.2% of the client load. Scalability on the server is hence another factor to be considered when deciding on the use of renewability, on the fragments to be made renewable, and on the renewal

refresh time (s)	user-space CPU time (s)			wall-clock exec. time (s)		
	Mean	StDev	overhead	Mean	StDev	overhead
baseline	156.1	0.4	-	156.7	0.4	-
1	161.0	0.5	3.1%	179.0	0.6	14.2%
2	158.8	0.4	1.7%	165.6	0.4	5.6%
3	157.4	0.6	0.8%	162.5	0.6	3.7%
4	156.9	0.5	0.5%	160.6	0.5	2.5%
5	156.3	0.6	0.1%	159.8	0.5	2.0%

Table 6.6: Client CPU consumption and wall-clock execution times of renewable WBC

refresh time (s)	transferred blocks		transferred MBs	
	Mean	**StDev**	**Mean**	**StDev**
1	179.8	0.7	205.1	0.8
2	83.4	0.5	95.1	0.8
3	54.9	0.4	62.6	0.4
4	40.9	0.3	46.7	0.4
5	32.5	0.5	37.0	0.6

Table 6.7: Network throughput of renewable WBC

policy enforced by the server. The same obviously holds for scalability of the network capacity.

A similar experiment with a C++ WBC crypto application was based on Dušan Klinec's implementation of the Chow WBC scheme without external encodings [20], available at `https://github.com/ph4r05/Whitebox-crypto-AES`. The decryption primitive and its embedded key are implemented by means of large tables that total 1.14MB. Renewing this routine and its tables to renew the decryption key hence involves the downloading of a mobile block of about 1.14MB. This is significantly larger than the code blocks that were downloaded in the bzip2 experiments.

Table 6.6 reports client user-land CPU consumption times and client wall-clock execution times of the baseline version without renewability, and of the renewable version at different refresh rates. The differences between the overheads in both measurements is significant. This is of course due to the fact that the client side spends a significant amount of time waiting for the large mobile blocks to arrive. However, during that wait, no CPU resources are consumed. Still, even for the version that only refreshes the routine and its embedded key every 5 seconds, the user-land CPU time increases significantly. The reason is that the Downloader and Binder components take up some computation time,

and that the code transformations that are necessary to implement code mobility and renewability also have a small, but significant effect on performance.

Table 6.7 shows how the network throughput scales with the refresh rates. The number of transferred blocks, which equals the number of refreshes (plus 1) scales superlinearly with the refresh frequency because the execution time of the benchmark increases with higher refresh frequencies. For this form of renewability, which inherently involves large mobile blocks, the measurements confirm that network scalability is an important issue to consider.

6.8 Related Work

Our framework combines and extends concepts from network-based protections, and software diversity. Network-based software protection techniques leverage software updates and trusted network services. The updates may be implemented for the functional part of the program, and for the protection techniques used to protect it [29]. Both Collberg et al. [27] and Falcarin et al. [40] proposed the continuous replacement of binary code, with the former making use of CIL (Common Intermediate Language) to generate the diversified code. Collberg et al. supports both syntactic and semantic diversity, using what they call Protocol-Preserving and Non-Protocol-Preserving Transformation Primitives, respectively. Contrary to the work of Collberg et al. [27], our framework works by directly replacing binary code, giving it more freedom in terms of granularity and composability with other techniques. Contrary to both, our framework not only makes it possible to renew application code. It can renew entire protection techniques, in an automated, specialized, manner.

Previous Java work implemented dynamic replacement of remote attestation protection code downloaded by a trusted server, using extended Java Virtual Machines [96]. Other techniques such as remote attestation extend code guards with a network server. The Pioneer [70] system relied on a verification function running on the client as an OS primitive, and an attestation server. Garay et al. [48] presented an approach where a trusted challenger sends a challenge to the potentially corrupted responder. The challenge is an executable program that can execute any function on the responder, which must compute the challenge fast enough to prove its integrity.

In literature [32, 46, 108], software diversity relied on random generation of diversified copies, starting from the same source code, extending the idea of compiler-guided code variance [45]. A survey [72] compares the different approaches for software diversity in terms of performance and security, and recently software diversity has become practical due to cloud computing enabling the computational power to perform massive diversification [72]. Past software diversity approaches have been based on some form of obfuscation [22], load-time binary transformation [68], virtualization obfuscation based on customized virtual machines [58], or OS randomization [117]. Other approaches rely on binary transformation based on a random seed [104], or multi-compilers and cloud computing [46] to create a unique diverse binary version of every program, and they apply such diversification for mobile apps [108]. The XIFER framework [32] randomly diversifies Android apps at load time by means of a binary rewriter. Both spatial and temporal software diversity has been proposed as a solution to a wide range of problems: code randomization has been used to defend against code-reuse attacks [98], return-oriented programming attacks [57], and code injection attacks [112]. More fine-grained forms of diversification have been proposed to raise the bar even further [19, 51], including for code dynamically generated with JIT compilers [59]. Dynamic temporal diversity has been proposed to mitigate timing side channel attacks [47]. Diversification can prevent collusion attacks to identify vulnerabilities [29].

Compared to the discussed work, our renewability framework provides a foundation to combine, compose, extend, and hence fortify several existing defenses. The tool flow supports combinations and compositions, meaning that multiple protections can be deployed together on the same program or even on the same code fragment. This follows in part from its conception as part of the ASPIRE Compiler Tool Chain, the software protection tool chain developed in the ASPIRE project as automated support for a wide range of software protections. Our framework is fully compliant with the ASPIRE software protection reference architecture [14, 116]. The whole ASPIRE Compiler Tool Chain is available at `https://github.com/aspire-fp7/framework`. As demonstrated, it is applicable to native code, and is hence not limited to code distributed in higher-level, more symbolic (and hence easier to attack) formats such as Java bytecode or CIL. The granularity of the renewability is furthermore not limited to coarse code fragments such as whole functions. Much smaller (security-sensitive) code regions can instead be made renewable.

As already discussed in Sections 6.1 and 6.6, the framework and concrete instantiations of its capabilities can mitigate concrete attack paths. Recently, Ceccato et al. reported results of a qualitative analysis of how professional hackers as well as amateurs understand protected code while performing attack steps [16]. The resulting taxonomy of concepts used by the hackers to describe their attacks towards code understanding, and the inferred models of their activities and their reasoning, provide further insights into how the proposed renewability framework can impede certain attack paths and attack strategies. Several activities are impacted by renewability as supported by our architecture and tool flow, including but not limited to: static analysis, tracing, debugging, statistical analysis, assessing the effort, building of workarounds, undoing of protections, overcoming of protections, formulating hypotheses, and confirmation of hypotheses. The latter two play an important role in real-world attacks. They depend to a large degree on repeatability of attack activities, which is directly addresses by the forms of renewability our framework supports,

6.9 Conclusions and Future Work

We presented the ASPIRE framework, architecture and tool flow support for native code renewability. This framework supports several forms of renewability, in which renewed and diversified code and data, belonging to either the original application or to linked-in protection components, is delivered from a secure server to a client application on demand. This results in frequent changes to the software components when they are under attack, thus making dynamic attacks harder. Several applications of the renewability framework have been discussed, some of which extend existing protections, and some of which enforce existing protections. The prototype implementation was evaluated successfully on a number of use cases, including complex libraries representative for real-world, industrial use cases. Most of the prototype implementations are available online as open source.

Some future work would be the authentication and verification of renewed mobile blocks. This was deliberately not discussed, as we consider it to be out of scope for this chapter, but does pose some challenges. Executing code one just downloaded from the internet without verifying it first is of course a recipe for disaster, and is almost the definition of arbitrary code execution. Client-side verification of these renewed mobile blocks is thus a must, and the code responsible for this verification ought

to be suitably hardened. The server-side infrastructure to sign mobile blocks is of course also required, and can become a target as well.

Chapter 7

Conclusions and Future Work

*"Now, **here**, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"*

—The Red Queen, *Through the Looking-Glass*

In my research I focused on two attack models. In one model, attackers use bugs in programs to gain *arbitrary code execution* in these programs with the goal of subverting the OS, and taking actions for which they have no authorization. In a *MATE attack*, on the other hand, attackers are assumed to have complete control over the OS, and it is the program itself which needs to provide protections against attackers attempting to compromise its assets. In this chapter I will briefly restate my contributions to the state of the art in defending against both attack models, and provide some possible future research directions.

Program diversification provides a probabilistic defense against attacks aiming to gain arbitrary code execution, and obstructs MATE attackers as well. Having every user run their own program binary presents some challenges however, slowing the adoption of diversification techniques. One of these challenges is running a crash-reporting system for diversified binaries. In Chapter 3 we therefore presented ΔBreakpad, an approach where programs are diversified and extended with some Δdata, allowing the crash-reporting server to correctly interpret any minidump sent by the program. Our approach thus makes it easier to use diversification in practice, which allows for improved security.

Further improvements to our approach can be made with respect to the diversification techniques we used in our implementation. Currently these are rather simple, and it is worthwhile to investigate whether more complex techniques can be supported and which effect this will have in terms of Δdata. Specifically, it would be interesting to investigate program diversification with dynamic techniques. A dynamic diversification technique only diversifies the program at run time. Every execution of the program can then be diversified differently, and a program can even be re-diversified during its execution. The diversification happens client-side, however, where no symbol file or ΔBreakpad functionality is present. While the seed used for a dynamic diversification can be sent to the crash-reporting server just as easily as before, the question of whether and how any Δdata is generated, is harder.

Although many techniques protecting against MATE attacks already existed, there was still room for improvement. Chapter 4 and Chapter 5 present our improvements to the self-debugging and code mobility techniques, respectively. First, in our improved self-debugging technique, code fragments are migrated from the program to its self-debugger. This way, the semantics of the code in the self-debugger is not predetermined, and multiple control flow paths are possible for every invocation. This makes attacks on self-debugging programs significantly harder. Second, our improved code mobility technique allows for selectively making certain parts of the program mobile, through source code annotations. This makes code comprehension harder, but also allows for replacing parts of the program and its protections.

While improving individual protection techniques is important, making it harder for MATE attackers to use one attack vector might focus their attention on other attack vectors that become easier by comparison. Therefore, it is even more important to combine multiple techniques to ensure possible paths-of-least-resistance are hardened. Diversifying these combinations and their constituent techniques across different user instances makes it harder for attackers to scale up their attacks. Furthermore, repeatability is important to attackers, who depend on the expectation that attacks keep working on repeated executions of the same software. Consequently, diversifying across time and renewing parts of protections shortens the time during which they can develop and derive income from an attack. Therefore, in Chapter 6 we presented the renewability framework. This framework builds on our code mobility technique to support several forms of renewability. Renewed and diversified code and data belonging to either the original application or

to protection components is delivered from a secure server to a client application on demand. The frequent changes to the software components under attack, and consequently make it even harder—and less profitable—to attack the software.

The development of this renewability framework and these improvements to existing techniques have pushed the state of the art in protecting against MATE attacks. They make it harder for attackers to make unauthorized use of valuable assets embedded in software, and thus make attacks both less likely and less profitable. There is still worthwhile research to be done both in improving these protection techniques, and in combining and renewing them. An improvement that should definitely be investigated is reciprocal debugging, where the self-debugger not only debugs the program, but the program also debugs the self-debugger. In the past year I have performed research to that effect, the results of which could not be included in this dissertation for reasons of intellectual property rights. As for combining protections, future work could focus on optimizing these combinations in terms of paths-of-least-resistance, within certain limits to performance overhead. Certain combinations of protection techniques can also create synergies, with the combination being "stronger" as the sum of the individual protections. Investigating such synergies further would be a worthy enterprise, just like investigating the renewability of more protection techniques and even of communication protocols.

In conclusion, I presented a number of improvements to the defensive side of software security. Undoubtedly, in the future attackers will refine their own tools, prompting the development of even better protections. Such, as they say, is life.

Appendix A

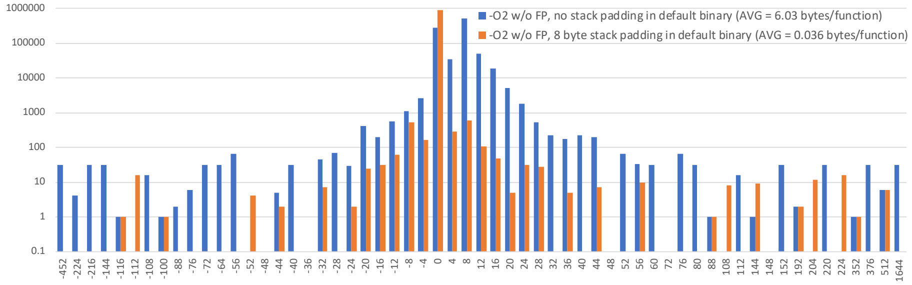
Quantitative Analysis for Δ-Minimization

Histograms (a) and (b) in Figure A.1 quantify the effect of adding (default) padding to functions on their code size. These histograms show how the function code sizes change as a result of adding 32 different amounts of padding (8, 16, ..., 256) to each function in our benchmark suite compiled with `-O2 -fomit-frame-pointer` for part (a) and with `-O2` for part (b)—the histograms look similar with other options. The blue and gray histograms show the changes when the default binary does not include 8 bytes of padding, the orange and purple histograms show the changes when the default binary does include 8 bytes of padding.

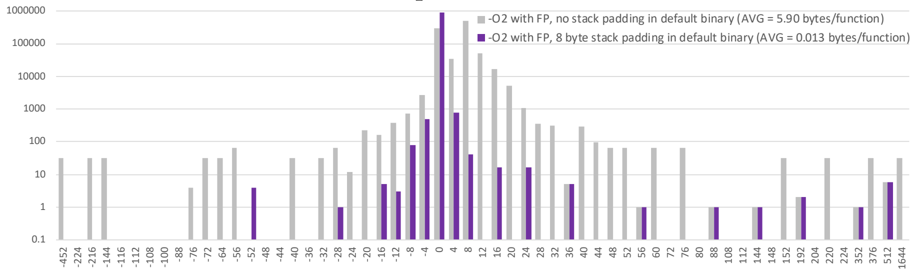
Notice that many size increases and size reductions are obtained exactly 32 or 64 times in the blue and gray histograms. This follows from the fact that the same increase or reduction in size was observed for all of the 32 diversified versions of a specific function compared to its default version without any padding. In the orange and purple histograms, that situation does not occur. Clearly, the changes on average become much smaller with the default padding. The average (absolute values of the) changes are 6.03 (respectively, 5.90) bytes/function without default padding, and only 0.036 (respectively, 0.013) bytes/function with default padding. Also, the orange and purple histograms peak at zero, whereas the blue and gray ones peak at 8. So with the default padding, there are many more functions for which diversified stack padding has no effect at all on code size. Clearly, the default padding of 8 bytes is advantageous for Δ minimization.

These numbers also indicate that the function size deltas between default and diversified files are smaller on average for code compiled

(a) Benchmarks compiled without FP, with and without default padding, and with SP/FP-optimization disabled



(b) Benchmarks compiled with FP, with and without default padding, and with SP/FP-optimization disabled



(c) Benchmarks compiled with FP, with default stack padding, with SP/FP-optimization enabled and disabled

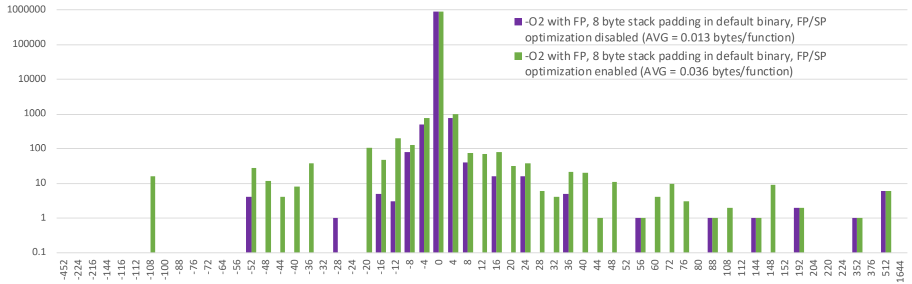


Figure A.1: Histograms of the variation in function size. The Y-axes start at 0.1 to visualize the difference between 0 and 1. The presented average numbers are averages of absolute values of positive and negative variations.

with FPs than for code compiled without FPs. The difference is almost completely due to function versions where the non-zero delta when compiled with FP grows bigger (i.e., more positive or more negative) in code compiled without FP. The number of function versions with zero delta compared to the default 8 byte padding version remains almost

constant with or without FP: Over 99.94% of the 892K function versions (out of 895k total) that do not grow or shrink in our experiments as a result of stack padding when compiled with FP, still do not grow or shrink when compiled without FP.

Histogram (c) in Figure A.1 visualizes the effect on function code size of disabling the SP/FP relative stack access optimization. On average, the difference in size drops from 0.036 bytes/function to 0.013 bytes/function.

Bibliography

[1] ARM. *ARM Architecture Reference Manual – Thumb-2 Supplement*. Pearson Education, 2005.

[2] ARM. Building a Secure System Using TrustZone Technology. *Manual*, page 108, 2009.

[3] Cyrille Artho. Iterative Delta Debugging. *International Journal on Software Tools for Technology Transfer*, 13(3):223–246, jun 2011. ISSN 14332779. doi: 10.1007/s10009-010-0139-9. URL `http://link.springer.com/10.1007/s10009-010-0139-9`.

[4] David Aucsmith. Tamper Resistant Software: An Implementation. In *Information Hiding*, pages 317–333. Springer, 2012. doi: 10.1007/3-540-61996-8_49.

[5] Cataldo Basile and Others. ASPIRE Framework Report. Deliverable D5.11 v1.0, ASPIRE, 2016.

[6] Thomas Bell. The Concept of Dynamic Analysis. *ACM SIGSOFT Software Engineering Notes*, 1999. ISSN 01635948. doi: 10.1145/318774.318944.

[7] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 41–46, 2005. ISBN 1-931971-28-5. doi: 10.1080/15710880600580702. URL `https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/freenix/bellard.html`.

[8] Olivier Billet and Henri Gilbert. A Traceable Block Cipher. In *Advances in Cryptology – ASIACRYPT*, pages 331–346, 2010. doi: 10.1007/978-3-540-40061-5_21.

[9] Dan Bornstein. Dalvik VM Internals. In *Google I/O 2008*, volume 23, pages 17–30, 2008. URL `https://sites.google.com/site/io/dalvik-vm-internals/`.

[10] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure:

SGX Cache Attacks Are Practical. In *USENIX WOOT*, 2017. URL http://arxiv.org/abs/1702.07521.

[11] Julien Bringer, Herve Chabanne, and Emmanuelle Dottax. White Box Cryptography: Another Attempt. Cryptology ePrint Archive, Report 2006/468, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.661&rep=rep1&type=pdf.

[12] Tom Brosch and Maik Morgenstern. Runtime Packers: The Hidden Problem. *Black Hat USA*, 2006. ISSN 0162-3737. doi: 10.3102/0162373711431098. URL http://www.orkspace.net/secdocs/Conferences/BlackHat/USA/2006/RuntimePackers-TheHiddenProblem.pdf.

[13] Carbon Monoxide. ScyllaHide, 2019. URL https://bitbucket.org/NtQuery/scyllahide.

[14] Lorena Castañeda. A Reference Architecture for Software Systems. In *13th Working IEEE/IFIP Conference on Software Architecture*, pages 291–294, 2012.

[15] Mariano Ceccato and Others. Barrier Slicing for Remote Software Trusting. In *International Working Conference on Source Code Analysis and Manipulation*, pages 27–36, 2007.

[16] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. Understanding the Behaviour of Hackers While Performing Attack Tasks in a Professional Setting and in a Public Challenge. *Empirical Software Engineering*, pages 1–47, 2018. ISSN 15737616. doi: 10.1007/s10664-018-9625-6. URL https://doi.org/10.1007/s10664-018-9625-6.

[17] Hoi Chang and Mikhail J. Atallah. Protecting Software Code by Guards. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2320, pages 160–175. Springer, 2002. ISBN 3540436774. doi: 10.1007/3-540-47870-1_10.

[18] Yuqun Chen, Saurabh Sinha, Ramarathnam Venkatesan, Ruoming Pang, Mariusz H. Jakubowski, and Matthew Cary. Oblivious Hashing: A Stealthy Software Integrity Verification Primitive. In *International Workshop on Information Hiding*, pages 400–414. Springer, 2007. doi: 10.1007/3-540-36415-3_26.

[19] Kil Chongkyung, Jun Jinsuk, Christopher Bookholt, Xu Jun, and Ning Peng. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the Annual Computer Security Applications Conference, ACSAC*, pages 339–348, 2006. ISBN 0769527167. doi: 10.1109/ACSAC.2006.9.

[20] Stanley Chow, Philip A Eisen, Harold Johnson, and Paul C van Oorschot. White-Box Cryptography and an AES Implementation. In *Proceedings of the 9th International Workshop on Selected Areas in Cryptography*, pages 250–270, 2002. ISBN 3-540-00622-2.

[21] Cristina Cifuentes and K. John Gough. Decompilation of Binary Programs. *Software: Practice and Experience*, 1995. ISSN 1097024X. doi: 10.1002/spe.4380250706.

[22] Frederick Cohen. Operating System Protection Through Program Evolution. *Operating systems & Technology*, 6(5):45–50, 2013. URL `http://all.net/books/tech/evolve.pdf`.

[23] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of Remote Attestation. *International Journal of Information Security*, 2011. ISSN 16155262. doi: 10.1007/s10207-011-0124-7.

[24] Christian Collberg. The Tigress C Diversifier/Obfuscator, 2019. URL `http://tigress.cs.arizona.edu/`.

[25] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 184–196. ACM, 2003. doi: 10.1145/268946.268962.

[26] Christian Collberg, Jasvir Nagra, and Will Snavely. bianlian: Remote Tamper-Resistance with Continuous Replacement. Technical report, Technical Report TR08-03, Department of Computer Science, University of Arizona, 2008.

[27] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed Application Tamper Detection via Continuous Software Updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, page 319, 2012. ISBN 9781450313124. doi: 10.1145/2420950.2420997.

[28] Christian S. Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic Graph-Based Software Fingerprinting. *ACM Transactions on Programming Languages and Systems*, 29(6):35–es, 2007. ISSN 01640925. doi: 10.1145/1286821.1286826.

[29] Bart Coppens, Bjorn De Sutter, and Koen De Bosschere. Protecting Your Software Updates. *IEEE Security and Privacy*, 11(2):47–54, 2013. ISSN 15407993. doi: 10.1109/MSP.2012.113.

[30] Bart Coppens, Bjorn De Sutter, and Jonas Maebe. Feedback-Driven Binary Code Diversification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):24:1—-24:26, 2013. doi: 10.1145/0000000.0000000.

[31] James R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 2006. ISSN 01676423. doi: 10.1016/j.scico.2006.04.002.

[32] Lucas Davi. Xifer: A Software Diversity Tool Against Code-Reuse Attacks. In *Others*, pages 1–3, 2012. ISBN 9781450315289.

[33] Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Saumya Debray. Combining Global Code and Data Compaction. *ACM SIGPLAN Notices*, 36(8):29–38, 2005. ISSN 03621340. doi: 10.1145/384196.384204.

[34] Bjorn De Sutter, Bertrand Anckaert, Saumya Debray, Koen De Bosschere, Patrick Moseley, and Matias Madou. Software Protection Through Dynamic Code Mutation. In *International Workshop on Information Security Applications (WISA)*, pages 194–206, 2006. ISBN 978-3-540-31012-9. doi: 10.1007/11604938_15.

[35] Saumya Debray, Robert Muth, and Matthew Weippert. Alias Analysis of Executable Code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, 2003. doi: 10.1145/268946.268948.

[36] Chris Eagle. *The IDA Pro Book – The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2011. ISBN 9781593272890.

[37] ECRYPT-CSA Consortium. The WhibOx Contest, An ECRYPT White-Box Cryptography Competition, 2017. URL `https://whibox.cr.yp.to/`.

[38] Frank C Eigler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of SystemTAP: A Linux Trace. *Probe Tool*, 2005. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.109.1364`.

[39] Paolo Falcarin, Riccardo Scandariato, and Mario Baldi. Remote Trust with Aspect-Oriented Programming. In *Proceedings of the International Conference on Advanced Information Networking and Applications, AINA*, 2006. ISBN 0769524664. doi: 10.1109/AINA.2006.286.

[40] Paolo Falcarin, Stefano Di Carlo, Alessandro Cabutto, Nicola Garazzino, and Davide Barberis. Exploiting Code Mobility for Dynamic Binary Obfuscation. In *2011 World Congress on Internet Security (WorldCIS-2011)*, pages 114–120, 2011. ISBN 978-1-4244-8879-7.

[41] Paolo Falcarin, Christian Collberg, Mikhail Atallah, and Mariusz Jakubowski. Guest Editors' Introduction: Software Protection. *IEEE Software*, 28(2):24–27, 2011. ISSN 07407459. doi: 10.1109/MS.2011.34.

[42] Peter Ferrie. Anti-Unpacker Tricks. In *Amsterdam: CARO Workshop*, 2008.

[43] Peter Ferrie. The Ultimate Anti-Debugging Reference, apr 2011. URL http://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf.

[44] Ferrit. OllyExt 1.8, 2014. URL https://tuts4you.com/download.php?view.3392.

[45] S. Forrest, A. Somayaji, and D.H. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 67–72. IEEE Computer Society, 2002. doi: 10.1109/hotos.1997.595185.

[46] Michael Franz. E Unibus Pluram: Massive-Scale Software Diversity as a Defense Mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, pages 7–16, 2010. ISBN 9781450304153. URL http://dl.acm.org/citation.cfm?id=1900550.

[47] Michael Franz, Per Larsen, Andrei Homescu, Stefan Brunthaler, and Stephen Crane. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Proceedings of the Network and Distributed System Security Symposium*, feb 2015. doi: 10.14722/ndss.2015.23264.

[48] Juan A. Garay and Lorenz Huelsbergen. Software Integrity Protection Using Timed Executable Agents. In *Proceedings of the ACM Symposium on Information, computer and communications security*, page 189, 2006. doi: 10.1145/1128817.1128847.

[49] GCC. LinkTimeOptimization, 2009. URL https://gcc.gnu.org/wiki/LinkTimeOptimization.

[50] Gdb. Gdb – GNU Project Debugger, 2016. URL https://www.gnu.org/software/gdb/.

[51] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-Grained Address Space Randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 475–490. USENIX Association, 2012.

[52] GNU. GNU Binary Utilities, 2019. URL https://sourceware.org/binutils/docs/binutils/index.html.

[53] GNU. GNU gprof, 2019. URL https://sourceware.org/binutils/docs/gprof/.

[54] Google. Breakpad, 2019. URL https://chromium.googlesource.com/breakpad/breakpad/.

[55] Brendan D. Gregg. DTrace Tools, 2019. URL http://www.brendangregg.com/dtrace.html.

[56] Y X Gu and Others. Point/Counterpoint. *IEEE Software*, 28(2):56–59, 2011.

[57] Aditi Gupta, Sam Kerr, Michael S Kirkpatrick, and Elisa Bertino. Marlin: A Fine Grained Randomization Approach to Defend Against ROP Attacks. In *7th International Conference on Network and System Security*, pages 293–306, 2013.

[58] David A. Holland, Ada T. Lim, and Margo I. Seltzer. An Architecture a Day Keeps the Hacker Away. *ACM SIGARCH Computer Architecture News*, 33(1):34, 2005. ISSN 01635964. doi: 10.1145/1055626.1055632.

[59] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. librando: Transparent Code Randomization for Just-In-Time Compilers. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & communications security – CCS '13*, pages 993–1004. ACM, 2013. ISBN 9781450324779. doi: 10.1145/2508859.2516675. URL `http://dl.acm.org/citation.cfm?doid=2508859.2516675`.

[60] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-Guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013*, pages 1–11. IEEE Computer Society, 2013. ISBN 9781467355254. doi: 10.1109/CGO.2013.6494997.

[61] Wei Hu, David Evans, Jack W. Davidson, John C. Knight, Adrian Filipi, Anh Nguyen-Tuong, Dan Williams, Jason Hiser, and Jonathan Rowanhill. Secure and Practical Defense Against Code-Injection Attacks Using Software Dynamic Translation. In *Proceedings of the 2nd international Conference on Virtual execution environments*, page 2. ACM, 2006. doi: 10.1145/1134760.1134764.

[62] Amjad Ibrahim and Sebastian Banescu. StIns4CS. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pages 61–71, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4576-7. doi: 10.1145/2995306.2995313. URL `http://doi.acm.org/10.1145/2995306.2995313`.

[63] Internet Engineering Task Force (IETF). Web Sockets, dec 2011. URL `http://tools.ietf.org/html/rfc6455`.

[64] Markus Jakobsson and Michael K. Reiter. Discouraging Software Piracy Using Software Aging. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2002. ISBN 3540436774. doi: 10.1007/3-540-47870-1_1.

[65] Jean. hack.lu CTF – Challenge 12 WriteUp. Technical report, Sogeti ESEC Lab, 2010. URL `http://esec-lab.sogeti.com/posts/2010/11/29/hacklu-ctf-challenge-12-writeup.html`.

[66] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM-Software Protection for the Masses. In *Proceedings of the International Workshop on Software Protection, SPRO 2015*, pages 3–9. IEEE, 2015. ISBN 9781467370943. doi: 10.1109/SPRO.2015.10.

[67] Mateusz Jurczyk. Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8), 2019. URL `http://j00ru.vexillium.org/ntapi/`.

[68] James E. Just and Mark Cornwell. Review and Analysis of Synthetic Diversity for Breaking Monocultures. In *Proceedings of the 2004 ACM workshop on Rapid malcode*, page 23. ACM, 2005. doi: 10.1145/1029618.1029623.

[69] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting Self-Modification Mechanism for Program Protection. In *Proceedings of the International Conference Computer Software and Applications*, pages 170–179, 2004. doi: 10.1109/cmpsac.2003.1245338.

[70] Pradeep Khosla, Arvind Seshadri, Leendert van Doom, Mark Luk, and Adrian Perrig. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. *ACM SIGOPS Operating Systems Review*, 39(5):253–289, 2007. doi: 10.1007/978-0-387-44599-1_12.

[71] James C King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[72] Per Larsen, Stefan Brunthaler, and Michael Franz. Security Through Diversity: Are We There Yet? *IEEE Security and Privacy*, 12(2):28–35, 2014. ISSN 15407993. doi: 10.1109/MSP.2013.129.

[73] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE Computer Society, 2014. doi: 10.1109/SP.2014.25.

[74] Per Larsen, Stefan Brunthaler, and Michael Franz. Automatic Software Diversity. *IEEE Security and Privacy*, 13(2):30–37, 2015. ISSN 15584046. doi: 10.1109/MSP.2015.23.

[75] Per Larsen, Stefan Brunthaler, and Michael Franz. Error Report Normalization, 2016.

[76] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, pages 557–574, 2016. ISBN 978-1-931971-40-9. URL `http://arxiv.org/abs/1611.06952`.

[77] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

[78] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. *Proceedings of the 10th ACM Conference on Computer and communication security – CCS '03*, page 290, 2004. ISSN 15437221. doi: 10.1145/948109.948149.

[79] Linux Programmer's Manual. dlopen(3) – Linux manual page, 2017. URL `http://man7.org/linux/man-pages/man3/dlopen.3.html`.

[80] Linux Programmer's Manual. fork(2) – Linux manual page, 2017. URL `http://man7.org/linux/man-pages/man2/fork.2.html`.

[81] Linux Programmer's Manual. proc(5) – Linux manual page, 2019. URL `http://man7.org/linux/man-pages/man5/proc.5.html`.

[82] LLVM. LLVM Link Time Optimization: Design and Implementation, 2019. URL `https://llvm.org/docs/LinkTimeOptimization.html#llvm-link-time-optimization-design-and-implementation`.

[83] Matias Madou, Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Hybrid Static-Dynamic Attacks Against Software Protection Mechanisms. In *Proceedings of the 5th ACM workshop on Digital rights management*, page 75. ACM, 2006. doi: 10.1145/1102546.1102560.

[84] Henry Miller. Beginners Guide to Basic Linux Anti-Anti-Debugging Techniques. *CodeBreakers Journal*, 2005.

[85] Mrexodia. TitanHide, 2019. URL `https://github.com/mrexodia/TitanHide`.

[86] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[87] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! *ACM SIGARCH Computer Architecture News*, 37(1):265, 2013. ISSN 01635964. doi: 10.1145/2528521.1508275.

[88] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009. ISBN 0132702037. URL `https://books.google.com/books?id=mig-bH3u0Z0C&pgis=1`.

[89] Nicholas Nethercote and Julian Seward. Valgrind: A Framework For Heavyweight Dynamic Binary Instrumentation. *a Framework for Heavyweight Dynamic Binary Instrumentation*, 42(6):89–100, 2007. ISSN <null>. doi: 10.1145/1250734.1250746.

[90] Oreans Technologies. Themida – Advanced Windows Software Protection System, 2016. URL `http://www.oreans.com/themida.php`.

[91] Oreans Technologies. Code Virtualizer: Total Obfuscation Against Reverse Engineering, 2017. URL www.oreans.com/codevirtualizer.php.

[92] Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D. Keromytis. Retrofitting Security in COTS Software with Binary Rewriting. In *IFIP Advances in Information and Communication Technology*, 2011. ISBN 9783642214233. doi: 10.1007/978-3-642-21424-0_13.

[93] Pancake. radare, 2019. URL https://rada.re/r/.

[94] PaX Team. Address Space Layout Randomization, 2003. URL http://pax.grsecurity.net/docs/aslr.txt.

[95] Pellsson. Starcraft 2 Anti-Debugging, 2010. URL http://www.bhfiles.com/files/StarCraftII/WingsofLiberty%28Beta%29/0x1337.org--SCIIAnti-Debug.htm.

[96] Riccardo Scandariato, Yoram Ofek, Paolo Falcarin, and Mario Baldi. Application-Oriented Trust in Distributed Computing. In *Proceedings of the 3rd International Conference on Availability, Security, and Reliability*, pages 434–439, 2008. ISBN 0769531024. doi: 10.1109/ARES.2008.21.

[97] Tyler Shields. Anti-Debugging – A Developers View. Technical report, Veracode, 2009. URL http://www.shell-storm.org/papers/files/764.pdf.

[98] Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu. Code Shredding: Byte-Granular Randomization of Program Layout for Detecting Code-Reuse Attacks. In *ACSAC '12* (*Annual Computer Security Applications Conference*), pages 309–318, 2012. ISBN 9781450313124. doi: 10.1145/2420950.2420996.

[99] StrongBit Technology. EXECryptor: Bulletproof Software Protection, 2018.

[100] Michael Sutton, Adam Greene, and Perdam Amini. *Fuzzing – Brute Force Vulnerability Discovery*. Pearson Education, 2007. ISBN 0321446119. URL http://bxi.es/Reversing-Exploiting/FuzzingBruteForceVulnerabilityDiscovery.pdf.

[101] Laszlo Szekeres, Mathias Payer, Lenx Tao Wei, and R. Sekar. Eternal War in Memory. *IEEE Security & Privacy*, 12(3):45–53, 2014. ISSN 1540-7993. doi: 10.1109/msp.2014.44.

[102] The ASPIRE Consortium. ASPIRE, 2016. URL https://aspire-fp7.eu/.

[103] Ubuntu Wiki. SecurityTeam/Roadmap/KernelHardening – Ubuntu Wiki, 2013. URL https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace_Protection.

[104] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. DIABLO: A Reliable, Retargetable and Extensible Link-Time Rewriting Framework. *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, 2005:7–12, 2005. doi: 10.1109/ISSPIT.2005.1577061.

[105] Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bert Abrath, and Bart Coppens. Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 73–84, 2016. ISBN 978-1-4503-4576-7. doi: 10.1145/2995306.2995315. URL `http://doi.acm.org/10.1145/2995306.2995315`.

[106] VMProtect Software. VMProtect Software Protection, 2012. URL `www.vmprotect.ru`.

[107] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, 2016. ISSN 15455971. doi: 10.1109/TDSC.2015.2411254.

[108] Gregor Wagner, Todd Jackson, Michael Franz, Christian Wimmer, Andreas Gal, Karthikeyan Manivannan, Andrei Homescu, Stefan Brunthaler, and Babak Salamat. Compiler-Generated Software Diversity. In *Moving Target Defense*, pages 77–98. Springer, 2011. doi: 10.1007/978-1-4614-0977-9_4.

[109] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of Software-based Survivability Mechanisms. In *Science*, pages 193–202. IEEE, 2001.

[110] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and communications security – CCS '12*, page 157. ACM, 2012. ISBN 9781450316514. doi: 10.1145/2382196.2382216. URL `http://dl.acm.org/citation.cfm?id=2382216`.

[111] Michael E Whitman and Herbert J Mattord. *Principles of Information Security*. Cengage Learning, 2014. ISBN 9781111138219. doi: 10.1016/B978-0-12-381972-7.00002-6.

[112] Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. Security Through Diversity: Leveraging Virtual Machine Technology. *IEEE Security and Privacy*, 7(1):26–33, 2009. ISSN 15407993. doi: 10.1109/MSP.2009.18.

[113] Brecht Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, 2009.

[114] Brecht Wyseur. Let's Get Real! We Need WBC and Io. WhibOx 2016, workshop on White-Box Cryptography and Obfuscation, 2016.

[115] Brecht Wyseur and Others. ASPIRE Attack Model. Deliverable D1.02 v2.1, ASPIRE, 2016.

[116] Brecht Wyseur, Bjorn De Sutter, and Others. ASPIRE Reference Architecture. Deliverable D1.04 v2.1, ASPIRE, 2016.

[117] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 260–269. IEEE, 2003. ISBN 0769519555. doi: 10.1109/RELDIS.2003.1238076.

[118] Yama. ptrace_scope, 2019. URL `https://www.kernel.org/doc/Documentation/security/Yama.txt`.