

Cybersecurity Architecture

The Design and Security of Modern Computing Systems

Bruce Jacob

with contributions from **William Casey** and **Anthony Melaragno**

**United States Naval Academy
Cyber Science Department**

Disclaimer: The views within do not necessarily represent the views or opinions of the U. S. Naval Academy, Department of the Navy, or Department of Defense (DoD) or any of its components.

Copyright © 2023, 2024, 2025 Bruce Jacob

Copyright © 2025 Will Casey and Anthony Melaragno

Dedication

To my wife Valerie — thank you for giving me the best life imaginable

Contents

Prefacexiii

Part I: Fundamentals 1

1.Computing-System Design, Operation, and Security3

- 1.1. Perspective: Why Teach the Two Topics Together3
- 1.2. Background/Review: Stuff You [should] Already Know4
 - 1.2.1. *What Is Programming?*4
 - 1.2.2. *What Does it Mean to ‘Execute’ Code?*7
- 1.3. Fundamentals of Code Representation8
 - 1.3.1. *Step 1: The Compiler Translates High-Level Code to Assembly*8
 - 1.3.2. *Step 2: The Assembler Translates Assembly to Machine Code*11
- 1.4. Fundamentals of Microprocessor Design13
 - 1.4.1. *The Processor’s Fetch-Execute Cycle*14
 - 1.4.2. *The Internal Architecture of the Microprocessor*18
- 1.5. Fundamentals of Logic Design20
 - 1.5.1. *Combinational Logic*21
 - 1.5.2. *Sequential Logic (Memory Logic)*24
- 1.6. Fundamentals of Operating Systems27
 - 1.6.1. *What Is an Operating System?*27
 - 1.6.2. *The Operating System Is the First Line of Defense*30
- 1.7. Fundamentals of Microsystem Security31
 - 1.7.1. *Some Basics*32
 - User Authentication*32
 - Privileged Mode*33
 - Privileged Utilities*33
 - 1.7.2. *Attacking and Protecting the Logic*34
 - Forbidden (Unaccounted-for) States*34
 - Data and Control Inputs*35
 - Clock Inputs*35
 - 1.7.3. *Attacking and Protecting the Microprocessor and its Subsystems*36
 - Handling Privilege Levels in a Pipelined or Multi-Issue Microprocessor*36
 - Virtual Memory and I/O*37
 - Spectre/Meltdown Vulnerabilities*39
 - Side-Channel Analysis, More Generally*42
 - 1.7.4. *Attacking and Protecting the Programming Language*42
 - 1.7.5. *Attacking and Protecting the Operating System*47
 - Stack Values Exposed*47
 - Side-Channel Exploits*48
- 1.8. The Scope of the Book48

2.Digital Logic and Digital Circuits49

- 2.1. Circuitry: The Expression of Logic as Hardware49
- 2.2. Additional Examples of Digital Logic Circuits52
 - 2.2.1. *Wider MUXes*52
 - 2.2.2. *Arithmetic Logic Unit, with AND Unit and OR Unit*54
 - 2.2.3. *Adder*57
 - 2.2.4. *Encoder vs. Priority Encoder (or MUX vs. Priority MUX)*61
- 2.3. Sequential Digital Logic Circuits64
 - 2.3.1. *Verilog Representation*65

<i>Non-Blocking Assignment</i>	66
<i>The @(clockedge) Assignment Block</i>	67
<i>Provide a Way to Reset the State</i>	69
2.3.2. <i>Philosophy of Hardware Design</i>	70
2.3.3. <i>Hardware Design Steps</i>	71
2.4. Examples of Sequential Digital Logic Circuits	72
2.4.1. <i>Counters</i>	72
2.4.2. <i>Saturating Counters</i>	74
2.4.3. <i>(Saturating) Up/Down Counters</i>	78
2.4.4. <i>Shift Registers</i>	81
2.4.5. <i>Linear Feedback Shift Registers</i>	83
<i>Fibonacci LFSRs</i>	84
<i>Galois LFSRs</i>	85
2.4.6. <i>Memory Cells — The Process of Retaining Information</i>	87
2.5. Logic Gates are Made of Transistors	90
2.5.1. <i>MOSFETs</i>	90
2.5.2. <i>Logic Gates</i>	92
<i>Inverter</i>	92
<i>NAND and NOR Gates</i>	95
2.6. Some Latch and Register Designs	97
2.6.1. <i>Example Latches and Their Behavior</i>	97
2.6.2. <i>Latch Load, Clocking, and Transparency</i>	99
2.6.3. <i>Dual-Edge Clocking (Double Data Rate)</i>	105
2.7. Fundamentals of Timing in Digital Circuits	106
2.7.1. <i>How Fast Can I Run My State Machine?</i>	106
2.7.2. <i>How Fast Can I Send Data?</i>	109
▶ Project 0: Get Your Verilator Environment Up & Running	114
▶ Project 2.1: Build an Arithmetic Logic Unit	116
▶ Project 2.2: Build a Saturating Down-Counter with Init	119
▶ Project 2.3: Build [Saturating] Up/Down Counters	121
▶ Project 2.4: Build a Linear Feedback Shift Register	125
▶ Project 2.5: Build a Sequential Adder	131
▶ Project 2.6: Build a Sequential Multiplier	133
2.8. Security Issues Regarding Digital Circuits	135
2.8.1. <i>Forbidden Logic States</i>	135
2.8.2. <i>Electromagnetic Attacks on Pins</i>	138
3. The Microprocessor	141
3.1. The Function of a Microprocessor	141
3.2. The Design of a Microprocessor	143
3.2.1. <i>Data-Flow for ALU-Type Instructions</i>	144
3.2.2. <i>Data-Flow for ALU-Immediate Instructions</i>	145
3.2.3. <i>Data-Flow for Memory Instructions</i>	146
<i>Load-Word</i>	146
<i>Store-Word</i>	147
3.2.4. <i>One Circuit Configuration To Handle All Instructions</i>	148
<i>ALU-Type Instructions</i>	150
<i>Immediate-Type Instructions</i>	150
<i>Load-Word Instructions</i>	151
<i>Store-Word Instructions</i>	151
3.3. A Specific Design: The RiSC-16 (rev. 3) Instruction-Set Architecture. 151	
3.3.1. <i>The Instructions</i>	151

3.3.2. <i>The Register File</i>	152
3.3.3. <i>What the Various Instructions Do</i>	153
3.3.4. <i>Immediate Values</i>	156
3.3.5. <i>Full ISA at End of Book</i>	156
3.4. RISC-16 Instruction Control and Dataflow	156
3.4.1. <i>ADD Instruction</i>	157
3.4.2. <i>ADDI Instruction</i>	158
3.4.3. <i>NAND Instruction</i>	159
3.4.4. <i>LW Instruction</i>	160
3.4.5. <i>SW Instruction</i>	161
3.4.6. <i>BEZ Instruction</i>	162
3.4.7. <i>JALR Instruction</i>	163
3.5. Putting It All Together	164
3.5.1. <i>ALU-Type Instructions, Redux</i>	166
3.5.2. <i>Immediate-Type Instructions, Redux</i>	168
3.5.3. <i>Load-Word Instructions, Redux</i>	170
3.5.4. <i>Store-Word Instructions, Redux</i>	172
3.5.5. <i>Branch Instructions</i>	174
3.5.6. <i>Jump Instructions</i>	176
▶ Project 3.1: Build the RISC-16 Assembler	177
▶ Project 3.2: Build the RISC-16 Core (Structural Design)	191
▶ Project 3.3: Write Some Assembly Code	198
▶ Project 3.4: Build the RISC-16 Core (Behavioral Design)	204
3.6. Synthesis and A Tale of Two Solutions	207
3.6.1. <i>What Hath Verilog Wrought?</i>	210
3.6.2. <i>Synthesizable Verilog</i>	211
3.7. Security Issues Regarding the Assembler	215
3.8. Security Issues Regarding the Microprocessor	215
4. Performance, Caches, and Pipelines	217
4.1. Calculating Performance	217
4.1.1. <i>Execution Time = TC</i>	218
<i>An Example Calculation</i>	219
4.1.2. <i>Amdahl's Law</i>	220
4.1.3. <i>Cost/Performance and Pareto Optimality</i>	221
4.2. Microprocessor Caches	224
4.2.1. <i>Important Cache Principles</i>	227
<i>The Principle of Locality</i>	227
<i>Cache Inclusion</i>	228
<i>The Four Types of Cache Misses</i>	229
4.2.2. <i>How Your Data Aligns in the Cache</i>	230
4.2.3. <i>Cache Organization, Briefly</i>	236
4.3. Microprocessor Pipelines: In-Order, Single-Issue	239
4.3.1. <i>Overview: Pipeline Organization and Pipeline Registers</i>	242
4.3.2. <i>Details: Data Forwarding and Register-File Bypassing</i>	249
<i>Sidebar: Do We Forward into the Decode Stage or the Execute Stage?</i>	253
<i>Back to Our Forwarding Example, a Cycle-by-Cycle Breakdown</i>	257
4.3.3. <i>Details: Load-Use Interlocks</i>	259
4.3.4. <i>Details: Multiply/Divide Interlocks</i>	263
4.3.5. <i>Details: Branch Prediction (Briefly)</i>	263
▶ Project 4.1: Build a Pipelined Microprocessor	265
4.4. Interrupts & Exceptions Are Handled at the End	275
4.5. Calculating Pipeline Performance	276

4.5.1. Pipeline Performance with Perfect Caches.....	276
4.5.2. Performance Including Cache Effects	278
4.6. Security Issues Regarding Caches and Pipelines.....	278

Part II: Elaborations 281

5. Hardware Support for Operating Systems283

5.1. Operating System Overview	283
5.1.1. The Purpose of the Operating System	284
5.1.2. Operating System Functions and Hardware Support	285
5.2. Multitasking (Execution of Multiple Programs)	286
5.2.1. Address-Space Protection via Virtual Memory	287
Virtual Memory Translates Between User Addresses & Physical Memory ...	288
Virtual Memory Mapping Function: The Page Table	289
Address Translation via Hardware: The Translation Lookaside Buffer	293
How to Manage Multiple Processes — Use a Per-Process ID	294
So What Again Is Address-Space Protection?	295
5.2.2. The Illusion of Simultaneity and the Use of Interrupts	297
The Vectored Interrupt Facility	298
Periodic Regaining of Control	301
5.2.3. Hardware Support for Process Context Switching	302
5.3. Managed Access to Input/Output and Protected Files	303
5.3.1. Traps and Returns — Moving in and out of Privileged Mode	304
Trap into the Kernel	304
Context Switch — Saving and Restoring Context	307
Returning Result Values to the User Application	309
5.3.2. Privileged Instructions	311
5.3.3. Protected Regions of Memory, Including Control Registers	312
5.3.4. So How Does This Protect Files and I/O Devices?	313
5.4. Security Issues	314
5.4.1. Virtual Memory Is a Big Deal	314
5.4.2. Hardware Support for Virtual Memory	315
5.4.3. A Secure Means to Enter and Exit Privileged Mode	315

6. The [Backend of the] Compiler317

6.1. The Design of a Compiler	317
6.1.1. Extending the RISC-16 Instruction Set to Assist the Compiler	318
New Instructions	319
New Pseudo-Instructions	320
Large-Immediate Instructions	320
S.LT, S.LTE, S.EQ, and BNZ	321
BOOL	322
INV	323
JAL	324
Other Instructions	324
6.1.2. Arithmetic/Logic Operations — Global Variables	324
6.1.3. More Complex Computations	326
6.1.4. Automatic/Stack Variables	326
6.1.5. Pointers	329
6.1.6. Array Accesses	331
6.1.7. Loop Control Structures: While Loops	333
6.1.8. Loop Control Structures: For Loops	335
6.1.9. Handling ‘continue’ and ‘break’ Constructs	337
Continue	337
Break	339
6.1.10. Handling Nested Loops	343
6.1.11. If-Then-Else Control Structures	345

<i>If-Then</i>	345
<i>If-Then-Else</i>	347
<i>If-Then-Else-If</i>	347
6.1.12. <i>Functions: The Stack</i>	349
<i>The RiSC-16 Stack</i>	349
<i>Stack Operations</i>	352
6.1.13. <i>Functions: The Function Call, Entry, and Return</i>	354
<i>Function Call</i>	355
<i>Function Entry</i>	356
<i>Function Return</i>	356
6.2. <i>Ridiculously Simple C</i>	357
▶ <i>Project 6.1: Build a Simple _start Routine</i>	360
▶ <i>Project 6.2: Extend _start to Create argc and argv</i>	362
▶ <i>Project 6.3: Implement Support for do-while Loops</i>	365
6.3. <i>Security Issues Regarding the Compiler</i>	366
6.3.1. <i>The Use of Privileged Instructions</i>	367
6.3.2. <i>The Question of Compiler Correctness</i>	367
<i>Register Accesses Replace Memory Accesses</i>	367
<i>Other Optimizations</i>	369

7. The Kernel373

7.1. <i>The Design of a Kernel</i>	373
7.1.1. <i>Organization and Privilege Levels</i>	374
7.1.2. <i>Operating System Functions, an Example of the Kernel in Action</i>	376
7.1.3. <i>The Kernel Is Not a Server — It Is a State Machine</i>	384
7.2. <i>Process Management</i>	387
7.2.1. <i>The Process Control Block, Process Table, and User Area (u. struct)</i>	388
7.2.2. <i>Context Save/Restore and Returning Results from the Kernel</i>	390
7.2.3. <i>Scheduling, Including Multicore Issues</i>	392
<i>Multi-Level Feedback Queue (Single-Processor Heuristic)</i>	393
<i>Lottery Scheduling (Single-Processor Heuristic)</i>	394
<i>Uniprocessor Scheduling vs. Multiprocessor Scheduling</i>	394
<i>Gang Scheduling</i>	396
<i>Affinity Scheduling</i>	396
<i>Creech Scheduling — SCAF</i>	398
<i>Conclusion</i>	400
7.3. <i>Interrupt Handling</i>	401
7.3.1. <i>Interrupt Types and Their Values</i>	401
7.3.2. <i>The Vector Table</i>	402
<i>An Arm Variant</i>	403
<i>The Equivalent RiSC-16 Design</i>	405
7.3.3. <i>Interrupt Handlers</i>	405
7.4. <i>Virtual Memory</i>	410
7.4.1. <i>Some Alternatives for Managing Memory</i>	411
<i>One Alternative: Physical Addressing</i>	411
<i>Another Alternative: Base+Offset Addressing (Segmentation)</i>	411
<i>A Third Alternative: Virtual Addressing at a Page Granularity</i>	412
<i>Conclusion</i>	413
7.4.2. <i>Virtual Memory Fundamentals</i>	413
7.4.3. <i>Paged Segmentation</i>	417
<i>General Implementation and Flow</i>	419
7.4.4. <i>Page Table Designs: Hierarchical Tables</i>	421
<i>Hierarchical Page Tables, Walked Top-Down</i>	421
<i>Hierarchical Page Tables, Walked Bottom-Up</i>	422
7.4.5. <i>Page Table Designs for Extremely Large Memory Spaces</i>	423

<i>Inverted Page Tables</i>	424
<i>PTE Caches</i>	426
<i>Other Directions for Even Larger Memory Spaces</i>	427
<i>One Last Thought on Large-Memory Designs: Backup Is Needed</i>	427
7.4.6. <i>Hardware- vs. Software-Walked Page Tables</i>	428
7.5. File Systems	428
7.5.1. <i>General Implementation and Flow</i>	429
<i>Physical Layout of a Disk</i>	429
7.5.2. <i>Data as Cached in Volatile Memory</i>	433
7.5.3. <i>Future Merging with the Virtual Memory System</i>	435
7.6. This Particular Kernel Implementation — Osprey	437
7.6.1. <i>The General State-Machine Implementation and Flow</i>	437
7.6.2. <i>Segmented Organization of Virtual Memory and Files in Osprey</i>	440
<i>Perspective on the System's Design Space</i>	440
<i>The Paged Segmentation Model</i>	442
<i>The inode Structure — Mapping Files in the File System</i>	445
<i>The Process Control Block — Mapping Process Address Spaces</i>	446
7.7. Building the World's Simplest Kernel, in Four Stages	447
7.7.1. <i>The Microprocessor's Instruction Set</i>	447
<i>New Instructions</i>	450
<i>New Pseudo-Instructions</i>	450
<i>Large-Immediate Instructions</i>	451
7.7.2. <i>The Microprocessor's System-Level Details</i>	451
<i>Virtual Memory System</i>	451
<i>Interrupt Mechanism and Memory-Mapped Special Registers</i>	453
<i>Kernel Layout</i>	455
<i>System Boot Process</i>	455
▶ Project 7.1: Build a Context Save/Restore Facility	456
▶ Project 7.2: Build a Multitasking Scheduler	461
▶ Project 7.3: Build a System-Call Interface	469
▶ Project 7.4: Return Data from a System Call	478
7.8. The Design of the User Interface (The Shell)	481
▶ Project 7.5: Implement Process Invocation & the Shell	481
7.9. Security Issues Regarding the Kernel	483
7.9.1. <i>User IDs and Authentication</i>	483
7.9.2. <i>Interrupts and Privilege Escalation</i>	484
7.9.3. <i>Stack-Based System Calls</i>	485
7.9.4. <i>Stack-Based Buffer Exploits</i>	486
7.9.5. <i>Side-Channel Exploits</i>	486
8.High-Performance Processor Architectures	489
8.1. High Performance = Below 1 Cycle Per Instruction	489
8.2. An Example VLIW Microprocessor	490
8.2.1. <i>RiSC-32 VLIW Instruction Set</i>	490
<i>Pseudo Instructions</i>	494
<i>Word Addressing & Memory Access</i>	495
<i>Large Immediate Values</i>	495
<i>Example Assembly Code & Assembler Output</i>	496
8.2.2. <i>The Power of Parallelism</i>	497
8.2.3. <i>Endianness</i>	499
8.2.4. <i>Schematic Diagram</i>	500
8.2.5. <i>RiSC-32 Pipeline Registers</i>	503
8.2.6. <i>Pipeline Behavior</i>	505

Fetch Stage & PC Update (including Branch Prediction)	505
Decode Stage	506
Execute Stage	507
Writeback Stage	511
8.2.7. Control Modules	511
8.3. An Example Out-of-Order Microprocessor	513
8.3.1. Background	513
Tomasulo's Algorithm	514
Reorder Buffer	515
Register Update Unit	516
8.3.2. RISC-16 Out-of-Order Implementation: Overview	517
Fetch Phase	520
Enqueue Phase	520
Data-Incoming Phase (state = IQ_ARGS)	521
Ready to Issue Phase (state = IQ_IREQ)	521
Execute Phase (state = IQ_FUNC)	521
Commit Phase (state = IQ_DONE)	521
8.3.3. Example Execution of Meltdown Speculative-Execution Code	522
8.4. Multicore and Cache Coherence	531
8.4.1. General Cache Organization and Operation	531
8.4.2. Multilevel Caches	532
Shared Caches vs. Private Caches	535
What Cache Coherence Does	535
Cache Inclusion	537

Part III: Implications **543**

9. Some Failures of Security.....545

9.1. Privilege Escalation Should Not Happen in Systems	545
▶ Project 9.1: Implement Some Buffer-Overflow Attacks	548
9.2. Mitigation vs. Return-Oriented Programming (ROP)	565
▶ Project 9.2: Implement Some ROP Attacks	566
▶ Project 9.3: Implement a Meltdown Side-Channel Attack	574

10. Some Ways Forward **599**

10.1. We Need a New Definition for 'Architectural State'	599
10.2. Review of the Root Mechanisms that Enable Attacks	603
10.2.1. Buffer-Overflow Attacks	604
10.2.2. Spectre/Meltdown Attacks	604
10.3. The Case for Atomic Instruction Execution	608
10.3.1. Atomic Cache Access	609
10.3.2. Atomic Branch Prediction	609
10.3.3. Resulting Design	609
10.3.4. Atomic Everything Else	611
10.3.5. Sequestered Kernel	611
10.4. We Must Teach Hardware and Security Together	611

Appendix: A Formal Model for Analyzing Security Properties .613

Analyzing the Security Properties of Microprocessors	613
The Serial ISA Microarchitecture	614
The Out-of-Order Microarchitecture	615
Shadow State	616
Logic and Reasoning	617
The Logic of Security	617

<i>Modal Logic</i>	618
<i>Defining States, Transitions and Atomic Propositions</i>	619
<i>Kripke Frames</i>	619
<i>Kripke Model</i>	622
<i>Evaluating Modal Logic Formulas</i>	622
<i>Example Kripke Models Describing the Logic of Security</i>	623
<i>Hybrid Logic</i>	624
Hybrid Logic Analysis of the Out-of-Order Microarchitecture	625
Open Hardware Security Challenges	628
<i>Challenge Area One: Instruction-Based Data Structures</i>	629
<i>Challenge Area Two: Shared Resources</i>	629
<i>Challenge Area Three: Input/Output (I/O) Devices</i>	630

Appendix: The RiSC-16 Instruction-Set Architecture & System Interface631

The Instructions	631
What the Various Instructions Do	633
<i>Original Set</i>	633
<i>Additional Set</i>	634
<i>Large-Immediate Instructions</i>	635
The Register File	635
The Control Registers	637
The Start-Up Process	639
Hardware-Walked Page Tables	639

Appendix: A Brief Primer on Behavioral Verilog641

Basics of Behavioral Verilog	642
<i>Basic Program ('Module') Format</i>	643
<i>N-bit Busses/Registers/Memories</i>	643
<i>Register Assignment: blocking vs. non-blocking</i>	644
<i>Wire Assignment: 'continuous'</i>	645
<i>Subscripts</i>	645
<i>Control</i>	645
<i>If Statements and Case Statements</i>	645
<i>Ternary Operator</i>	646
<i>Concatenation, Replication</i>	646
Synthesis and A Tale of Two Solutions	647
<i>What Hath Verilog Wrought?</i>	649
<i>Synthesizable Verilog</i>	651
Hardware Design Steps	654

Preface



Historically, software engineers and hardware engineers are disparate groups that don't talk to each other. The result is that *computer design* is taught by one group (hardware people), and *computer security* is taught by another (software people). Because these two groups don't interact much, neither knows much of the other's discipline, and therefore *those they teach* learn next to nothing of the other's discipline. This is problematic because computer design and computer security are intimately dependent on each other, and when system designers fail to understand both topics, catastrophe results.

As we will show, two of the more critical security holes known today (buffer overruns and *Spectre/Meltdown*) exist only because system designers unwittingly undermined the very protections that would otherwise have guaranteed absolute insulation from abuse. In addition, because the topics of security and computer design are taught separately, students cannot see the root causes *behind* security failures ... this explains why many of the widely proposed and adopted solutions to buffer overruns and Spectre/Meltdown fail to address the root issues.

The division between hardware and software is seen in both education and the workplace. Software and hardware engineers are typically housed in different departments within different organizations, often located in completely different buildings. They don't typically see eye-to-eye. Software people consider hardware people to be uni-brow neanderthals who think that clubs, sharpened rocks, and C programming are paragons of high-tech (think the opening scene in Stanley Kubrick's *2001: A Space Odyssey*). Hardware people consider software people to be flighty, self-satisfied pansies who complain far out of proportion to what useful work they accomplish (think Harvey Ko-

rman and Andréas Voutsinas as the bickering French noblemen in Mel Brooks's *History of the World, Part I*).

The division in perspective leads to division in real system designs. Typically, hardware is designed with little regard to the software that will run on it, and software is almost always designed with little regard to the hardware it runs on. For instance, *portability* is considered a laudable achievement in the land of software — and it literally means the writing of software in such a way that it is *not* dependent *at all* upon the underlying hardware. Again, we will show how today's biggest security holes come from misalignment between one's hardware and software designs.

We hope that this textbook will help to address the problems that the division has created. We have found that it is a natural fit to teach computer organization/architecture alongside computer security; doing so leads to a natural understanding of the purpose behind hardware and operating-system primitives. We hope that treating the two domains together in the same semester-long course will put an end to the practice of designers unknowingly disabling critical security features for the sake of convenience.

This textbook presents the design and security of microprocessors and the operating systems that run on them. An example computer system is built from the ground up, so that all of the implicit assumptions are visible and identifiable. This course covers the rudiments of logic design, computer organization and architecture, compiler design, and operating systems design, so that the student can participate in the design and development of all of these components. Then it discusses security aspects of the design in depth, so that the student can see the ways in which current designs both succeed and fail to protect against intrusion.

The design projects are presented as incomplete solutions: they are our system-implementation code, with the 'intelligent' portions removed. The security projects are essentially guided tours through the various system behaviors, with the student implementing each of the attacks and learning how one can defend against them.

By the end, the student should have a solid understanding of how current systems are built, why they were designed in the ways that they were, what their weaknesses are, what current mitigation strategies are considered to be appropriate ... as well as some thoughts on how one can do better in future designs, both hardware and software. The result should be a relatively comprehensive view on current design and security principles of modern computing systems.

Good luck, and happy coding.
—The Authors, Annapolis