

# Contents

<b>Acknowledgements</b>	<b>ix</b>
<b>Contents</b>	<b>x</b>
<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xvi</b>
<b>List of Listings</b>	<b>xvii</b>
<b>Publications</b>	<b>xix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Problem Statement . . . . .	1
1.2 Research Questions . . . . .	3
1.3 Contributions and Outline . . . . .	5
<b>2 Background</b>	<b>7</b>
2.1 Binary Analysis Primitives . . . . .	7
2.2 The x86/x64 Instruction Set Architecture . . . . .	8
2.3 Static versus Dynamic Analysis . . . . .	8
2.4 Disassembly Methods . . . . .	9
2.5 Binary Instrumentation . . . . .	10
2.6 Symbols, DWARF, PDB and Stripped Binaries . . . . .	11
2.7 Complex Constructs: Challenges for Static Analysis . . . . .	12
<b>I Retrofitting Security Into Stripped Binaries</b>	<b>15</b>
<b>Outline</b>	<b>17</b>
<b>3 StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries</b>	<b>19</b>
3.1 Introduction . . . . .	19

3.2	Threat Model . . . . .	21
3.2.1	Spatial Attacks . . . . .	21
3.2.2	Temporal Attacks . . . . .	22
3.2.3	Defenses . . . . .	22
3.3	StackArmor Overview . . . . .	23
3.3.1	Stack Protection Analyzer . . . . .	24
3.3.2	Definite Assignment Analyzer . . . . .	26
3.3.3	Buffer Reference Analyzer . . . . .	27
3.3.4	Stack Frame Allocator . . . . .	29
3.3.5	Binary Rewriter . . . . .	31
3.4	Implementation . . . . .	33
3.4.1	Binary Disassembly and Analysis . . . . .	33
3.4.2	Instrumentation . . . . .	34
3.4.3	Limitations . . . . .	35
3.5	Evaluation . . . . .	35
3.5.1	Security Against Spatial Attacks . . . . .	36
3.5.2	Security Against Temporal Attacks . . . . .	37
3.5.3	Performance . . . . .	38
3.5.4	Memory Usage . . . . .	40
3.5.5	Multithreading Support . . . . .	41
3.6	Related Work . . . . .	42
3.7	Conclusion . . . . .	43
<b>4</b>	<b>Practical Context-Sensitive Control-Flow Integrity</b>	<b>45</b>
4.1	Introduction . . . . .	45
4.2	Context-sensitive CFI . . . . .	47
4.2.1	Legal flows . . . . .	47
4.2.2	Challenges . . . . .	49
4.3	PathArmor Overview . . . . .	50
4.3.1	Kernel Module . . . . .	51
4.3.2	Path Analyzer . . . . .	53
4.3.3	Binary Instrumentation . . . . .	55
4.4	Implementation . . . . .	57
4.5	Evaluation . . . . .	58
4.5.1	Security . . . . .	58
4.5.2	Memory Usage . . . . .	62
4.5.3	Analysis Time . . . . .	63
4.5.4	Runtime Performance . . . . .	64
4.5.5	LBR Pollution . . . . .	65
4.6	Discussion . . . . .	66
4.6.1	History Flushing Attacks . . . . .	66
4.6.2	Non-control Data Attacks . . . . .	67
4.6.3	Endpoint-pruning Attacks . . . . .	67

4.6.4	Instrumentation-tampering Attacks . . . . .	68
4.7	Related Work . . . . .	68
4.8	Conclusion . . . . .	70
<b>5</b>	<b>Parallax: Implicit Binary Code Integrity Verification Using Return-Oriented Programming</b>	<b>73</b>
5.1	Introduction . . . . .	73
5.2	Background . . . . .	76
5.2.1	Return-Oriented Programming . . . . .	76
5.2.2	Threat Model . . . . .	76
5.3	Parallax Overview . . . . .	77
5.4	Protecting Code Integrity . . . . .	78
5.4.1	A Tamperproofed <code>Ptrace</code> Detector . . . . .	78
5.4.2	Binary Rewriting Rules . . . . .	81
5.5	Verifying Code Integrity . . . . .	83
5.5.1	Implementation of Function Chains . . . . .	83
5.5.2	Dynamically Generated Function Chains . . . . .	84
5.5.3	Instruction-Level Verification . . . . .	86
5.6	Attack Resistance . . . . .	86
5.6.1	Code Restoration . . . . .	87
5.6.2	Verification Code Replacement . . . . .	87
5.6.3	Verification Code Modification . . . . .	87
5.7	Evaluation . . . . .	88
5.7.1	Protectable Code Locations . . . . .	88
5.7.2	Runtime Overhead . . . . .	89
5.8	Discussion and Limitations . . . . .	90
5.8.1	Dynamic Circumvention . . . . .	90
5.8.2	Control-Flow Integrity . . . . .	91
5.8.3	Protection Coverage . . . . .	91
5.9	Related Work . . . . .	92
5.10	Conclusion . . . . .	93
	<b>Discussion</b>	<b>95</b>
<b>II</b>	<b>Evaluating and Improving Disassembly</b>	<b>99</b>
	<b>Outline</b>	<b>101</b>
<b>6</b>	<b>An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries</b>	<b>103</b>
6.1	Introduction . . . . .	103
6.2	Evaluating Real-World Disassembly . . . . .	105
6.2.1	Binary Test Suite . . . . .	105
6.2.2	Disassembly Primitives . . . . .	106

6.2.3	Complex Constructs . . . . .	106
6.2.4	Disassembly & Testing Environment . . . . .	107
6.2.5	Ground Truth . . . . .	107
6.3	Disassembly Results . . . . .	108
6.3.1	Application Binaries . . . . .	108
6.3.2	Shared Library Objects . . . . .	118
6.3.3	Static Linking & Link-time Optimization . . . . .	120
6.4	Implications of Results . . . . .	121
6.4.1	Control-Flow Integrity . . . . .	121
6.4.2	Decompilation . . . . .	123
6.4.3	Automatic Bug Search . . . . .	123
6.5	Disassembly in the Literature . . . . .	124
6.6	Discussion . . . . .	127
6.7	Related Work . . . . .	128
6.8	Conclusion . . . . .	129
<b>7</b>	<b>Compiler-Agnostic Function Detection in Binaries</b>	<b>131</b>
7.1	Introduction . . . . .	131
7.2	Background . . . . .	133
7.2.1	Definition of Function Detection . . . . .	133
7.2.2	Scope of Function Detection . . . . .	134
7.2.3	Signature-Based Approaches . . . . .	134
7.2.4	Challenging Cases . . . . .	135
7.3	Nucleus Overview . . . . .	135
7.3.1	ICFG Generation . . . . .	135
7.3.2	Connected Components Analysis . . . . .	137
7.4	Implementation . . . . .	137
7.4.1	Disassembly and ICFG Generation . . . . .	138
7.4.2	Switch Detection . . . . .	138
7.4.3	Function and Entry Point Detection . . . . .	138
7.5	Evaluation . . . . .	139
7.5.1	Test Setup . . . . .	139
7.5.2	Function Detection Results . . . . .	140
7.5.3	Analysis of Results . . . . .	143
7.5.4	Runtime Performance . . . . .	146
7.6	Analysis of Machine Learning in Function Detection . . . . .	146
7.6.1	Function Detection Performance . . . . .	147
7.6.2	Evaluation Methodology . . . . .	148
7.7	Discussion . . . . .	149
7.8	Related Work . . . . .	151
7.9	Conclusion . . . . .	151
	<b>Discussion</b>	<b>153</b>

<b>8 Conclusions</b>	<b>155</b>
8.1 Results . . . . .	155
8.2 Limitations and Future Work . . . . .	156
<b>References</b>	<b>159</b>
<b>Contributions to Papers</b>	<b>177</b>
<b>Summary</b>	<b>179</b>
<b>Samenvatting</b>	<b>181</b>

# List of Figures

2.1	Linear versus recursive disassembly . . . . .	10
3.1	Comparison of stack protection techniques . . . . .	23
3.2	High-level overview of <i>StackArmor</i> . . . . .	24
3.3	Sample <i>SP-unsafe</i> function . . . . .	25
3.4	Sample <i>DA-unsafe</i> function . . . . .	27
3.5	Sample function with an ambiguous stack reference . . . . .	28
3.6	<i>StackArmor</i> 's stack frame allocation strategy . . . . .	30
3.7	Call site instrumentation in <i>StackArmor</i> . . . . .	32
3.8	SPEC CPU2006 performance overhead . . . . .	38
3.9	RSS increase due to <i>StackArmor</i> . . . . .	41
3.10	RSS increase due to <i>StackArmor</i> in multithreaded programs . . . . .	41
4.1	Overview of <i>PathArmor</i> . . . . .	50
4.2	CDF of gadgets permitted by $\overline{\text{CCFI}}$ and CCFI . . . . .	61
4.3	Fraction of single-target backward edges for CCFI when simulating an increasingly large LBR . . . . .	62
4.4	Runtime normalized against the baseline for SPEC CPU2006 . . . . .	66
5.1	An example ROP chain . . . . .	76
5.2	A high-level overview of <i>Parallax</i> . . . . .	77
5.3	Verification at function and instruction level . . . . .	84
5.4	Generating a function chain by combining vectors from a basis $B$ . . . . .	85
5.5	Code bytes protectable by rules from Section 5.4.2 . . . . .	88
5.6	Slowdowns and whole-program overheads for function chains . . . . .	90
6.1	Results of our disassembly precision measurements . . . . .	109
6.2	False positives for function start detection . . . . .	112
6.3	Prevalence of complex constructs in SPEC CPU2006 binaries . . . . .	117
7.1	Overview of the <i>Nucleus</i> function detection algorithm . . . . .	136
7.2	F-scores for function start detection . . . . .	141
7.3	F-scores for function boundary detection . . . . .	143
7.4	Runtime performance for function boundary detection . . . . .	146

# List of Tables

3.1	Mean attack surface reduction induced by <i>StackArmor</i> . . . . .	37
3.2	Normalized benchmark runtimes . . . . .	38
3.3	Instrumentation decisions and call stack statistics for SPEC CPU2006 . .	39
4.1	Control flow properties of evaluated programs . . . . .	59
4.2	Comparison of permitted control flows . . . . .	60
4.3	Fraction of legal indirect targets for (ideal binary-level) context-sensitive versus context-insensitive forward-edge CFI . . . . .	63
4.4	Path analysis time and runtime cache statistics . . . . .	64
4.5	Runtime normalized against the baseline and benchmark statistics . . . .	65
4.6	LBR pollution caused by library calls for SPEC CPU2006 . . . . .	67
6.1	IDA Pro 6.7 disassembly results for server tests . . . . .	116
6.2	Disassembly results for <code>glibc</code> . . . . .	118
6.3	IDA Pro 6.7 disassembly results for static and link-time optimized SPEC C benchmarks . . . . .	120
6.4	Primitives/disassemblers used in the literature . . . . .	124
6.5	Set of papers discussed in the literature study . . . . .	125
6.6	Properties and assumptions of binary-based papers . . . . .	126
7.1	Precision/recall for function start detection . . . . .	142
7.2	Precision/recall for function boundary detection . . . . .	144
7.3	Precision/recall/F-scores for function start and boundary detection com- pared to machine learning-based work . . . . .	147

# List of Listings

1.1	Example of disassembled switch statement . . . . .	2
5.1	A <code>ptrace</code> detector with gadgets overlapping sensitive areas . . . . .	79
5.2	An attempt to disable the <code>ptrace</code> detector . . . . .	80
5.3	A split <code>mov</code> with overlapping gadgets . . . . .	82
6.1	False negative indirectly called function for IDA Pro . . . . .	113
6.2	False positive function for Dyninst, misapplied prologue signature . .	113
6.3	False positive function for Dyninst, code misinterpreted as epilogue (x86 ELF) . . . . .	113
6.4	False positive function for Dyninst, code misinterpreted as epilogue (x64 ELF) . . . . .	114
6.5	False positive function for Hopper, misclassified switch case block .	114
6.6	Overlapping instruction in <code>glibc-2.22</code> . . . . .	119
6.7	Multi-entry function in <code>glibc-2.22</code> . . . . .	119
7.1	Effective <code>nop</code> instructions emitted by <code>gcc v5.1.1</code> on x86 . . . . .	138
7.2	False negative due to <code>tailcall</code> . . . . .	145
7.3	False negative due to <code>fallthrough</code> from non-returning <code>call</code> . . . . .	145