Fault Attack Resilience on Error-prone Devices

A study into the effects of error injection on micro-controllers and software security strategies to recognise and survive attacks

Martin S. Kelly

A thesis presented for the degree of
Doctor of Philosophy

Information Security Group
Royal Holloway, University of London

2022

Declaration

These doctoral studies were conducted under the supervision of Prof. Keith Mayes.

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in Royal Holloway's Information Security Group as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Martin S. Kelly

January, 2022

Acknowledgments

The work described in this report would not have been possible without the assistance of **John Walker**, ex of SiVenture Ltd. John's help in preparing samples, along with his infectious enthusiasm for sharing and passing on knowledge and skills, made the early lab-work educational and entertaining. In a similar vein, my thanks and gratitude are due to **Prof. Keith Mayes** who supervised this project. Keith's patience and support have been very much appreciated as work here progressed in fits and starts while being interleaved with full-time employment. The third individual worthy of special thanks is **Julian Brown** of Nexus Electronics, Cambridge. Julian's suggestions and corrections for some of the circuit boards developed for this study saved countless hours and helped avoid expensive revisions.

All three of the above have my eternal thanks and gratitude. Without them, this work would not have been completed.

Additional thanks are also due to **Prof. Konstantinos Markantonakis** and to **Dr. Raja Akram** for their ad-hoc advice, encouragement and recommendations, as this project progressed.

This list would be incomplete without expressing my gratitude to **Dr. David Everett**, ex of NatWest Development Team, and **Dato' "Tony" Lee Kwee Hiang**, founder of Iris Technologies Sdn. Bhd., both of whom gave me free rein to cause electronic trouble with chips while working on groundbreaking smartcard initiatives. Their encouragement and trust led to a career in implementing defensive code that ultimately resulted in this study.

All of the above have been friends as well as colleagues throughout this project, and I hope, at some point, to be able to return the favours.

Thank you all!

Acknowledgements and thanks are also due to the following organizations for the use of their images and code. The images' copyrights remain the property of the organizations and individuals listed below.

- Figure 2-9 — **Nanoscope Services Ltd.**, Bristol, BS15 4PJ.
- Figures 2-5 & 2-6 — draw heavily on Roberto Avanzi's original work in "TikZ for Cryptographers", `https://www.iacr.org/authors/tikz/`.
- AVR Speck implementation by LuoPeng [10], used in Section 4.3.
- Digital Locksmiths Ltd., Cambridge. For access to a back catalogue of commercially developed and security certified smartcard applications, funding these studies, and paying for all of the hardware developed during this investigation.

Abstract

This thesis demonstrates a new practical approach to understanding a micro-controller's behaviour when subjected to error inducing attacks. It also shows a novel mechanism for understanding the effects of errors and the efficacy of counter-measures. The insights gained enabled the development and evaluation of a new C compiler capable of inserting effective counter-measures that could not otherwise be realised via off-the-shelf tools.

While conducting this research, we identified properties of the equipment used to induce errors that enabled us to construct a new, very flexible, low-cost error injection workstation. The new tools provide a framework for accurately injecting perturbation errors and for retrieving the resulting device state. This demonstrates the ease with which an adversary can attack a target and provides the ability to self-test one's defences.

The findings of this study have particular relevance in the field of general-purpose micro-controllers. These devices are playing an ever-increasing role in everyday life, for example, home automation gadgets in the Internet-of-Things. The consequence of this increased diversity of application is that products are often specified and commissioned without considering the vulnerabilities of stand-alone micro-controllers. Similarly, the development and programming tasks are often delegated to engineers who are unfamiliar with the coding disciplines required to resist attack.

This study shows that the tools and techniques required to protect such devices can be made readily available and are not the sole preserve of well-funded laboratories or big corporations.

Contents

Page

Cover **1**

Declaration . . . 1

Acknowledgements . . . 3

Abstract . . . 5

Contents **7**

List of Contents . . . 7

List of Tables . . . 10

List of Figures . . . 12

List of Abbreviations . . . 17

Forward **25**

1 Introduction **27**

1.1 Research Questions . . . 30

1.2 Methodology . . . 33

1.3 Significance . . . 34

1.4 Structure of this Thesis . . . 37

2 Background **39**

2.1 Threats . . . 41

2.2 Attacks . . . 48

2.3 Errors . . . 68

2.4 Defence Techniques . . . 88

2.5 Observations . . . 91

3 Categorising Errors **93**

3.1 Fault Models . . . 96

3.2 Test Strategy . . . 99
3.3 Experiments . . . 106
3.4 Data and Interpretation . . . 136
3.5 Summary . . . 136

4 A New Laser Workstation 139
4.1 Components . . . 141
4.2 Multi-Pulse Proof of Capability . . . 152
4.3 Creeping Barrage - Blind Attack on Known Code . . . 158
4.4 Summary . . . 161

5 Testing Security Defences 163
5.1 Practicalities . . . 166
5.2 Defences . . . 169
5.3 Results and Analysis . . . 181
5.4 Application . . . 188
5.5 Summary . . . 190

6 Automating Defence Generation 193
6.1 Background . . . 197
6.2 Defensive C Compiler . . . 201
6.3 Defending Execution Path . . . 203
6.4 Code Efficiency . . . 224
6.5 Summary . . . 230

7 Security Impact 233
7.1 Implications . . . 235
7.2 Strategies . . . 244

8 Conclusions and Further Work 247
8.1 Original Goals . . . 250
8.2 Future Research Directions . . . 258
8.3 The Last Word . . . 261

Bibliography 263

A Test Harness 279

A.1 Components 280
A.2 Roles and Responsibilities 282

B Test Circuit Boards 287

B.1 Purpose 288
B.2 Board 1 288
B.3 Board 2 291
B.4 Board FPGA 294

C Defensive C-Compiler 301

C.1 Operation 303
C.2 Samples 304

List of Tables

2.1 Key Nulling Exploit 74

3.1 ATTiny841 Features 102
3.2 No Operation Results 120
3.3 Inc/DecNo Operation Results 121
3.4 Arithmetic Operation Results 122
3.5 Observed laser induced errors on load/store operations 123
3.6 Conditional branch 124
3.7 Laser induced errors 125
3.8 Branch Test Results 129
3.9 Branch Test Results, fixed location, Third Q-cycle 130
3.10 Errors by Power and Aperture 135

4.1 Test Pulses, Permutations and Time. $(n = 32, at\, 4\, Samples\, s^{-1})$ . . . 156
4.2 Jump Matrix Termination States 157
4.3 Speck Round Attack 160

5.1 Test Roles 182
5.2 Test Samples 184
5.3 Hybrid Defence Test 190

6.1 Triple Values 205
6.2 Call/Return - Single Pulse Attack 213
6.3 Call/Return - Double Pulse Attack 215
6.4 Conditional Branches - Single Pulse Attack 219
6.5 Conditional Branches - Double Pulse Attack 221
6.6 Macro Defended Conditionals 223
6.7 Code Volume 225
6.8 Bootstrap - Code Compilation 229

6.9 Banking MAC - Code Compilation 229

List of Figures

2-1 CMOS Inverter . . . 58
2-2 Power Measurement . . . 59
2-3 Typical Power Waveform . . . 60
2-4 Crude Exponentiation . . . 61
2-5 DES Round 1 . . . 63
2-6 DES S-Box Structure . . . 64
2-7 DPA Accumulated Trace . . . 64
2-8 ePassport Chip . . . 66
2-9 Chip Probing . . . 66
2-10 DES Final Rounds . . . 72
2-11 Java Source Code . . . 75
2-12 Java bytecode . . . 75
2-13 Body Bias Transistor . . . 79
2-14 Semiconductor Band Structure . . . 81
2-15 Laser workstation . . . 83
2-16 Frontside and Rearside Attack . . . 84
2-17 Etched Package . . . 84
2-18 Rebonded IC . . . 85
2-19 Static RAM, Single Bit . . . 86
2-20 Simplified SRAM Layout . . . 86
2-21 AVR Registers . . . 87
2-22 Silicon on Insulator . . . 89

3-1 ATTiny851 Pin-Out . . . 102
3-2 Nd:YAG Laser Workstation . . . 105
3-3 Basic Test Program . . . 108
3-4 DUT Support, Board 1 . . . 110
3-5 Board 1 Control Logic . . . 111

3-6 Test Campaign Configuration 113
3-7 Test Program 1 114
3-8 Error Distribution Plots 115
3-9 Lock-up Error 115
3-10 DUT support, Board 2 117
3-11 Board 2 Control Logic 117
3-12 Instruction Test 118
3-13 Laser Firing 119
3-14 Error Locations 125
3-15 Error Sensitivity Map 127
3-16 Branch Test Code 128
3-17 Pre-Fetch Test 131
3-18 Signal Timing 132
3-19 Power and Aperture Test 133

4-1 Laser Diode with Heatsink and Lens 142
4-2 Laser Station 143
4-3 Trigger Signal Propagation 145
4-4 Laser Control After Signal Propagation 145
4-5 Laser Test Rig 146
4-6 Regular Pulse 147
4-7 Laser Tripple Pulse 148
4-8 Alignment Basic 148
4-9 Alignment Parallax 149
4-10 Alignment Offset 149
4-11 Workstation 150
4-12 Branch Tests 152
4-13 Branch Test Execution Paths 153
4-14 Branch Test Execution Paths 155
4-15 Speck Encryption 159

4-16 Speck Test Control 160

5-1 Execution States 168
5-2 Unprotected 171
5-3 Double Test 171
5-4 Retest at Destination 172
5-5 Inverse Test 173
5-6 Double Data 174
5-7 Data Inverse 174
5-8 Checksum Data 175
5-9 Redundant Representation 176
5-10 Repeat Calculation 176
5-11 Modified Compensated 177
5-12 Alternative Algorithm 178
5-13 Inverse Calculation 178
5-14 Jump Id 179
5-15 Waymark, Late Test 180
5-16 Waymark, On the Fly 180
5-17 Termination States 185
5-18 Disassembled Double Test 187
5-19 Disassembled Inverse Test 187
5-20 Hybrid Defence 189
5-21 Hybrid Defence Termination States 190

6-1 Simple Double Test Source 195
6-2 Accidental Optimised Output 195
6-3 Optimised Output 196
6-4 Unoptimised Output 197
6-5 Compilation Stages 202
6-6 Test Structure 204
6-7 Triple Update 204

6-8 Defended Call 207
6-9 Defended Entry/Exit 208
6-10 Call/Return Test Program 209
6-11 Possible Paths with Two Skipped Instructions 211
6-12 Defended Branch 217
6-13 Branching Test Program 218
6-14 Double Checking Macros 222

7-1 Equipment Availability 236
7-2 Device Vulnerability 240
7-3 Development Skills 242

A-1 Test Rig Schematic 280
A-2 Sample Alignment 282
A-3 Zone Identification 283
A-4 Campaign Parameters 284
A-5 Test Script 285
A-6 Campaign Progress 285

B-1 Board 1 Populated 289
B-2 Board 1 Schematic 290
B-3 Board 2 Populated 292
B-4 Board 2 Schematic 293
B-5 Board FPGA 295
B-6 Board FPGA Rear 295
B-7 Board FPGA Power Schematic 296
B-8 Board FPGA DUT 297
B-9 Board FPGA Connections 298
B-10 Simulation 299

C-1 DCC Project Directory Structure. 302
C-2 Call & Return — Source Code 304

C-3 Call & Return — Undefended Output 305
C-4 Call & Return — Defended Output 306
C-5 Branching Test — Source Code 307
C-6 Branching Test — Undefended Output 308
C-7 Branching Test — Defended Output 309
C-8 Variable Declaration — Source Code 310
C-9 Variable Declaration — Output 310

List of Abbreviations

μC - Micro-Controller.

A small computer on a single integrated circuit chip. 29–32, 34, 35, 40–43, 45–49, 53–55, 58, 60, 62, 63, 76, 78, 80, 87–92, 95–97, 99–101, 103, 106, 110, 111, 115, 116, 118–120, 126, 127, 130, 137, 146, 152, 162, 164–167, 174, 183, 192, 203, 204, 207, 225, 231, 238, 242–245, 249, 252, 254, 256

AES - Advanced Encryption Standard.

A standardized symmetric encryption algorithm designed as a replacement for DES. 73, 74, 80, 97

APDU - Application Protocol Data Unit.

The communication packets exchanged between a smartcard reader and a smart card. 175

CA - Certification Authority.

The issuer of digital certificates.

Digital certificates attest to the relationship between a public key, the named subject associated with the key. 52

CC - Common Criteria.

International standard for information technology security evaluation. . . 95

CCD - Charge-Coupled Device.

Integrated circuit image sensor used primarily in digital cameras. . 144, 149, 258, 281, 282

CD - Compact Disk.

60 mm radius optically readable digital data storage device. 147

CFI - Control Flow Integrity.

Ensuring program flow control transfers are deliberatly instigated by the executing program and not a consequence of faulty execution. 199, 203, 207, 230, 231, 243, 245, 257, 260

CMOS - Complementary metal–oxide–semiconductor.

The principle technology used in integrated circuit manufacture. 79

CPA - Correlation Power Analysis.

An enhancement on DPA involving a model of a device's power consumption to augment trace categorization process. 64, 65

CPU - Central Processing Unit.

The main processor responsible for executing program code. 98, 111, 116, 132, 157, 165, 190, 191, 209, 251, 252, 260, 289, 302

CRT - Chinese Remainder Theorem.

A theorem which gives a unique solution to simultaneous linear congruences with coprime moduli. It is of particular interest when applied to RSA as it significantly reduces computer execution time when performing exponentiation to large moduli. 70, 95

CSV - Comma-Separated Value.

An ASCII text file in which each record occupies a single line and each field in the record is separated by a comma (or similar punctuation). 112

DCC - Defensive C Compiler.

The experimental code generator written for this investigation. 201, 203, 210, 212, 216–218, 222, 224–228, 230–232, 256, 257, 302

DDoS - Distributed Denial of Service.

An synchronised attack, coming from multiple sources, with the intention of making the target unusable or unobtainable. 47

DES - Data Encryption Standard.

A widely deployed standardised symmetric encryption algorithm. It has been superseded by AES. 63, 73, 227

DFA - Differential Fault Analysis.

A cryptographic attack performed by comparing the erroneous output from a cryptographic process. 69, 74, 80, 87, 158

DIL - dual in-line package.

Chip packaging that uses two parallel rows of pins with 2.54 mm pin to pin separation (0.1 inch). 107

DPA - Differential Power Analysis.

Statistical analysis of side-channel leakage relating the power consumption of a device. 62, 65, 192

DUT - Device under test.

The sample being investigated or attacked. 31, 33, 34, 37, 38, 50, 55, 59, 97, 99, 103–116, 119, 132, 133, 137, 138, 141–146, 150, 151, 158, 159, 165, 166, 169, 192, 226, 234, 238, 244, 250, 251, 253, 254, 256, 257, 259, 280–284, 287–289, 291, 294, 299

EM - Electro Magnetic Radiation.

Waves in the electro-magnetic field propagating through space. . . . 54, 238

EMV - Europay Mastercard Visa.

The industry standard for bank cards and payment terminals. 74, 95, 106, 169

FIB - Focused Ion Beam.

A device, similar in operation to an electron microscope, that uses a focused beam of ions for deposition or ablation of material. 67, 88

FPGA - Field-Programmable Gate Array.

Configurable integrated circuit that is customisable in-circuit. . 146, 150, 294, 299

GCC - the GNU Compiler Collection.

Open source compiler tools originally written as the compiler for the GNU operating system. . 172, 173, 192, 198, 202, 210, 217, 222, 224–228, 232, 256, 257

GIGO - Garbage in, Garbage Out.

Unexpected results derived from calculations on unexpected data. 29

GPIO - General Purpose Input Output.

Uncommitted digital signal pins on a μC that may be configured to interact with external peripherals. 99, 102, 103

HDL - Hardware Description Language.

A computer language used to describe the structure and behaviour of electronic circuits. E.g. Verilog and VHDL. 249

IC - Integrated Circuit.

Otherwise known as a silicon chip. 44, 50, 58, 59, 62, 67, 68, 74, 77, 79–89, 97, 99, 100, 107, 112, 125–127, 134, 249

IoT - Internet of Things.

A network of physical devices with embedded μCs capable of exchanging data. 35, 46–48, 99, 101, 137, 243, 261

IR - Intermediate Representation.

Internal data structures created by a compiler after parsing the source code.. 200–202, 260, 261, 302, 303

ISA - Instruction Set Architecture.

An abstract definition of instruction behaviour that does not depend on the characteristics of the implementation. 165, 190, 236, 249, 251

ISR - Interrupt Service Routine.

Code invoked in response to an interrupt signal. 131, 132

JTAG - Joint (European) Test Access Group.

A hardware debugging interface to electronic components. 107, 109

LIDAR - Laser Imaging, Detection, and Ranging.

Range determination using time-of-flight of reflected laser pulses. . . 144, 253

LLVM - The LLVM Compiler Infrastructure Project.

The name "LLVM" itself is not an acronym; it is the full name of the project. 256, 257

LSI - Large Scale Integration.

Whole circuits, or ICs, on silicon as opposed to individual components. . . 97

LVDS - Low-voltage differential signaling.

Data signalling by transmiting information as the voltage difference between two adjacent wires. 146, 299

MAC - Message Authentication Code.

A cryptographic checksum used to confirm the integrity, and sometimes also the origin, of a message. 45, 227, 228

MitM - Man in the Middle.

An agent who is invisibly placed between both parties in a conversation who has the opportunity to read and modify messages before forwarding them to the intended recipient. 45

NDA Non-Disclosure Agreement.

A legally binding contract defining a confidential relationship whereby the signing parties agree that sensitive information they share will not be made passed on third parties without the original owner's consent 99, 100, 225

NIR - Near Infra Red.

The electromagnetic spectrum between 780 nm and 2500 nm. 83, 84, 87, 101, 133, 143, 258

OBIC - Optical Beam Induced Current.

A laser injection technique to induce current flow within a semi-conductor. 81, 82

PCB - Printed Circuit Board.

A non-conductive material providing support for electronic components and providing inter-connection between those components via conductive tracks printed or etched on the board material. 104, 107, 109, 291, 294

PKI - Public Key Infrastructure.

The set policies, processes and procedures using asymmetric cryptographic algorithms to manage, distribute, and verify digital certificates and public keys. 52

ROP - Return Oriented Programming.

Deliberate corruption of the call stack to invoke code, typically libraries, already resident in the machine. 198

RSA - Rivest Shamir Adleman.

A public key cryptographic algorithm used for encryption and digital signature. 69, 70

SER - Soft Error Rate.

The probabillity of a Single Event Upset (SEU) occurring. 68, 69

SEU - Single Event Upset.

A transient error in a silicon device caused by and external ionizing event. 68, 69, 77, 235, 241

SMID - Single Instruction Multiple Data.

Parralel operation of the same instruction on multiple data points. . . . 100

SOIC - Small outline integrated circuit.

A surface-mounted integrated circuit package with pin separation of 1.27 mm (0.05 inch). 102

SPA - Simple Power Analysis.

Analysis of a devices power consumption to infer internal behaviour. . 60, 62, 91, 137, 192

TTL - Transistor-Transistor Logic.

Defines the threshold voltages for digital logic implemented with 3.3 V or 5 V devices. 144

USB - Universal Serial Bus.

A definition for cables, protocols and power, standardizing the connection of peripherals and computers. 146, 253

VHDL - VHSIC Hardware Description Language.

Language to model the behavior of digital systems. 192

VHSIC - Very High-Speed Integrated Circuits Program.

A United States Department of Defense research program to develop very high speed integrated circuits. 192

VLSI - Very Large-Scale Integration.

The process of integrating hundreds of thousands of transistors on a single microchip. 249

VM - Virtual Machine.

Software emulation of a machine or computer. 76

WORM - Self replicating computer program.

A program that propagates through a network by duplicating itself and executing on neighbouring machines. 47

YAG - neodymium-doped yttrium aluminium garnet Nd:YAG.

A crystal used as the laser medium for medium power solid-state lasers. 105, 134, 138, 141, 142, 144, 145, 150, 151, 161, 192, 252, 259, 288, 291, 294

Forward

In 1997 I found myself debugging some curious production-line failures. The world's first ePassport was due to be launched in Malaysia, and while production of the inlays was gearing up, the unit failure rate became alarmingly high. It was my code and, therefore, my problem. Each unit consisted of two integrated circuits and, when tested after assembly, some responded with unexpected data. The issue was traced back to the bonding machine supervisor, who, worried about production reliability, frequently inspected the alignment of the wire bonding machine. High failure counts coincided with these inspections. The inspection light was the root cause of the issue. The chip's internal program was corrupting its memory because of this light. The immediate solution was to perform electrical tests in the dark; the long term solution was to modify the internal program to survive execution errors. This fix was not a defence against software bugs or unanticipated data crashing an application. It was a defence against the CPU itself not reliably running its program. This was my first exposure to *Defensive Programming*, and at this time, we considered errors to be a nuisance rather than a threat.

Two years later, in 1999, while working on MyKad, Malaysia's project for an electronic national I.D. card, we saw the software partner responsible for the e-Purse element attacking our code with a camera flash-gun. Flashes of light were being used to induce errors deliberately. The defensive code behaviour needed to be extended to cover all life-cycle states of the application. Defensive programming was needed to protect assets, not just to improve production line efficiency.

It was not until 2002 that "Optical Fault Induction Attacks" were first publicly described. Earlier theoretical papers on the consequences of errors could now be combined with a reliable mechanism to induce them. What had been an open secret in the industry was now public knowledge. Every smartcard application from here on would need to be defensively coded and tested for fault resilience.

Over the following years, it became apparent that testing fault-tolerance was a poorly understood art. I was responsible for getting multiple smartcard applica-

tions evaluated for security certification, and evaluators for ePassport, EMV, and JavaCard/Global-Platform sometimes had conflicting views regarding defensive code strategies. It appeared that the error effects, and consequently defences, were driven by folk-lore and tradition. Apparently, nobody knew the best approach. This uncertainty was expensive for me as a developer. One application, in particular, was problematic. A single ROM-Mask, implementing the EMV payment card protocols, was intended to be configurable to be either MasterCard or Visa as a production line option.

The security evaluators assessing the MasterCard related code behaviour insisted on a particular style of defence. The assessors of the Visa components insisted on another. It was impossible to satisfy both at the same time.

This study was born out of a desire to shed light on that confusion, improve the effectiveness of defensive code, and simplify its deployment.

Chapter 1

Introduction

Contents

1.1 Research Questions **30**

1.1.1 Nature of Errors 31

1.1.2 Practicalities of Error Induction 31

1.1.3 Defences 32

1.1.4 Deployment 32

1.2 Methodology **33**

1.3 Significance **34**

1.3.1 Publications 36

1.4 Structure of this Thesis **37**

Errors in software, and computers in general, are ubiquitous. Human error, manufacturing defects and misunderstood problems introduce uncertainty into what is widely perceived to be prescriptive or predictable behaviour. These errors can take multiple forms and can have serious repercussions; they may be the consequences of poor program design, implementation bugs, or, most dangerously, being deliberately induced by an attacker.

- First of all, there is the simple case of inaccurate coding. Here typographical mistakes by the programmer lead to erroneous results. Sometimes these errors can go undetected through many rounds of debugging and testing but still have catastrophic effects, for example, NASA's Mariner 1 spacecraft [122]. Here a missing hyphen [123] lead to the destruction of the launch vehicle and its payload.

- There is flawed logic or faulty algorithms where accurate implementation still yields erroneous results because of a flawed recipe. Such bugs may take years to manifest themselves as problems. A pertinent example would be the Julian calendar's algorithm for inserting leap years and its subsequent replacement with the Gregorian calendar one and a half millennia later [60]. Similar rounding errors were responsible for an accumulated 50% inaccuracy in the Vancouver Stock Exchange's index of leading shares [142], and for the failure of a missile defence system [189] resulting in the loss of 28 lives.

- Another source of errors is faulty input data. There are two categories of this type of error;

  - Unexpected structure of data may cause internal buffers to overflow and corrupt neighbouring data. In extreme examples, the effect can be exploited to run unauthorised software on a computer [11]. These bugs arise when the data receiver fails to check the size and syntax of the incoming data.

  - Well-formed but erroneous data may also be presented to a computer.

Commonly referred to a Garbage In Garbage Out (GIGO) errors, they can have equally devastating effects; most famously when metric and imperial units were transposed in navigation software on NASA's Mars Climate Orbiter [35].

- Broken or faulty hardware can prevent a computer from performing correctly. Such errors usually prevent the use of the device until the issue has been physically identified and fixed. A dead moth obstructing a mechanical relay in an early computer is credited as being the original computer bug and its removal the origin of the term *debugging* [161].

- Finally, we have transient effects. Here bug-free code, processing well-formed data and executing on an undamaged processor, may momentarily be influenced to perform a faulty operation before resuming normal processing behaviour. The effects of this transient error may then propagate in much the same way GIGO errors do. These transient effects may be attributed to 'acts of god' or cosmic rays [26], or may have been deliberately induced by an attacker [75].

Deliberate induction of errors in an otherwise healthy Micro-Controller (μC) is the focus of this thesis. Commonly referred to as perturbation errors or *Glitch Attacks*, the adversary attempts to make the chip perform unauthorised functions or disclose secret information. The mechanisms available to perform glitch attacks are diverse, and the consequences can be significant. These are discussed in detail in Chapter 2.

Error induction is perceived as challenging to perform and leads to a degree of complacency, with many developers believing the obstacles to an attack provide sufficient defence for their products. This remains a valid argument when the attack's cost significantly outweighs the value of the information gained. However, a close eye needs to be kept on the evolving repertoire of attacks and their implementation costs.

Where the value of the information within a device is high, the device itself must take steps to recognise the attack and take defensive action. This is commonly referred to as *Defensive Programming*. Unfortunately, defensive code comes with penalties in

terms of performance degradation, additional code volume and additional development effort. It has led many developers to focus on critical functions within the system as they juggle the competing drivers of security, performance and time to market.

The design of efficient defences requires an understanding of the properties of the induced errors. Personal experience of the EMV code-review and penetration testing process [66] suggests that much folk-law, instinct and wishful thinking exists. Contradictory advice from different testing laboratories relating to the same piece of code was the trigger that initiated this body of research.

1.1 Research Questions

In this study, we address four basic questions about device security. In particular, we look at the implications for devices deployed in the low-cost consumer electronics field, where users increasingly and unwittingly place their trust in an ever-expanding array of intelligent devices.

RQ1 — Do induced errors have repeatable characteristics that would assist developers in predicting a device's likely modes of failure?

RQ2 — Is it practical for attackers to induce multiple errors into software executing on a μC and exploit their effects without needing access to sophisticated laboratory equipment?

RQ3 — Can a better understanding of a device's modes of failure be translated into improved security via targeted software countermeasures?

RQ4 — Is it practical to automate the generation of defensive measures within a μC's software development tools?

1.1.1 Nature of Errors

RQ1 — *Do induced errors have repeatable characteristics that would assist developers in predicting a device's likely modes of failure?*

This is a simple question in principle, but several complicating factors need to be addressed. A representative target must be identified that is unencumbered by preexisting defences that will skew observed behaviour or result in the device's destruction. We must also find a viable mechanism for capturing the internal state of the Device Under Test (DUT) immediately after the induction of an error. The intention is to do this using a normally executing μC, removing the potential effects debug mode may have and making the attack scenario as similar as possible to typical deployment. Finally, many samples and experiments will be required. The process will involve creating many test scenarios and repeating these tests while directing the attack onto different physical locations on the surface of a μC. Automation of the data collection process will be essential, and tools will need to be built to do this.

1.1.2 Practicalities of Error Induction

RQ2 — *Is it practical for attackers to induce multiple errors into software executing on a μC and exploit their effects without needing access to sophisticated laboratory equipment?*

The repertoire of attack techniques is wide, as discussed below in Section 2.2, and some of them require very expensive, difficult to access equipment. Some other attacks have been demonstrated in laboratory settings, but it may be possible to replicate them with modest resources and without privileged access to laboratory services. Reducing the cost and logistical difficulties associated with attacking a device has implications for a wide range of devices that have hitherto been assumed to be not worth attacking. Consequently, the software they run will need to implement additional defences.

1.1.3 Defences

RQ3 — *Can a better understanding of a devices' modes of failure be translated into improved security via targeted software countermeasures?*

The deployment of defences is always a compromise between application efficiency and security. Speed, code size and development effort compete with the ability to resist a range of attacks. The tools and techniques used to characterise glitch errors will also be suitable for evaluating the efficacy of defences. The expectation is that the knowledge gained about the glitch errors' nature will assist the definition of optimal defences. This will be achieved by identifying techniques to recognise errors and assessing their impact.

Removing the guesswork from the application of defences addresses the initial motivation for this study.

1.1.4 Deployment

RQ4 — *Is it practical to automate the generation of defensive measures within a μC's software development tools?*

It is already understood that some defences are complicated, intricate to deploy, and difficult to test. This factor makes it unlikely that the required knowledge and skills will be available in most development teams, particularly in the less security conscious but potentially vulnerable consumer products market. Ideally, it will be possible to encapsulate the required knowledge into development tools to include appropriate defences automatically. Such a tool will improve the security of simple products without the need to educate and retrain developers. For high-security products and skilled developers, it will complement the programmers' skills, reduce the opportunity for defence omission by human error, and simplify the slow and expensive process of independent code review.

Turning the laboratory results into a practical tool completes this mission.

1.2 Methodology

To a large extent, the strategy used to answer the research questions was dictated by those same questions.

An initial investigation was carried out to identify prior art, relevant tools, and operating parameters. In particular, reports of attempts to categorise errors and measure the efficacy of defences would be sought out. This review would ensure the relevance of the proposed research questions and identify the appropriate equipment to use as the starting point for this investigation. This equipment would need to enable controlled generation of error stimuli, need minimal manual input and be capable of running uninterrupted for long periods.

The first requirement was to create a fault induction system that could be used to evaluate the error responses from a DUT. This step involved removing unrelated obstacles that complicate data collection without affecting those data. These obstacles add complications to the implementation of an attack, and each could be considered, to a degree, a defence in its own right, adding cost to an attack's implementation. These obstacles are unrelated to the effects this study targets, so, where possible, their elimination is required to simplify access to the features of interest. This simplification would be achieved by developing bespoke circuit boards to control the environment the DUT operates in and by careful choice of DUT.

The follow on requirement was to collect data from the DUT while subjecting it to attack with the test rig. This blitz of error induction would be time-consuming, as many processor instructions would need to be exhaustively attacked and the results collected. The variable parameters included the physical position of the laser focus on the DUT's surface, the timing of the attack relative to the execution of an instruction, as well as the power and size of the laser pulse. The data collected through this process would then need to be analysed to identify characteristics that could be exploited. Bespoke tools would need to be written to coordinate experiments, collect data and analyse them.

The techniques for characterising error responses also apply to testing the effi-

cacy of programmed defences. Other researchers have attempted defence testing via simulation, but we would be able to obtain real-world results with the envisaged equipment. This equipment would enable a relatively straightforward evaluation of defences and would also answer the first of the research questions (RQ1).

The properties of the induced errors, and the complications observed while collecting them, would lead to optimisations in the attack mechanism and the development of a low-cost, more capable replacement for the test rig. This improved test rig would answer the second research question (RQ2).

Searching for and recognising identifiable characteristics in the DUT's error induced behaviour should also suggest new and more robust defences, specifically optimised for the DUT; this would provide the answer to the third of the research questions (RQ3).

Finally, after identifying demonstrably strong defence techniques, the viability of automatic generation of defensive code would be investigated. The intention would be to extend the behaviour of a compiler by applying appropriate defences during its object code generation stage. The efficacy of this code generator would be tested with the tools and techniques developed previously. This tool would answer the last of the original research questions (RQ4).

1.3 Significance

There is no doubt that error-free computation is critical to the security of devices such as smartcards. Errors are not just an inconvenience that crashes a device or causes embarrassment by failing to permit an authorised action. Errors can be exploited. As far back as 1997, Boneh et al. [37], and Biham et al. [32], demonstrated how to recover keys from encrypted data if an arithmetic error occurs during the encryption process.

Errors are easy to induce. All μC's come with a datasheet that explains the safe working ranges for properties such as supply voltage, signal frequencies and, operating temperature. Exceed these limits, and errors will occur. Also, by controlling when

and how long these limits are exceeded, an attacker can localise errors to specific parts of an algorithm. Very fine control of error injection in terms of physical location on a chip and the moment of an attack can be achieved with a focused laser beam. This technique has been in the public domain since Skorobogatov and Anderson [165] published details in 2002.

Defences against errors can be added to the μC's hardware. Sensors to detect input signals that deviate from the specification and sensors for light are relatively common in security chips. In some μCs, duplicate CPUs simultaneously performing identical actions can detect errors by continuously comparing the cores' states. This level of defence is reserved for the highest security devices such as Infineon's SLE78 family [88].

For many μCs designed for consumer electronics, and for the burgeoning Internet-of-Things (IoT) market, such hardware security augmentation is rare or non-existent. Instead, the programmer needs to write self-checking code, which involves anticipating attacks and coding defences that detect abnormal behaviour within a program. These coding techniques require a skill set that is not common amongst those tasked with developing software for consumer electronic devices. However, as is detailed by Desai [61], such devices are increasingly being trusted with personal information and have access to private networks.

Defences have a cost of their own. Development time increases, execution speed degrades, and code volume expands. Redundant code fragments that cannot be exercised under normal operating conditions further complicate the testing process.

In the following chapters, we describe a technique that we developed to identify the effects of induced errors. Understanding these effects is a critical first step in defining efficient software defences. Furthermore, our discoveries about the nature of the errors indicated that this process did not require particular specialist equipment. We went on to develop a low-cost alternative error injector with superior capabilities. This equipment was then used to measure the efficacy of a range of defensive code structures, an exercise previously only seen in simulation, for example, by Theissing [173].

Knowledge of the nature of errors and the best ways to defend against them is of great value if it can be deployed in fields where such techniques are frequently neglected. We developed a C-compiler that automatically inserts defensive code to demonstrate that this is possible. This tool has multiple and wide-ranging advantages.

1. It makes the techniques available to a wider audience without the need to re-train an army of developers.

2. It ensures systemic coverage of defences. Often defences are only placed in code that the programmer perceives as vulnerable, overlooking equally vulnerable support functions.

3. Defences that cannot be described in the syntax of the source-code language can be inserted.

4. Even for high-security applications where specifically skilled programmers are deployed, this tool can remove the opportunity for the accidental omission of defences. It is expected that this would also reduce the time required to review and certify code that is destined for scheme accreditation [66] or Common Criteria certification [53]. The focus of code reviewing can shift to the characterisation of errors and efficacy of defences. This process is more straightforward and less error-prone than searching an application's source code for overlooked opportunities to add defences or where defences are incorrectly implemented.

1.3.1 Publications

The following publications have been produced during the course of this investigation. Each paper covers the major phases of this investigation.

1. **Characterising a cpu fault attack model via run-time data analysis.** Published 2017 - IEEE International Symposium on Hardware Oriented Security and Trust (HOST).
   doi: `http://dx.doi.org/10.1109/HST.2017.7951802`

2. **High precision laser fault injection using low-cost components.**
   Published 2020 - IEEE International Symposium on Hardware Oriented Security and Trust (HOST).
   doi: `http://dx.doi.org/10.1109/HOST45689.2020.9300265`

In addition, the experimental defensive compiler developed as part of this study can be found in a public repository: `https://github.com/digitallocksmiths/DCC`.

1.4 Structure of this Thesis

Chapter 2 describes the background to the issues addressed in this study. It provides a history of the evolution of attacks targeting execution errors and the mechanisms by which such errors can be induced. It describes the threats posed by such errors and the arsenal of countermeasures available. It describes the practical issues encountered when attempting to induce erroneous execution and the choices available to a would-be attacker.

Chapter 3 describes the technique we devised for determining the effects of induced errors. It describes the collection data relating to the internal state of the DUT and the categorisation of the effects of the errors. Finally, it draws conclusions that guide the development of new test equipment.

The development of a new laser fault injection workstation is described in Chapter 4. This chapter covers the components used, the new device's capabilities, and the engineering obstacles overcome during its development. It ends with experimental results showing that it is more capable than the commercially obtained device it replaces.

Chapter 5 shows how we used the new laser workstation to evaluate a range of software defences. We demonstrate its use to quantify the efficacy of defences and measure the cost of these defences in terms of performance penalties and code volume. The ability of the fault injector to automate data collection was invaluable. It greatly simplified testing of the output of a defensive code generator in the form of a new C-compiler, which forms the subject matter for Chapter 6.

The impact of a more realistic perception of vulnerabilities and the improved opportunities for defence deployment is considered in Chapter 7.

The conclusions drawn from this study and further work opportunities are presented in Chapter 8.

Finally, the appendices describe the significant components developed during this investigation. Most of the work presented in this section relates to engineering and development; however, the functional objectives were guided by the research requirements. These appendices demonstrate the capabilities of the tools and their modes of use. They are primarily of interest to anyone intending to build upon the results presented here. Appendix A describes the test harness that coordinates the microscope, laser, test sample and data collection. Appendix B covers the circuit boards developed to support the DUT and synchronise the laser pulses with the executing test programs. Appendix C covers the features of the new Defensive C-compiler, management of its source code, its output and how to use it or modify it.

Chapter 2

Background

Contents

2.1 Threats **41**

2.1.1 Motivation 43

2.1.2 Mitigation 44

2.1.3 Relevance 45

2.2 Attacks **48**

2.2.1 Logical Attacks 50

2.2.2 Side-Channel Attacks 53

2.2.3 Physical Attacks 66

2.3 Errors **68**

2.3.1 History of Fault Attacks Techniques 68

2.3.2 Consequences of Faulty Execution 69

2.3.3 Mechanisms of Fault Injection 76

2.3.4 Lasers 82

2.4 Defence Techniques **88**

2.5 Observations **91**

This chapter describes a broad spectrum of attacks that a μC is vulnerable to and the relative ease with which they can be performed. It highlights that the cornerstone of effective defences lies not in ever more ingenious cryptography but through reliable execution of relatively simple code. Humble operation of basic arithmetic and branching guards the most valuable assets within a μC. It leads to the question, what is required to make these operations trustworthy?

It has been a popular misconception that a bug-free program, implementing a carefully defined protocol, executing on a well-maintained computer would provide all that was necessary for reliability and security. The implicit assumption is that the computer will reliably do what it is told to do and, so long as the task defined has no logical weaknesses, the outcome will be entirely deterministic. The only risks worth considering were physical. There is the risk that the device, or its sub-components, may be stolen; the defence against this is akin to having a safe, making it very heavy and bolting it to the office floor. Similarly, there is the risk that it may be attacked in-situ; the defence here for a safe is to reinforce it to make it resist attempts to break it open. For silicon chips, this meant adding physical barriers to prevent micro-probing.

This thesis is about what happens when one cannot fully trust a device to execute a program and what programmers can do to mitigate this threat. It is about the effects such errors have, the consequences of those effects, and programable defensive strategies to mitigate them.

Defences need to consider the other risks threatening an application and not unwittingly introduce new weaknesses. An awareness of the potential vulnerabilities, why they exist, how they are exploited, and the defences required in compensation is essential when considering new defences.

This chapter gives a broad overview of a μC's vulnerabilities and the range of attacks available to an adversary. It also highlights that underpinning all the sophisticated cryptography is a fundamental reliance on trustable execution.

2.1 Threats

Why would a micro-controller be attacked?

The form factor of a μCs invariably leaves it exposed to the possibility of attack, and it must therefore be considered vulnerable. Vulnerability in its own right is not a problem so long as adequate defences are used to mitigate the consequences of an attack. The required strength of those defences depends on multiple factors, primarily the ease with which an attack can be performed and the potential benefits gainable

by a would-be attacker. So, to answer this question, we need to consider the services that a μC provides and the environment it operates in.

μCs, in their many guises, provide a service for, and look after interests of, various parties. For example, a μC in a domestic appliance may be responsible for ensuring an optimal wash cycle in a washing machine or preventing an oven from burning your dinner. Here it acts as its owner's agent, it protects its owner's interests, and an owner-attacker will gain no benefit by compromising its operations. If they had access to it, a third party attacker would gain nothing either, unless their interest was in causing frustration or inconvenience. Even though an attack might be easy to perpetrate, it will be unlikely, and the consequences would be bearable if it were to happen.

Alternatively, a μC may manage a burglar alarm. In this scenario, its purpose is to protect its owner's interest, and that owner is interested in ensuring it is physically protected. A thief, however, would have a lot to gain if they could disable it or compromise its operations; such a device is an attractive target. It is in the owner's interest to keep it away from attackers, and it is reasonable to expect that the device will perform self-tests and summon assistance if it detects intrusion.

The third type of device provides one party with a service while protecting a third party's interests. A bank card, for example, provides a convenient way for an individual to access their money while simultaneously ensuring they cannot spend more than the bank allows. Such a μC is an exceptionally attractive target and requires robust defences because it is deployed outside of the control of its primary beneficiary.

These roles give us a concept of a target's low, medium and high attractiveness to an attacker.

- For devices with a **Low** attack attractiveness, attacks are unlikely, other than by those motivated by curiosity or possibly by competitors seeking to clone a device or discredit rivals. As a result, it is unlikely the device owner will pay any attention to securing access to the device or consider special options for disposal at the end of the product's life.

- Devices with a **Medium** level of attack attractiveness can expect a degree of physical protection to be provided by its beneficial owner. The threats here are logical ones that can be perpetrated remotely, for example, vulnerable passwords or remote software upgrades. For these devices, attacks requiring access to the device are unlikely but still require consideration, particularly to avoid the risk of tampering or substitution within the supply chain.

- Devices categorised as **High** on this attractiveness scale require extensive defences at all points in the product life-cycle. Manufacturers need to consider defences starting within their development environments and treat all participants in the product's implementation, construction, delivery, use, and disposal, as adversaries. These threats include insiders adding trap-door functions, assemblers building clones, tampering, substitution or theft during delivery and attacks by the end recipient.

2.1.1 Motivation

In performing its tasks, a μC is expected to fulfill two primary functions.

- **Protect secrets**. Secret keys, for example, if disclosed, enable an attacker to impersonate the μC and to obtain services fraudulently. A secure device must keep the important data secret from everybody and only use these secrets on behalf of the legitimate owner.

- **Control access to its services**. Possession of a device is not enough to prove a legitimate right to its services. Devices need to authenticate the user, and this role is just as crucial as protecting secrets. Additionally, it may need to prove its identity to let the user know it is a genuine article.

The benefits of an attack on a μC are varied but can be distilled down to four categories [164].

- **Theft of Service**. Cloning PayTV access cards was one of the first widespread frauds perpetrated against smart cards. PayTV operators even hacked rival's

cards in order to starve them of revenue and ultimately bankrupt them [172]. Similar risks exist for transport operators and their "smart tickets", although, to date, such subterfuge has not been detected.

- **Access to Information**. Identification theft. The ability to clone passports or even create fraudulent ones would give criminals a lucrative illicit income. Similarly, unauthorised access to devices such as password vaults would enable hackers to access information and services they had no right to access.

- **Cloning or Overbuilding**. Counterfeit products, e.g. printer cartridges. As supply chains globalise, increasing efforts are being expended to ensure subcontractors in the manufacturing chain cannot create additional stock and illicitly sell it into the market. Increasingly these defences involve intelligent Integrated Circuits (ICs), and the security of these components is the keystone in the defences.

- **Denial of Service**. Or, anti-competitive practices. If devices can be disabled or broken, the device holder will lose access to a service. And, since glitch attacks typically leave no trail, users will attribute blame for a product's reliability issues to the device itself or its retailer. To prevent reputational damage and to avoid the cost of replacing broken devices, the devices need to be robust against such attacks.

2.1.2 Mitigation

A product owner should seek to eliminate, or at least reduce, the motivation for an attack. This is achieved by obstructing attacks and reducing the rewards gained from a successful attack. The three main mechanisms available to the product owner are physical, cryptographic and logical.

The role of physical defences is to ensure that an attacker cannot extract secrets from the device. This is sometimes called *Hardening* and aims to prevent an attacker from dismantling the chip, monitoring or injecting signals within it, and manipulating

the operating environment. These attacks mechanisms are described in Section 2.2.

Cryptographic defences seek to disguise data or to identify unauthorised manipulation of them. Cryptography provides four services [70, 157] and, depending on the nature of the threat, combinations of these can be deployed to obstruct an attacker. These services are,

- **Confidentiality**. An eavesdropper, or Man in the Middle (MitM), should not be able to read a message. The use of encryption should ensure that messages in transit cannot be read by anyone other than the intended recipient(s).
- **Authenticity**. It should be possible for a message receiver to verify its origin. In other words, an adversary should not be able to assume a false identity. Where both parties in an exchange of messages can establish the counter-party's authenticity, it is referred to as Mutual Authentication.
- **Integrity**. The recipient should be able to confirm that the message has not been modified while passing from the sender to the receiver. This capability is commonly called Message Authentication and implemented via a Message Authentication Code (MAC) algorithm.
- **Non-repudiation**. A sender should not be able to claim that an imposter sent a message signed by him. This is a combination of Authenticity and Integrity, often referred to as a *Digital Signature*.

Logical defences ensure the device cannot be operated by, or on behalf of, an unauthorised third party. There is little point in a μC protecting the confidentiality of cryptographic keys if that device can be freely used to encipher data using those keys. Lost or stolen devices need to be of no use to an adversary.

2.1.3 Relevance

The importance of naturalising threats extends beyond the prominent examples of bank cards and burglar alarms. Increasing numbers of the devices we use in our daily lives contain μCs, and our dependence on these devices makes their reliability

of utmost importance. For example, 'old' telephones had basic functionality and were powered via their connection to the exchange. Modern telephones, with their in-built address books, hands-free operation and answering machines, appear substantially more useful. This additional functionality requires a μC and an independent power supply. Unfortunately, if the μC or the power supply fails, this device can no longer be used to summon help. The trend of increasing reliance on evermore intelligent devices places an onus on the reliability of these devices, and this reliability includes their ability to remain operational in adversity.

The expanding domain of IoT technology has seen humble appliances become internet-connected. Software flaws in a kettle have been exploited to reveal the access key to its owner's Wi-Fi network [84], providing the attacker with full access to the supposedly firewall-protected local network and unrestricted access to the internet at the kettle owner's expense. A similar bug in a BBQ grill [135] enabled the device to be controlled remotely. These devices and many like them are locatable by a practice known as Wardriving, which involves identifying them on Wi-Fi networks, usually from a moving vehicle.

Researchers have also demonstrated an attack on a moving car, giving the attackers control of steering, brakes and engine control. The problem is widespread, Jeep [72], Tesla [98], Toyota and Ford [181]. Similar mistakes by similarly naive developers indicate a pervasive lack of appreciation of both the risks and attackers' capabilities. The threat also exists for biomedical devices such as insulin pumps [48]. CNN reported that US vice-president Dick Cheney had his heart pacemaker modified in anticipation of just such an attack [68]. These attacks target logical weaknesses in the devices and highlight that they perform a trusted safety-critical service. While we can expect these 'early adopter' logical oversights to be rectified, it is also clear that the remedies will require additional secrets and secure programming within the devices, emphasising the need for secure coding.

Similar methods can be used to attack home appliances. Taking control of IoT devices and re-purposing them to form Botnets is a significant threat. Network infrastructure manufacturers have started issuing guidance for IoT equipment developers.

Trend Micro [114], and Nozomi-Networks [145] both provide detailed reports on various forms of malware, ransomware and common vulnerabilities. The need for this is emphasised by reports that intelligence agencies, such as Russia's FSB's Fronton Program [69, 149], are actively researching IoT botnets with these aims in mind and have successfully demonstrated a Distributed Denial of Service (DDoS) attack against Twitter, Netflix, Spotify, Paypal and Amazon. They were all taken offline for several hours in 2016 [146]. The security and integrity of the humble μC will become increasingly critical to the orderly functioning of the internet.

Even devices operating within an isolated private network are similarly vulnerable. A range of light bulbs manufactured by Philips contains a bootloader that permits them to be upgraded in-situ. Each bulb can communicate with near neighbours to ensure the prorogation of new system code. Researchers [150], using sensitive listening devices, monitored the bulb from outside a building and obtained the secret key required to infect a device with a self replicating computer program (Worm). This worm enabled the attackers to control or destroy all infected devices. The alarming feature here is that it is effectively a computer virus; it spreads from one nearby device to another. The devices need not have an owner in common or be attached to the same network. Where a critical density of devices exists, an unstoppable chain reaction occurs.

These issues have been understood in the high-security world of smartcards for a long time. Specialist programmers deploying defensive code alongside a support infrastructure of code reviewers and penetration testers combine to ensure defences are in place for all conceivable relevant attacks. Payment scheme owners collectively publish minimum security requirements that all products they accredit must satisfy [66]. Scheme owners then add their additional specific requirements, such as those for Visa [182] and Mastercard [111]. Scheme certifications provide a high level of security and confidence. However, this all comes at great expense. Some of these practices will likely become vital in the IoT. A report by PenTest Partners on the security of μC controlled devices, and in particular their software defences, states: *"Adoption of these has slowly increased in the server and desktop Linux market, but it is still rare*

to find them used in embedded systems" [174].

Even after verifying algorithms and double-checking the accuracy of their implementation, there remains the threat of faulty execution. Forcing μCs to make errors is far easier than many programmers realise. At the RaspberryPi Micro-computer press launch, a device ideally suited to IoT applications, flashguns on the press cameras caused the devices to crash; much to the amusement of the press [73] and the embarrassment of the developers [168].

Knowing when to induce an error, predicting and then exploiting the consequences, is the problem faced by an attacker. Anticipating errors, and reacting appropriately when they are detected, is the task faced by the defenders.

2.2 Attacks

There are multiple ways to attack a μC, various consequences of an attack, and different motivations for launching an attack.

An often-cited way to consider a threat is to consider the adversary [7, 184]. This categorisation looks at the availability of the knowledge required to perform an attack, access to privileged knowledge or processes, and the resources required to accumulate the knowledge.

- A1 **Clever Outsiders** — Would the device be vulnerable to an attacker who could obtain the samples and information required to implement an attack?
- A2 **Script Kiddies** — Could a vulnerability be exploited via a tool that encapsulates specialist knowledge? Like the clever outsider, *Script Kiddies* use publicly accessible knowledge, but they require a primary adversary first to generate and publish a tool that exploits a weakness. They do not need the same depth of knowledge to carry out a prescribed attack. Script kiddies are essentially nuisances between the time an exploit has been identified and before a fix can be deployed.
- A3 **Knowledgeable Insiders** — Could someone on the inside of an organisation compromise the product's security? This may be by informing and guiding an

outside attacker or by placing deliberately exploitable flaws in the product. The latter mechanism is sometimes called a *trap-door* or *logic-bomb* [104].

A4 **Funded Organizations** — What resources would an attacker require to perform an attack? Adversaries in this category range from well-equipped laboratories to organized crime. An organization with specialist equipment could implement attacks that are prohibitively expensive for amateurs or resource-constrained attackers. In this scenario, it is more likely that the attacker would need to justify the cost of the attack against the potential rewards. Intellectual curiosity or bragging rights within a peer group are unlikely to be the primary motivation for the attack.

Alternatively, funds can be deployed as bribes to gain inside knowledge from the product's developers or for financing threats and intimidation. The latter is also known as *rubber-hose cryptanalysis* [157].

The reliance on the distinction between Clever Outsiders and Knowledgable Insiders is *Security Through Obscurity*. The weaknesses of this as a security mechanism have been recognised for a long time [97], and observers argue that disclosure of implementation details improves security by allowing sympathetic analysts to review implementations, satisfy themselves regarding the quality of the code and highlight weaknesses that can be fixed [83].

An alternative way to look at threats, in the context of μC attack, is via the obstacles faced by the attacker when performing an attack [164].

- **Non-invasive** — These attacks operate via the normal interface to the unmodified device, and physical access to the device is not always necessary. A device can be compromised by analysing signals between it and its environment or manipulating its inputs or environment. The knowledge required to perform these attacks is readily available from a device's datasheets and published papers outlining attack methodologies. Non-invasive attacks are relatively low cost to implement as no special processing of the device is needed, and the equipment required to perform an attack is readily available.

- **Semi-invasive** — These attacks require physical access to the target device. The device itself remains functionally unmodified, but its packaging and some defensive structures may require removal. Internal signals can be directly measured by probing the exposed inner device, or features, normally inaccessible through the packaging, can be manipulated. These attacks often require relatively complicated processes to prepare samples for attack and require specialist equipment to perform attacks. It is also unlikely that the details of the exposed device will be publicly available. Specialist knowledge and time will be required to identify the attackable features of the device. These attacks can be considered as affordable to a motivated attacker.
- **Invasive** — Here the device is often physically modified, for example, cutting or joining tracks within the IC. These attacks require extensive knowledge of the DUT and highly specialised equipment; they are consequently expensive to perform. However, this equipment is sometimes accessible to poorly funded attackers, such as students within university research departments. Therefore, the cost of the equipment alone cannot be considered a defence.

Perhaps the best categorisation of attacks is based on the attack mechanism and the methodologies required to exploit results. These attack classes have been defined in different ways [14, 177, 164]. However, the common features of these categorisations give three significant classes of attack with different exploitation techniques within them. They are Logical attacks, Side-channel attacks and Physical attacks.

2.2.1 Logical Attacks

Logical attacks target the underlying logic or process used to protect information. They exploit features of an implementation to break a defence and are the oldest category of attack, predating computers. It is a never-ending competition between code makers and breakers, employing increasingly sophisticated mathematics to outwit the opposition.

The unifying characteristic of these attacks is that they are non-invasive. They

are performed on raw observable data collected while it is being transferred between two points.

Logical attacks can be further sub-divided into three primary categories.

2.2.1.1 Brute Force

Brute force attack is the most basic of attacks. It involves exhaustively testing all possible combinations until the answer is found, and it is the benchmark by which all other attacks are measured. The hypothetical perfect cypher algorithm, i.e. immune from all possible cryptanalytic techniques, will still be vulnerable to a Brute Force attack.

The limiting factor in Brute Force attacks is the time it takes to test every possible solution. This is effectively a measure of the current state of the art in computer performance.

For example, the ancient classic Caesar Cypher is perhaps the simplest example of a transposition cypher. It can be broken using pencil and paper, requiring the investigation of at most 26 possible solutions.

2.2.1.2 Analytical Attacks

Analytical attacks improve upon the base level efficiency of Brute Force by exploiting features of the data or the algorithm to narrow down the search space and thereby reduce the number of possible candidate solutions that need testing.

The weakness may be in either the raw algorithm or a side property of the data it processes. In some cases, the underlying assumptions about either mathematics of a system, or the algorithms available to crack a problem, are flawed; for example, Shamir's crack of the Merkle-Hellman knapsack cryptosystem [160].

Alternatively, systems can be compromised by using sneaky mechanisms to fool a counterparty into a false sense of trust. Poetically described as 'Programming Satan's Computer' [8], misleading or counterfeit data may be substituted for genuine data at a victim's site. The victim then, unknowingly, may use this information to verify forged data. The process can be as simple as getting a bogus public key onto a

victim's computer. When security relies on something stored on, and processed by, a computer, can anyone be sure it is safe to trust it? Defences against this problem exist as Public Key Infrastructure (PKI)'s certification hierarchy [136], where increasingly independent trusted agents sign and vouch for the authenticity of keys, giving a chain of certificates up to a (presumed) trusted root Certification Authority (CA). Alternatively, PGP's '*Web of Trust*' [2], replaces the hierarchy with a peer to peer trust model, removing the need for a trusted root.

Pure mathematical attacks are comparatively rare, but attacks based on the processed data's statistical properties are relatively common and have a long history. For example, knowing the relative frequency with which particular letters occur within a language can guide a search strategy towards likely candidates first, thereby increasing the chances of finding a solution early within the wider search space. Knowledge of a fraction of the enciphered text, known as a crib, has also been used to optimise search strategies and significantly improve a Brute-Force attack's performance. Most famously seen with the mechanised cracking of the German Enigma cypher by Turing et al. at Bletchley Park during World War II.

2.2.1.3 Exploiting Flaws

Sometimes, it is possible to take advantage of mistakes made when defining or implementing a feature within a device.

Implementation flaws are a common feature of many internet-based attacks. Here an attacker constructs a message for the target computer, and the target computer inadvertently discloses data as a consequence. The classic example is an oversized data packet, or *buffer overrun*. Typically a buffer overrun corrupts adjacent data and, depending on the buffer's location, many outcomes are possible. Corrupting the call stack can lead to a crash when the executing function attempts to return. Strategically placed data within the overrun can capture the crash and assume control of the device [167]. Alternatively, variables or keys may be corrupted, leading to exploitable responses from the device.

Bugs unknown to the device manufacturer can sometimes be located and exploited.

One example, known to the author, involved a password attack counter. Typically such an attack counter would be incremented with each unsuccessful verification event and reset after a successful event. If the count reaches a threshold, the device should be locked, and all further verification attempts should be rejected. This mechanism is intended to prevent a brute force attempt at guessing the password. In this particular case, and despite rejecting further attempts, the counter continued to be incremented until it reached 255. At that point, the counter wrapped round to zero, effectively resetting and permitting an additional set of guesses to be tested.

2.2.2 Side-Channel Attacks

Side-channel attacks occupy the space between Logical and Physical attacks. The attacks involve observing the device's behaviour during an attack but do not require the device to be tampered with or otherwise modified. Sides channels leak information about the device's internal operations while it performs a task. Traffic analysis is a classic historical example of a side-channel. Ostensibly an adversary can learn little from his enemies encrypted communications other than perhaps the source and destination of a message. However, a sudden increase in frequency or size of messages may be interpreted by an observer as the prelude to an impending action. Journalists used the same techniques to infer the start of the 'Desert Storm', the opening action in the first gulf-war. Here analysis of pizza deliveries to the White House indicated an abnormal number of late working staff [116, 153].

Side-channel analysis is a powerful attack technique, primarily because it is passive and the target remains unaffected by the attack. A target must assume it is under attack all the time because it has no way to determine when it is being attacked. For a μC, continuous defence deployment may make the device extra power-hungry or slower than could otherwise be achieved. The example above would require additional orders of unwanted pizzas on quiet days to disguise the occasional days when they are wanted.

2.2.2.1 Heat, Light and Sound

Heat, light and sound are relatively exotic and hard to exploit side-channels in μCs. Electrical currents, flowing in a semi-conductor, generate heat. Heat depends on the magnitude of the current and resistance of the conductor. The current depends on the rate of switching of the device's transistors. While many observers have postulated that this may be exploitable [17, 22, 46] there are significant practical difficulties in doing so [85]. The observable heating effect is averaged and delayed by the chip packaging and the packaging's thermal conductivity. Alternatively, visible access is required to the chip's surface to observe localised heating effects [163]; however, results suggest the approach is better suited to locating active areas of a device rather than for obtaining insights into the data being processed.

Localised heat causes localised physical expansion and contraction of the device. These mechanical stresses can cause acoustic noise, the volume and frequency of which is related to the rate of heating and cooling occurring within the chip. This approach has been successfully demonstrated against laptop computer to extract RSA keys [71]. It is reasonable to assume similar effects are observable in a μC. However, there are more accessible side-channels that can be exploited to indicate the same underlying behaviour.

2.2.2.2 EM Radiation

Electrical currents momentarily flow each time a semi-conductor gate changes state. The more gates that simultaneously change state, the higher the momentary current is, and pulsating electrical currents cause Electro-Magnetic (EM) radiation. The strength of the EM waves correlates with the activity within the μC and with the data it is currently processing. This attack technique is challenging to implement without shielding the device and the signal receiver from background noise. However, when it is possible to collect sample data, it has been shown to be very effective [112].

2.2.2.3 Timing

Timing data is the most readily available and most straightforward to interpret side-channel data source. Consequently, it is often the best-defended aspect of an application's implementation.

A typical timing attack requires the ability to time the delay between sending a message and receiving the response. This can often be performed directly by the attacker while delivering commands to a DUT. For very small time differences, an oscilloscope or similar device may be required to measure a signal wire directly.

In its simplest form, the time taken by the μC to respond to a stimulus can leak secret information. For example, if an application checks an inputted password using basic implementations of the C standard `strcmp()` or `memcmp()` functions, then the response time will vary depending on which byte of the input differed from the secret reference. The first byte of a password could be determined by testing all possible values of that byte. The correct guess can be identified because it takes marginally longer to respond than the other guesses. Once the first byte is known, the same mechanism is used to determine the second byte; and so on until the whole password has been reconstructed. For an alphabetic password of length n, this process would take, at most, $26 \times n$ tests to crack as opposed to the 26^n tests that a brute-force attack may require in the worst case. Obviously, (most) application developers are wise to this and perform time-invariant comparisons.

Sometimes, the defences themselves add weaknesses that a timing attack can exploit. Consider the previous example of a password attack counter. Recording a failed attempt, or resetting the counter, requires an update of non-volatile memory, and the moment that this occurs can be detected via a side-channel. Non-volatile memory updates are significantly slower than normal RAM or register updates, and interrupting the process very early can prevent the update from occurring. If, as is likely, it takes a different amount of time to initiate either of the updates, an attacker can identify which one is being performed. Quickly cutting the power to the device before the process begins enables the attacker to stop an attack counter from being

incremented and neutralises the defence. Once again, we would expect application developers to be aware of the threat and structure the program to resist the attack.

Simple timing attacks are relatively easy to predict and defeat by a programmer who is fully aware of the threat. However, a class of timing attacks uses statistical analysis of large numbers of samples to unpick a secret process. First described by Paul Kocher [100], these statistical timing attacks are particularly powerful.

Consider Algorithm 1.

Algorithm 1: basic time invariant RSA

Data: *Signature* C, *Message* A, *Modulus* M, *Key* $B = (b_{n-1} \dots b_1 b_0)_b$,

Result: $C = A^B \; mod \; M$

```
C ← 1
for i = n − 1 ... 0 do
    C ← C × C mod M
    if b_i = 1 then
        C ← C × A mod M
    else
        C ← C × 1 mod M
return C
```

Here the programmer has attempted to make the calculation time-invariant: i.e. execution time should remain constant for a given key length, irrespective of the value of the secret key. In practice, this algorithm is frequently implemented using Montgomery Multiplication [117]. Apart from some pre and post-processing of the *Message* and *Signature*, the RSA component remains the same. Montgomery's multiplication algorithm (taken from [113]) is shown here as Algorithm 2.

Algorithm 2: Montgomery multiplication

Data: $m = (m_{n-1} \ldots m_1 m_0)_b,\ x = (x_{n-1} \ldots x_1 x_0)_b,\ y = (y_{n-1} \ldots y_1 y_0)_b$,
with $0 \leq x, y < M,\ R = b^n$ *with* $gcd(m, b) = 1,$ *and* $M' = -M^{-1}$ *mod* b
Result: xyR^{-1} *mod* m

$A \leftarrow 0$ (notation $A = (a_n a_{n-1} \ldots a_1 a_0)_b$)
T: for $i = 0 \ldots n-1$ **do**
  $u_i \leftarrow (a_0 + x_i y_0)m'$ *mod* b
  $A \leftarrow (A + x_i y + u_i m)/b$
S: if $A \geq m$ **then**
  $A \leftarrow A - m$
return A

For a 100 bit key, this multiplier will be called 200 times. Step T will execute in constant time, while step S may add a small delay on some invocations of the routine.

If an attacker can request the signing of a large number of different messages and collect each signature calculation's execution time, he can then determine the secret key. Each signature's time will be similar but differ slightly, depending on how many times the subtraction step S was required. The process is as follows.

- Knowing the original *Message*, an attacker can predict if 0, 1 or 2 subtraction steps are required while executing the first round of the RSA loop, depending on whether bit b_{n-1} is either 0 or 1.
- For a supposed bit state, the attacker then categorises each sample as belonging to set 0, 1 or 2.
- If the average time of the members of set 0 is less than the average time of set 1 and that is less than the average time of set 2 then it is highly likely that the guess relating to the state of that bit of the key was correct.
- Conversely, if the average times of each set are not ordered, then the guess was probably wrong.
- Now, knowing the state of b_{n-1}, the attacker can repeat the process. Calculating the intermediate value of C for the loop round where $i = b_{n-1}$ and predicting the need to use the subtraction steps in the next round where $i = b_{n-2}$.

- This process can be repeated for consecutive passes of the loop. The known portion of the key, up to the loop round with the first unknown key bit, is used to calculate the intermediate value of C. The samples can then be recategorised, based on the expected use of step S on this round, and applying the same reasoning will disclose this next unknown bit. Each time the process is repeated, one more bit of the key is discovered, enabling the attacker to progress sequentially through each round until the whole secret key has been determined.

The simplified process described here can be extended to more realistic implementations of modular exponentiation and applies to many cryptographic algorithms [183]. Wherever the data being processed influences the execution time, it is vulnerable to a statistical timing attack. It is particularly difficult to defend against this attack if the algorithm cannot be made time-invariant. Adding random time delays merely dilutes the signal to noise ratio, which can be overcome by collecting more samples. Random transformations of the data or keys can provide protection but also exposes alternative opportunities for attack.

Execution time needs to be considered for all security operations. This runs contrary to the common desire to optimise all operations for speed.

2.2.2.4 Power

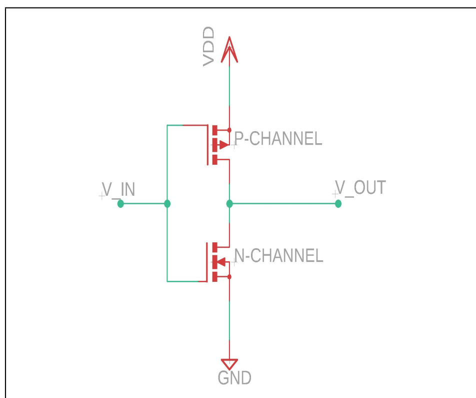


Figure 2-1: CMOS Inverter

For μCs, power analysis is perhaps the most versatile side channel-attack mechanism. The logic gates within an IC are constructed from combinations of npn & pnp transistors. For example, Figure 2-1 opposite, shows an inverter. When V_{in} is high, the N-channel gate opens, connecting V_{out} to ground; conversely, when V_{in} is low, the P-channel gate is open instead, resulting in a high output for V_{out}.

Whenever the gate changes state, a small current flows, and this is due to two factors. Marginally differing switching speeds between the two transistors may permit

a current flow for a very brief period, and the capacitance of V_{out} leads to a short-lived current whenever the voltage changes. On each clock cycle, potentially millions of transistors are switched. Each transition results in a small flow of current, and the sum of these small currents is observable as the current drawn by the device. The measured current directly relates to the sum of all the small changes occurring within the IC.

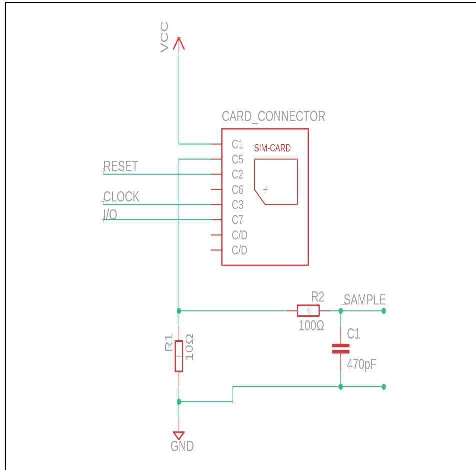


Figure 2-2: Power Measurement

A small (low value) resistor placed in series with the power supply enables an attacker to measure this current. By using an oscilloscope to measure the voltage at the identified SAMPLE point in Figure 2-2, the attacker can indirectly measure the current; it being proportional to the voltage across the resistor R1.

The current's magnitude relates to the number of transistors changing state, and propagation delays within the device result in transistors switching state at different times. This leads to characteristic shapes in the measured waveforms rather than momentary current spikes coincident with each clock edge.

A typical waveform is shown in Figure 2-3. The blue trace, labelled *Raw* is the data, as sampled by the oscilloscope. It contains minor quantisation errors and high-frequency noise typical of measurements taken directly from a DUT. The red trace shows the same data after filtering out the noise by using a simple moving average or boxcar filter*, Equation 2.1.

$$\bar{P}_i = \frac{1}{2w+1} \sum_{j=-w}^{w} P_{i+j} \tag{2.1}$$

The two pronounced spikes at either end of the box labelled C in the power trace sample (Figure 2-3) demonstrate another phenomenon. They are coincident with `Call` & `Ret` operations where a relatively large number of address bits change within

*The window width w, in this case, was 8, and the result shifted by +0.1µV for clarity.

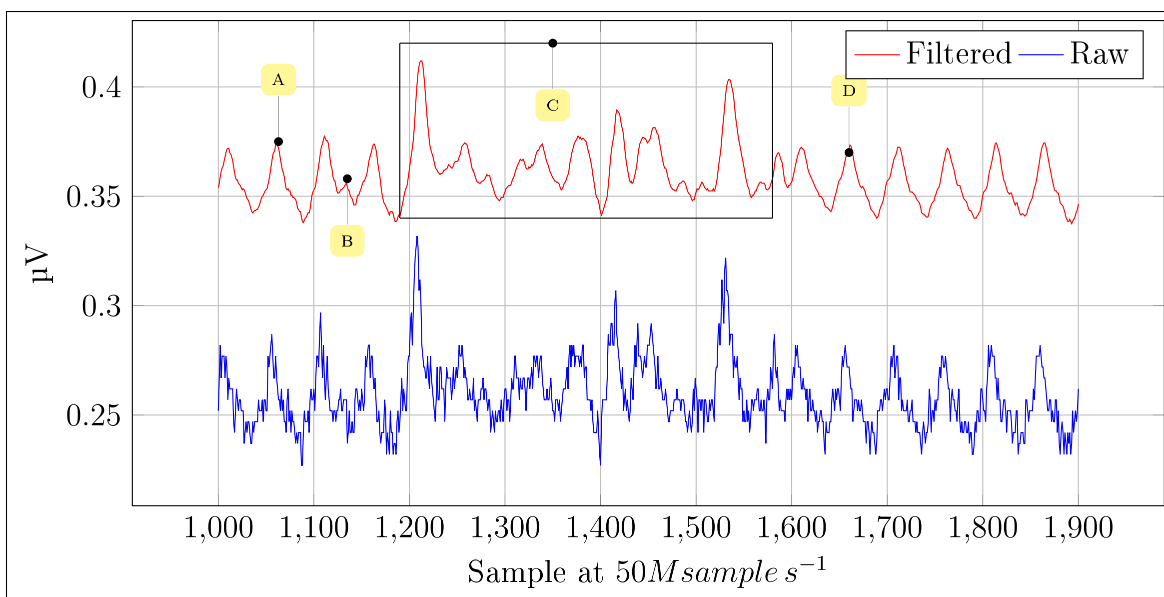


Figure 2-3: Typical Power Waveform

a single instruction. Besides the μC itself, integrated peripherals may be selectively activated, leading to distinctive periods of increased power demand. For example, a cryptographic co-processor would usually be idle. When it is used, we would expect to recognise this in the power trace as an increase in the power demand for the duration of the peripheral's operation.

2.2.2.4.1 SPA

Direct observations of the μC's power are the basis for Simple Power Analysis (SPA). SPA has three main uses in chip analysis.

Firstly, recognisable patterns in a power trace can be identified and used to synchronise other activities. In the trace sample (Figure 2-3), the periods A & D on either side of box C show repeated similar operations. It is a single-instruction loop, waiting for a status flag to change. Period C shows markedly different behaviour. Here, a subroutine is called, executed and returned from, breaking the simple loop's repetitive pattern. Events like this can be recognised and used to synchronise an attack. For example, cutting the power when a non-volatile memory write is detected is a way to prevent a device from recording an event.

Secondly, observations of the timing of events, the time between events, or the

events' duration may indicate secrets within the device. The process can be likened to that used by the stereotypical safe-cracker who listens for clicks emanating from within a lock mechanism to infer the lock's combination and gain access to the safe. Take, for example, a simple implementation of an exponentiation algorithm, as shown in Algorithm 3.

Algorithm 3: Crude Exponentiation

Data: *Number* X, *Exponent* $Y = (y_7 \ldots y_1 y_0)_b$
Result: $Z = X^Y$

$Z \leftarrow 1$
for $i = 7 \ldots 0$ **do**
  $Z \leftarrow Z^2$;
  if $y_i = 1$ **then**
    $Z \leftarrow Z \times X$;
return Z

Each loop begins with a squaring operation. There is a quick decision made as to whether to perform the multiplication step or not. It is then followed by another decision about repeating the loop or terminating. Figure 2-4 shows a power trace of this algorithm in action. The multiplication (or Squaring) process takes marginally more power than the simple control logic. Presumably, the multiplication circuitry in

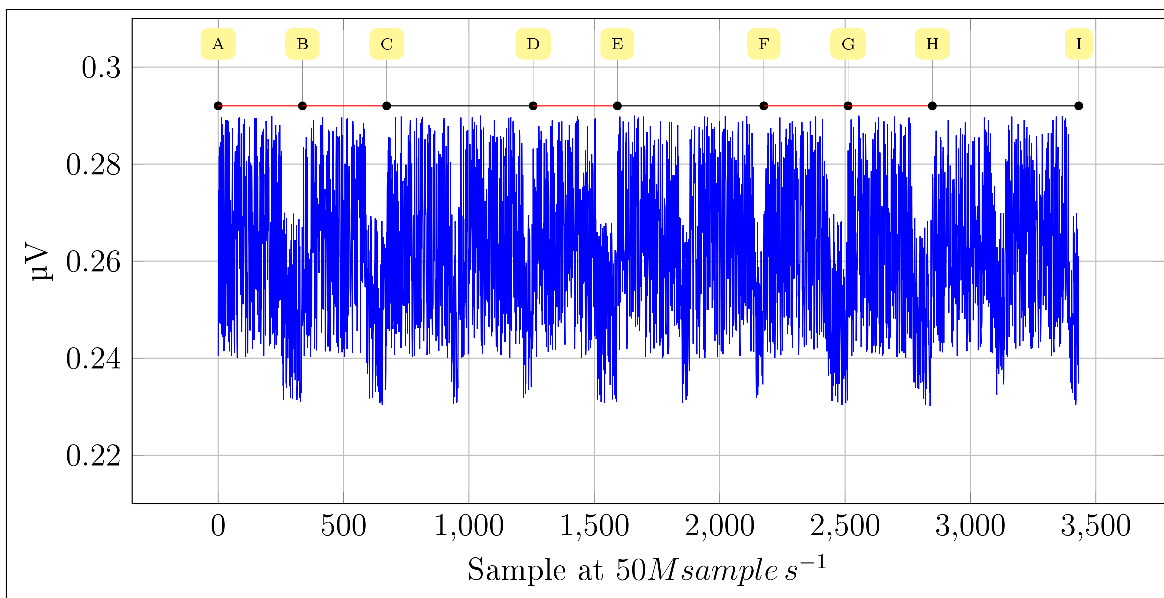


Figure 2-4: Crude Exponentiation

the IC activates more gates than the surrounding loop management. We can see that between power bursts, there are either short or long pauses, and from the algorithm, we can surmise that a long pause relates to the decision to skip the multiplication and perform the loop management logic. A short pause relates to performing the multiply immediately after the square. Thus from label A to B, we can assume bit Y_7 is 0. From B to C, bit Y_6 is also 0. A short delay separates the two power bursts following C; thus, we can infer that this pair of power bursts identifies a square followed by multiply, and that bit Y_5 must be 1. The process is repeated until the whole of $Y_{7...0}$ has been recovered. The exponent in this example is 00101001_b.

The third use of SPA is to identify familiar behaviour in an executing μC and identify abnormal behaviour. Power trace analysis has been used to recognise individual instructions as they execute within a μC [121]. This technique has the potential to verify code integrity, detect counterfeit components, or the presence of malware.

Nearly all code is suspectable to SPA unless the programmer takes specific defensive actions. The programmer must ensure execution time is not influenced by the data being processed because SPA can provide the same data as the statistical timing attack described earlier and requires fewer samples. Ideally, the execution path should also be constant to prevent the leakage of power profiles of the different instructions executed within the alternative paths. Another defence is to add noise to the power signal. However, this can be eliminated by Differential Power Analysis (DPA). Finally most effective, but not infallible, defence against SPA is the introduction of random delays. This confuses time measurements and makes comparisons between traces difficult.

2.2.2.4.2 DPA

Differential Power Analysis (DPA) is another statistical attack devised by Kocher [99], the author of the similar statistical timing attack described earlier. The attacker needs access to multiple power traces of the target performing a particular operation. The attacker needs to know either the input or the output data but does not need control over it. As with the timing attack, the attacker makes predictions about the internal

behaviour of the μC at a specific point in the calculation. This prediction defines a discriminant function that prescribes adding or subtracting the waveform from a running total. Repeating this for all sampled power traces produces a new averaged trace. Most of the points on this trace will have the same average value because their component influences were added or subtracted without any correlation between the predicted and observed behaviours. If an action occurs at some point in the trace that is consistent with the prediction, the averaging process will be biased. This point on the accumulated trace will deviate from the average of the other, unrelated points.

For example, consider an attack on first round of the Data Encryption Standard (DES) algorithm [140], as shown in Figure 2-5. Operation C, green in the figure, is a compression function involving *SBoxes*, each taking 6 bits of input and delivering a 4 bit result. This structure is shown in Figure 2-6. There are 32 possible values for bits of $k_1[bits\ 0 \ldots 5]$. Now assume an attacker can collect a set of power traces for known *Plaintext* input. For a given value of $k_1[bits\ 0 \ldots 5]$, it is possible to calculate the input and output of S_1 for each of the pre-collected traces. These traces are then added to, or subtracted from, an accumulated trace depending on the predicted output value of $S_1[bit\ 0]$. If this candidate value for $k_1[bits\ 0 \ldots 5]$ is correct, then the accumulated trace will show spikes wherever the prediction correlates with the observed behaviour. See Figure 2-7. When the guess was wrong, there will be no apparent spikes in the resulting trace and another candidate for $k_1[bits\ 0 \ldots 5]$ can be tested. It is not unusual to see more than one correlation spike as the property chosen as the discriminator may be exhibited more than once. In this case, the output of S_1 is also seen in the input to permutation P.

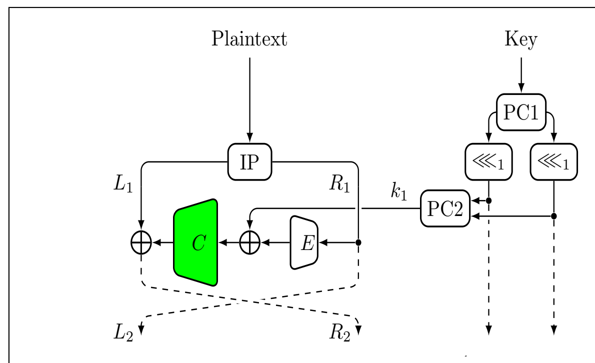


Figure 2-5: DES Round 1

The whole process can then be repeated for $k_1[bits\ 6 \ldots 11]$ and S_2. Then again for $k_1[bits\ 12 \ldots 17]$ and S_3, etc. until the whole of $k_1[bits\ 0 \ldots 47]$ has been discovered. The same set of power traces can be reused at each stage of the process.

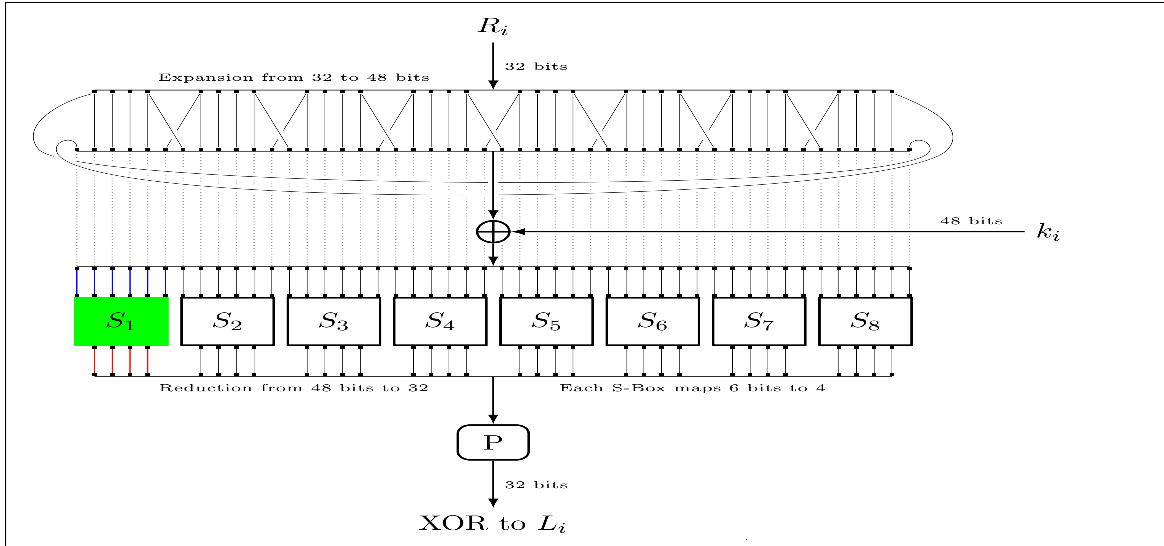


Figure 2-6: DES S-Box Structure

Depending on the characteristics of the algorithm under attack and possibly details of its implementation, alternative discriminants may be chosen and may be more or less effective. For example, the hamming weight of an intermediate result may be used, or even the difference in hamming weights between two consecutive calcualtions [54]. The latter method exploits the property that larger currents flow when gates are changed than when they retain their current values. Using a model of the underlying chip behaviour to optimise the discriminant function is known as Correlation Power

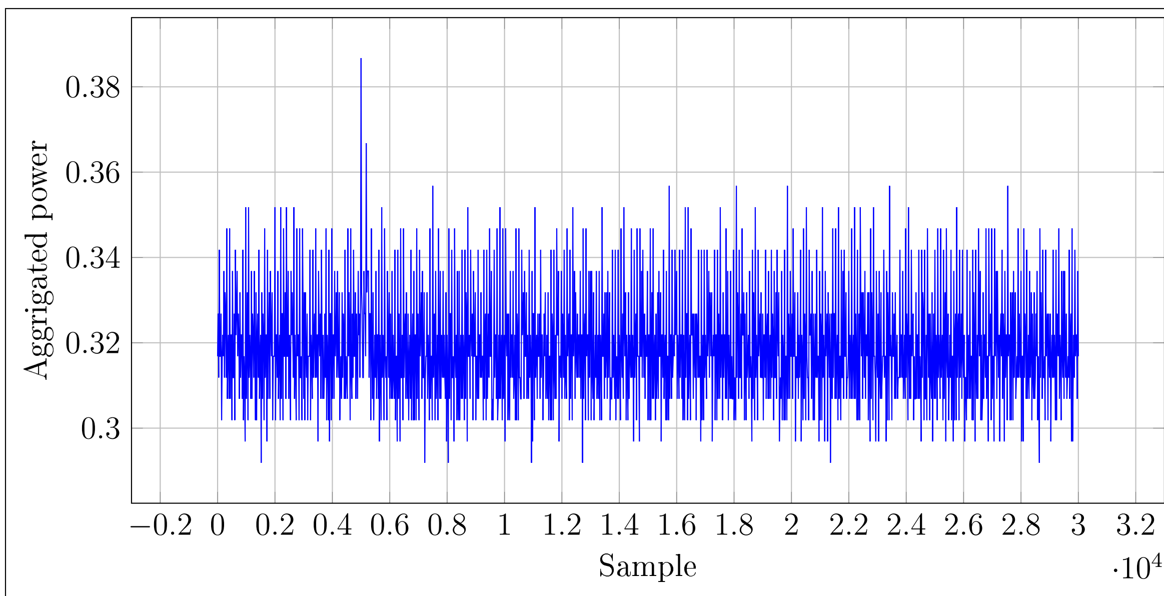


Figure 2-7: DPA Accumulated Trace

Analysis (CPA) and is sometimes identified as a separate attack methodology in its own right [45]. In CPA the process is logically the same as for DPA, however, the discriminant function is fine-tuned by observing the behaviour of the device with known input data and keys.

Defences against DPA include adding a temporal jitter to the processing to break alignments between the trace samples. Some security chips achieve this by randomly inserting wait-states in the execution of individual instructions. Similarly, random power consumption can be used to obfuscate operations. This is often implemented in silicon as a controllable defence feature. These techniques are examples of a defensive strategy known as Hiding, and they change the signal to noise ratio in the power traces. Unfortunately, Hiding can often be compensated for by collecting and examining more samples. An alternative, widely deployed defensive technique is Masking. Here the processed data is manipulated, or masked, in a way that does not affect the outcome but does affect the intermediate values and hence removes the opportunity for correlation on predictable intermediate data. Mangard et al. [110] have published a comprehensive reference for DPA, DPA defences, and attacks on defences.

Another technique, widely used in industry, has been generally ignored in academic literature. It involves recognising an attack and blocking further invocations of the vulnerable process to prevent the collection of usable power traces in the first place. It is not applicable in all cases, but mutual authentication protocols and session key generation are a case in point. In this scenario, each party in the exchange provides some unpredictable input, and both parties prove their authenticity by performing cryptographic operations on that data. This *Challenge / Response* behaviour is particularly vulnerable to DPA as the attacker has control over the data supplied to the victim. In normal operation, this process would be expected to terminate successfully, and when under attack, it would fail to complete because the attacker could not complete the cryptographic challenge. The device protects itself by maintaining an attack counter and registering the number of incomplete authentication attempts, much like the PIN try-limit described earlier.

2.2.3 Physical Attacks

The most straightforward and crude attack is the simple physical destruction of the device. Clearly, it would be foolish to launch such an attack on one's own e-purse or electronic travel card as this would result in a personal loss of money or in being denied a useful service.

Figure 2-8: ePassport Chip

However, destruction may be advantageous if the device was an electronic tachograph and held incriminating evidence of historical traffic violations. In the early roll-out phase of EMV's Chip-n-Pin program, it was acceptable for merchants to accept the legacy signature if the more secure PIN option was unusable. Criminals with stolen cards were known to break the chip to make purchases with an easily forged and seldom checked physical signature. Doing so bypasses the card's in-built security measures that restrict the number, and accumulated value, of unauthorised transactions. Some e-Passports make the chip visible, see Figure 2-8, making tampering more easily detectable. Defences for this class of attack require physical robustness for the device and an enforceable duty-of-care obligation on the holder.

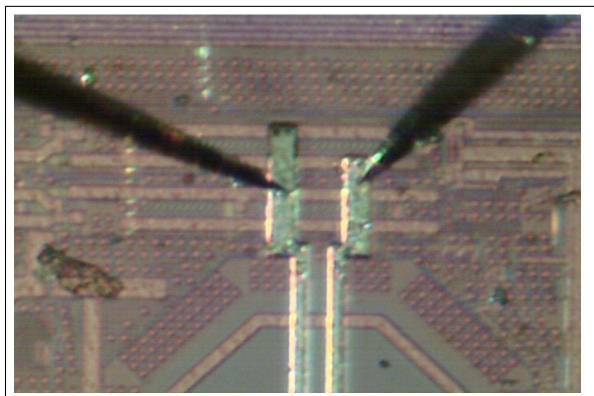


Figure 2-9: Chip Probing

Attacks that do not destroy the device are far more interesting but significantly more expensive to perform. Electrical connection with individual tracks on an exposed chip's surface can be made using fine-pointed needles, as seen here in Figure 2-9. Contact like this enables an attacker to monitor a signal line or inject a new signal into it while the

chip while running.

A Focused Ion Beam (FIB) workstation is perhaps the most versatile tool in the attacker's armoury. A FIB can be used to cut tracks to isolate internal components disabling a subcomponent or enabling the injection of externally synthesised signals. Holes can also be drilled with a FIB to enable physical access to deeper layers within a chip. This exposes more of the device's internal signals to probing. Finally, a FIB can also build up new metal deposits, creating new tracks and connections. Such equipment is very expensive and only available in the best-equipped laboratories.

Chips can also be reverse-engineered by a process known as delayering. By delayering a chip, an attacker can trace the paths taken by individual tracks and create a schematic diagram of the chip's circuitry. The most basic method involves polishing for the mechanical removal of thin slivers from a chip's exposed surface. Optical inspection of the device as the polishing proceeds enables the attacker to trace each track's path individually. This 'kitchen table' technique was used to great effect by Nohl [127] to crack Mifare's *Crypto-1* cypher. Open-source tools exist to assist in interpreting images of delayered chips and reconstructing the schematics [158].

Other techniques to delayer a chip exist too. Wet Chemical etching can selectively remove materials, and by alternately using different etching chemicals, a chip can be deconstructed, one layer at a time. Alternatively, Dry Plasma Etching can be used. Here the IC is immersed in a gas plasma, and by altering the composition of the plasma, selective removal of different materials from the chip's surface is possible. Finally, Ion Beam Etching enhances dry plasma etching by enabling the attacker to focus the etching rather than the all-or-nothing option offered by immersive etching.

The ability to disassemble and, in effect, re-wire a chip means that no chip is safe from the attacker when money is no obstacle. Kerckhoffs's principle [97] states that *"A cryptosystem should be secure even if everything about the system, except the key, is public knowledge"*. This remains true right down to the gate level of the ICs used.

2.3 Errors

The errors of interest are not faulty algorithms or program implementation bugs. They are abnormal behaviour in response to an external stimulus that makes a device do something it was not programmed to do. Referred to as glitches or transient faults, they can be deliberately induced and exploited to compromise, what would otherwise be, a secure device.

2.3.1 History of Fault Attacks Techniques

The recognition that transient faults may occur during program execution is not new. The risks associated with these faults are widely recognised in industries working on safety-critical devices and equipment designed to operate in hostile environments. Referred to as Single Event Upsets (SEU) they occur randomly and are induced by unpredictable external events.

Energetic charged particles can momentarily affect the electron distribution at a semiconductor junction, induce momentary current flows and create an effect akin to switching a transistor. The three primary sources of these particles are [25]:

- Alpha particles emitted from radioactive impurities in a device's packaging materials.
- Terrestrial radiation in the form of high-energy neutrons.
- Cosmic background radiation.

The last two mechanisms induce errors in silicon circuits when neutrons, colliding with atomic nuclei in close proximity to the device, cause nuclear fission and release charged particles. The probability of experiencing SEUs depends on the deployment environment, the size and packing density of individual transistors, and the overall size of the device [24]. This probability is called the Soft Error Rate (SER), and as ICs get bigger, containing ever more transistors, the threat from SEUs increases.

Chip manufacturers, recognising the risk, provide guidance datasheets on the subject for engineers [6, 44].

There have been attempts to simulate SEUs to investigate their impact and to test defensive strategies [81]. Many of the attack techniques used by cryptographers have been borrowed from, or are reinvented, techniques used for radiation-hardness testing [141, 67]. Likewise, the defences, in the form of self-testing software, have precedents in fault-tolerant computing research. Trends towards voltage and geometry shrink increase the SER, and the topic is gaining in everyday relevance outside the defence and space industries [144, 180]. The latest chips are so big and sensitive they are vulnerable to SEUs at home and at sea level.

The prospect of deliberately inducing such errors was not openly discussed in the smart card and cryptographic community until 1996 when Boneh, DeMillo & Lipton published the groundbreaking paper, 'On the Importance of Checking Cryptographic Protocols for Faults' [36]. Within a year, Biham & Shamir [33] coined the phrase, Differential Fault Analysis (DFA), and demonstrated that most cryptographic algorithms were vulnerable to attack if an attacker could induce an error during their calculations. Both parties speculated on the likelihood of being able to generate such errors in practice, and these reservations were backed up by RSA Laboratories, who also suggested in a special report that *"potential vulnerabilities are primarily of theoretical interest"* [92]. Within a year, researchers were demonstrating practical error attacks [7]. The cryptographers had discovered the engineers' work, and the cat was out of the bag.

2.3.2 Consequences of Faulty Execution

Does it matter if something goes wrong momentarily when executing a program? Data will be invalidated, so the question should be, can we learn anything from faulty execution?

The significance is clear for algorithms relying on modular exponentiation. The Rivest–Shamir–Adleman (RSA) algorithm [147] is relatively simple to understand and is the most widely used asymmetric cryptographic algorithm [108].

The RSA components are shown here.

$$
\begin{aligned}
P \,\&\, Q &= \text{secretly chosen prime numbers} \\
\phi &= (P-1) \times (Q-1) \\
e &= n \mid n \ \in \{1 \ldots \phi - 1\}, \text{ and } n \ \perp \ \phi \\
d &= e^{-1} \ mod \ \phi \\
N &= P \times Q
\end{aligned}
$$

The key owner publishes e and N as the public key. The other components are retained as the private key. Generating a key involves finding two large prime numbers P & Q. Once they have been found, it is easy to calculate the other components. The system's strength is based on the presumed difficulty of factoring N to recover P & Q. As computers get faster and cheaper, it is easy to increase the security of RSA by choosing ever-larger values for P & Q.

$$
\begin{aligned}
Signature &= Message^d \ mod \ N \\
Message &= Signature^e \ mod \ N
\end{aligned}
$$

The holder of the private key can sign a message. Anybody with the public key can confirm that the private key holder signed that message, but they cannot use the public key to construct the signature. This asymmetry is what makes RSA powerful and convenient to use.

e is not secret and is usually chosen to be relatively short. Consequently, the computation of $Signature^e \ mod \ N$ is quick. d is usually in the order of several thousand bits long and, $Message^d \ mod \ N$ can therefore take a long time to compute.

The most common optimisation for signature generation, or decryption, is based on the Chinese Remainder Theorem (CRT). A signature can be generated via CRT approximately 4 times faster than by the full length exponentiation using d & N.

Algorithm 4: RSA Signature generation (decryption) via CRT

Key Data: $P,\ Q =$ Secret prime numbers,
$D_p = e^{-1}\ mod\ (P-1)$,
$D_q = e^{-1}\ mod\ (Q-1)$,
$U = P^{-1} Mod\ Q$.
Input : $X =$ Message to Sign
Output : $Z = X^d\ mod\ N$

1 $S_p \leftarrow X^{D_p}\ mod\ P$
2 $S_q \leftarrow X^{D_q}\ mod\ Q$
3 $Z \leftarrow (((S_q - S_p) \times U)\ mod\ Q) \times P + S_p$

The so called Bellcore attack by Boneh et al. [36] is possible if the attacker can obtain two signatures of the same message, S and $\hat{S}$, where $\hat{S}$ results from an erroneous calculation during step 2, shown here in Algorithm 4. If this is possible, the attacker can retrieve P & Q, the private key.

$$P = gcd(S - \hat{S}, N),\ and$$
$$Q = N \div P$$

An optimisation on this attack by Lenstra [107] recovers the key from a single erroneous signature and works where an error occurred during either step 1 or step 2 of Algorithm 4.

$$P = gcd(X - ((\hat{S}^e\ mod\ N), N)$$

It is also possible to recover the keys from symmetric cyphers. Take the last two rounds of DES, for example, Figure 2-10, and consider the consequence of an arithmetic error during any of the operations highlighted in green. Such errors affect L_{16} and R_{16} can be recovered from the *Ciphertext* output.

Three conditions are required to perform the attack.

1. We must be able to repeat the calculation with the same input data. It is not necessary to know that data, only that it is the same for each run.

2. We also need to see the resulting *Ciphertext* output from which we can recover L_{17} & R_{17} by performing FP^{-1}.

3. We need to induce a fault during at least one DES operation during its 15th Round.

Let the round function, $\quad F(r,k) = C(k \oplus E(r))$

We know,

$$L_{17} = F(R_{17}, k_{16}) \oplus L_{16}$$
$$\hat{L}_{17} = F(\hat{R}_{17}, k_{16}) \oplus L_{16}$$

Or alternatively, $\quad L_{17} \oplus F(R_{17}, k_{16}) = L_{16} = \hat{L}_{17} \oplus F(\hat{R}_{17}, k_{16})$

Now we can search for all the possible values for k_{16} that satisfy $L_{17} \oplus F(R_{17}, k_{16}) = \hat{L}_{17} \oplus F(\hat{R}_{17}, k_{16})$. Thanks to the S-Boxes' organisation within the compression func-

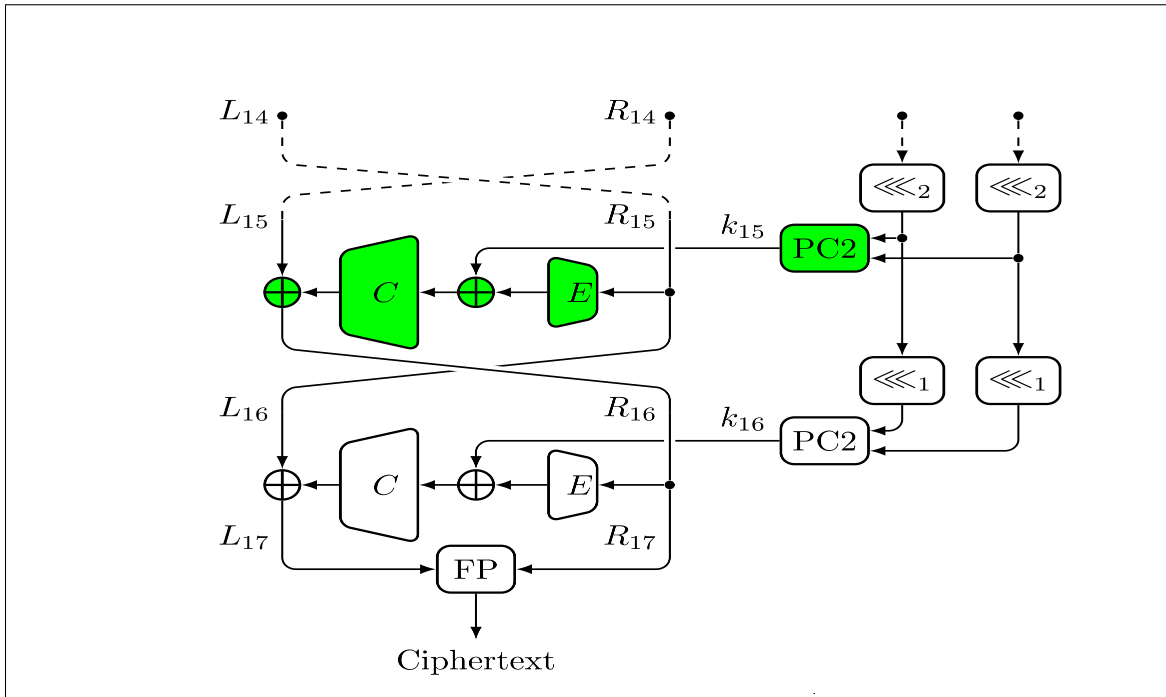


Figure 2-10: DES Final Rounds

tion C, See Figure 2-6, this search can be performed 6 bits at a time. By only considering those bits in L_{17} that are influenced by permutation P on an individual S-Box's output. $P(0xF0000000)$ identifies the bits derived from S-Box$_0$, $P(0x0F000000)$ S-Box$_1$, etc. We can quickly identify which values of each S-Box's 64 possible inputs are consistent with the observed results. Repeating this for each S-Box in turn, brings the total search space down from 2^{48} comparisons to a more manageable $2^6 \times 8$, or 2^9, comparisons. The process produces a list of possible candidates for each 6 bit component of k_{16}, and these lists can be reduced further if more erroneous samples are available. The process is shown here in Algorithm 5.

Algorithm 5: DFA DES round 16.

Key Data: $C[8, 64]$ = Candidate array, initialized to $'Valid'$,
R_{17} = Observed result 1,
$\hat{R}_{17}$ = Observed result 2.

```
1 for c ← 0...63 do
2     KeyCandidate_[0...47]_b = c_[0...5]_b || c_[0...5]_b || ··· || c_[0...5]_b
3     L = F(R_17, KeyCandidate)
4     L̂ = F(R̂_17, KeyCandidate)
5     for i ← 0...7 do
6         mask = P(F0.00.00.00_h ≫ (i × 4))
7         if (L ∧ mask) ≠ (L̂ ∧ mask) then
8             C[i, c] ← 'Invalid'
```

The complete set of candidates for k_{16} is found by combining the $Valid$ entries for each S-Box$[0 \ldots 7]$ from the array C. Our experimental observations have shown that a single S-Box error is enough to bring the full candidate search space down from 2^{48} to 2^{43} comparisons, and an error affecting two S-Boxes brings the search space down to 2^{35} comparisons. Sixteen samples of single S-Box errors, or eight examples of a double S-Box error, is enough to identify a subkey uniquely.

Each subkey uses 48 bits of the whole key's 56 bit key field. This leaves just 8 bits, or 256 variants, to test via brute-force to recover the whole key.

Since the first description of these attacks, the original DES standard has been superseded with the Advanced Encryption Standard (AES). The Rijndael algorithm was selected as the new AES because it is relatively is easy to implement with hard-

ware and appeared, at the time, to be resistant to DFA. Unsurprisingly, given the ingenuity of cryptographers, there are now many published examples of DFA attacks on AES [34, 51, 63, 74, 137]. There are even optimisations of existing attacks [76], and attacks on implementations utilizing duplicate/redundant hardware [159].

Not all attacks on Cryptography require exploitable mathematical errors. The basic administrative tasks of feeding data and keys into cryptographic functions are also fraught with risks. Consider the highly simplistic *Key Nulling* scenario where a software loop may be prematurely terminated by an error [17, 178] shown in Table 2.1. If an attacker can control the number of bytes copied for the key, then by performing encryption first with one copied/unknown byte, then with two, the attacker can search all 256 candidates for each byte. The brute force attack on an n byte key takes $256 \times n$ trial calculations instead of the expected 256^n.

Table 2.1: Key Nulling Exploit

Input	**Key**								**Output**
$M \rightarrow$	$K_0 = XX_0$	00	00	00	00	00	00	00	$\rightarrow C_0$
$M \rightarrow$	$K_1 = XX_0$	XX_1	00	00	00	00	00	00	$\rightarrow C_1$
$M \rightarrow$	$K_2 = XX_0$	XX_1	XX_2	00	00	00	00	00	$\rightarrow C_2$
$M \rightarrow$	$K_3 = XX_0$	XX_1	XX_2	XX_3	00	00	00	00	$\rightarrow C_3$
$M \rightarrow$	$K_4 = XX_0$	XX_1	XX_2	XX_3	XX_4	00	00	00	$\rightarrow C_4$
$M \rightarrow$	$K_5 = XX_0$	XX_1	XX_2	XX_3	XX_4	XX_5	00	00	$\rightarrow C_5$
$M \rightarrow$	$K_6 = XX_0$	XX_1	XX_2	XX_3	XX_4	XX_5	XX_6	00	$\rightarrow C_6$
$M \rightarrow$	$K_7 = XX_0$	XX_1	XX_2	XX_3	XX_4	XX_5	XX_6	XX_7	$\rightarrow C_7$

A similar threat exists when returning the final answer to a calculation. Injecting an error into loop variables may result in the delivery of unintended data. When it is possible to corrupt the buffer's location or size, a poorly implemented loop will dump arbitrary data from an IC's memory. This data may contain intermediate working variables from an earlier calculation, keys or other secret data. This is one of the most uncomplicated error attacks and is commonly used during EMV bank card penetration testing as part of the scheme approval process required before permission is granted for deployment. In this case, the Get Processing Options command [65] is targeted because it delivers a block of data from memory and can be repeatedly

exercised without any requirement for passwords or cryptographic verification.

Virtual machines are compromised by errors too. The erroneous execution of an interpreted instruction is enough to break the primary security claim of JavaCard. JavaCard's security model relies on the concept that the executing programs have been *Bytecode Verified*. Bytecode verification is a static analysis of a whole application, ensuring all variables and functions are used in ways appropriate to their declared data types. For example, it prevents arithmetic on object references or the casting of numeric data into object references. If a single instruction can be skipped or made to malfunction, then the JavaCard security model's foundation is broken.

```
1  byte[] baTmp;
2
3  byte Foo(byte [] baX, short s) {
4      return baX[s];
5  }
6
7  byte Bar(short s1, short s2) {
8      return Foo(baTmp, (short)(s1+s2));
9  }
```

Figure 2-11: Java Source Code

In this example, Figure 2-11, the bytecode verifier ensures that function `Foo()` is only invoked with a reference and a short in the parameter stack. In function `Bar()` we see `Foo()` being invoked correctly. This code will pass the bytecode verification process, and a basic JavaCard Virtual Machine will execute it without further checks.

```
1   .method  Bar(SS)B 129 {
2     .stack  4;
3     .locals 0;
4
5     L0: aload_0;
6         getfield_a_this 0;  // ref BcDemo.baTmp
7         sload_1;
8         sload_2;
9         sadd;
10        invokevirtual 5;    // Foo([BS)B
11        sreturn;
12  }
```

Figure 2-12: Java bytecode

The compiled bytecode for function `Bar()` is shown in Figure 2-12. Line 6, pushes the reference onto the parameter stack. Lines 7 & 8 place `Bar()`'s own parameters on the stack. They are removed by the addition operation, Line 9, and replaced with

their arithmetic sum. The stack now contains the reference and the short expected, and required, for the call to `Foo()`.

If the addition on Line 9 could be avoided or corrupted, then the invocation of `Foo()` could occur with two shorts at the top of the parameter stack. `Foo()` would be unaware of the error and treat the first short as the byte array reference it was expecting.

Historic attacks on JavaCard that relied on bypassing the off-card bytecode verifier and loading malformed bytecode were effectively prevented when JavaCard V3 (*Connected Edition*) implemented on-card bytecode verification. Unfortunately, runtime error induction, as shown above, enables these attacks to be reinvented as a combined physical and logical attack [19, 18]. Other researchers have used memory errors to make a Java Virtual Machine (VM) use faulty references and execute arbitrary code [78]. In recognition of these risks, modern JavaCard VM implementations perform more runtime sanity checking of their execution state [5, 102].

In general, fault attacks on secure embedded software present a high risk of exploitation. This threat is underestimated by many programmers as there is an instinctive assumption that μCs execute programs accurately and repeatably.

There is also no advantage in heavily defending a cryptographic operation if a simple error can ignore the result of that calculation. If an operation can proceed, irrespective of an authentication state, then that application is vulnerable regardless of the authentication algorithm's quality. Multiple studies have arrived at this same conclusion [75, 188], while also noting that defences have costs and need to be deployed sparingly.

2.3.3 Mechanisms of Fault Injection

We have seen that faults in both data and execution of μCs can have serious effects on the security of schemes that rely on them. The question remains, does this matter? In much the same way as a meteor impact could be devastating for all life on earth, no one will lose much sleep over the prospect because it is exceedingly unlikely. Unfortunately, the same is not true of errors in μCs. They are far more likely, both

as spontaneous events as in the SEU discussed previously and as deliberately induced events initiated by hackers.

Here we look at the catalogue of techniques available to the hacker, observe the ease with which they can be deployed, and the ease with which the effects can be exploited.

2.3.3.1 Glitch Attacks

The principle behind power and clock attacks is relatively simple.

All ICs have an operating range for the supply and signal voltages. Manufacturers list these on datasheets, and circuit designers ensure the ICs are supplied with the appropriate signals to ensure their product's reliability. If the voltage is too high, the IC may burn out as too high currents flow through the device. If the voltage is too low, the threshold for switching logical 0s & 1s is not met, and the device fails to operate. In a glitch attack, a voltage, high or low, may be applied momentarily for a short enough time to avoid damage to the IC and for long enough to make some internal behaviour fail. The IC then continues its normal behaviour but with some registers or other subcomponents in corrupted states.

Likewise, digital ICs have maximum operating frequencies described in the component's datasheet to guide circuit designers. Depending on the operating voltage, semiconductor material used, and the transistor size within an IC, a transistor's switching time varies. These switching times accumulate when one transistor's output feeds another's input. The sum of these delays is the signal's propagation time. In clocked digital circuits, a device will typically latch one state on a clock edge; this new state will trigger a cascade of logic resulting in a new state that is then latched on the following clock edge. The limit for the device's clock signal frequency is the time of the longest propagation delay. By creating a clock cycle shorter than this limit, the attacker can expect the device to latch some internal states before they have been fully calculated. Once again, the IC then continues its normal behaviour but with some registers or other subcomponents in corrupted states.

Glitch attacks are applied externally to an un-tampered device. They work be-

cause the device has been taken out of what has been termed its *Operating Envelope* [184]. They are relatively easy to implement at a very low cost in terms of components and equipment. The disadvantage is that the glitch effect is not localised, and the attacker has little control over what aspects of the device are affected. This imprecision can lead to an overwhelming number of errors and no usable effect or errors within features that are of no interest to the attacker. It is also possible that some errors may hinder the collection of valuable errors. For example, an error that renders the communication channel inoperative will make other internal errors unobservable.

Attempts have been made to identify the thresholds where errors begin to occur by fine manipulation of the parameters, timing, duration and magnitude. The suspicion is that a μC's attempts to access memory are most vulnerable. While register to register operations are the most stable [20]. This and other studies of glitch attacks have concluded that injecting a fault is relatively easy but that injecting an exploitable fault is hard [16]. While this is undoubtedly true, it must be remembered that the technique was successfully used in the criminal world of Pay-TV hacking before academic interest emerged [7]. The use of glitches is widely reported [132, 137, 17, 94], although it should also be noted that it is primarily in the context of defences rather than examples of new exploitations. Modern security devices, such as those used in smart cards, implement hardware defences against these attacks. This technique remains a potent threat for other μCs, lacking such defences. The ease and low cost of implementation mean this practical attack is available to many potential attackers.

The third type of glitch attack worthy of mention is heat. Again operating temperature is part of the *Operating Envelope* characterised by manufacturers and listed on a typical datasheet. It is not possible to rapidly apply and remove heat as it is with electrical inputs, but researchers have shown that at specific critical temperatures, faults start to occur. On their μC of choice, Atmel's AVR, they witnessed increasing numbers of faults occurring between 152 & 158 °C [85]. The evidence suggests that meticulous control of temperatures within this range may be exploitable.

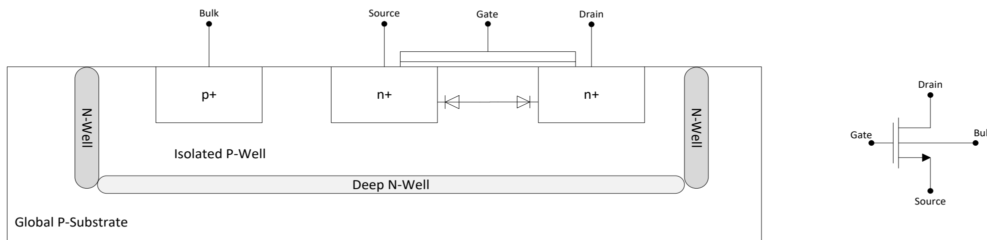


Figure 2-13: Body Bias Transistor

2.3.3.2 Body Biasing Injection Attacks

Body Bias is used in low-power CMOS designs to adjust the threshold voltage of transistors to enable them to work with lower supply voltages [143].

In this configuration, the transistor has four terminals, Figure 2-13. As with an ordinary transistor, applying a voltage to the Gate draws charge into the depletion zone between the Source and Drain. At a critical gate voltage, this breaks down the high resistance formed in the n-p-n junction and enables a current to flow. The fourth terminal, Bulk, biases the p-doped substrate and affects the voltage threshold for transistor switching. Isolation of regions of the chip with negatively doped barriers enables different logical components within the IC to operate with different switching characteristics. Operating with smaller voltages reduces the current, and consequently, components can run faster and use less power.

The Bulk terminal affects a region of the IC and many transistors. This opens up the possibility of influencing this signal and affecting the transistors, not by changing the Gate or switch signals, or by inducing current flow between Source & Drain, but by changing the point at which the sub-component switches state. A probe touching the rear side of the IC with a momentarily applied voltage is enough to achieve this effect [175]. Pulses of 160 V for 200 ns have been shown to generate localized effects with approximately 200 µm resolution [31]. This enables the attack to be focussed on specific areas of the IC rather than the system-wide hit induced by clock or voltage glitches. This attack vector bypasses the defences used insecurity chips for voltage or clock glitches while generating similar effects.

2.3.3.3 EM Attacks

Electro Magnetic perturbations are induced by generating powerful electromagnetic waves in close proximity to the running IC. For example by using a spark generator [156]. Alternatively, a tiny wire loop antennae driven with an electrical pulse can be used to generate a momentary magnetic field [129]. Changing magnetic fields induce currents in nearby conductors, currents can switch transistors, and unexpected switching of transistors generates errors.

Two types of EM perturbations have been successfully used to attack ICs; harmonic emissions, where an oscillating field is used, and transient pulses, where the momentary application of a powerful pulse is applied to an IC.

Harmonic emissions of approximately 1 GHz have been used to influence the operation of stand-alone ring oscillators. Such oscillators are often used within ICs as a source of random noise and used in random number generators [27, 58]. Reliable, unbiased random numbers are critical for secure cryptographic protocols. Therefore, this technique is a significant threat to devices that use ring-oscillators to generate so-called 'true' random numbers.

Transient pulses of EM radiation have been used to induce errors in executing μCs. Single byte memory look-up errors have been observed and used to replicate the oft-cited Bellcore attack [156], described earlier on Page 71.

EM attacks are equally effective when performed from either the top-side and rear-side of an IC [156]. The technique does not have the same spatial resolution as body-bias attacks, but it is still highly effective at generating computation errors. It is often cited as a source of error in papers describing cryptographic attacks based on DFA, for example, Dehbaoui's attack on AES [58].

2.3.3.4 Light Attacks

Light attacks exploit the *Photovoltaic effect* in the semiconductor.

Conduction in solids depends on electrons' ability to move between the valence band and the conduction bands within an atom. When these bands overlap, no

additional energy input is required for electrons to change bands, and the material will be a conductor. Conversely, when the energy required for an electron to switch bands is large, we have an insulator. Semiconductors have a small but distinct band-gap and require modest energy input to accelerate electrons from the valence to the conduction band.

In silicon, the energy required to cross the band gap is 1.1 eV at 300 K and is the energy of a single photon with wavelength 1130 nm. This is the photoelectric effect, and 1130 nm corresponds to near-infrared light. All light sources from near-infrared through visible can generate this effect in silicon.

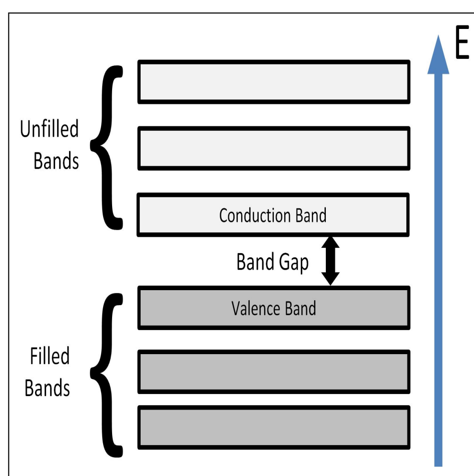


Figure 2-14: Semiconductor Band Structure

The promotion of an electron to the conduction band creates an electron-hole pair within the semiconductor. If no electric field is present, the non-equilibrium will recombine with no net flow of charge carriers. If electron promotion occurs within a reverse-biased pn junction, the holes migrate to the p-region and the electrons to the n-region. This is known as Optical Beam Induced Current (OBIC) [171].

Within a powered transistor, OBIC has the same effect as applying a voltage to its gate. If enough electrons are excited, then the gate will switch. Thus the application of light to a transistor can make it switch without reference to its controlling gate's input voltage. Therefore it is possible to use light to induce an error in a silicon circuit. Smaller components require smaller OBICs, so modern ICs are potentially more vulnerable to this effect than older designs.

Sustained application of light may cause circuits to burn out. See Figure 2-1; here, the two gates should be in opposite states when controlled by the V_{in} signal. When switched by photons, they could simultaneously be open, leading to a short circuit and possible burnout of the devices. Pin-point application of light can be used to selectively destroy components [171].

The momentary application of light has a similar effect as an externally applied glitch. It causes transient non-fatal errors and can be achieved with a simple photographic flash-gun. The technique was first described publicly by Skorobogatov [166] but had been widely used in industry for many years before disclosure.

Lasers offer a significantly more precise mechanism to inject light into an IC. Finer control of both focus and power is possible. A laser focussed through a microscope's objective lens enables an attacker to induce OBIC in single transistors within a large IC, and the relative ease of control have made lasers the weapon of choice for most sophisticated attacks based on error induction.

2.3.4 Lasers

Lasers provide a very flexible and precise way to inject errors into an IC. Being a single wavelength light source, they can be accurately focussed with relatively simple optics. Their use in error induction is almost as old as the semiconductor industry itself [81]. This early interest was driven by the defence industry, seeking reliability, particularly in harsh environments.

It had been argued that the complications and costs of mounting an attack with lasers were in themselves defences. The capital cost of the equipment and expert knowledge required to mount an attack meant that low-value targets were not worth attacking [21]. However, relatively powerful semiconductor lasers have emerged on the market to support telecommunications (fibre optics) and the DVD/BlueRay market. These combine ease of control with sufficient power to induce OBICs, and do so at an almost trivial cost.

2.3.4.1 Implementation Practicalities

A typical laser fault injection station consists of a microscope with a laser light source mounted so that the beam passes through the objective lens onto the target. When using a visible light laser, the focus will coincide with the optical focus obtained by looking through the eyepiece. Ideally, the target will be placed on stage with fine

control available for both X & Y coordinates. The workstation used in this study is seen here in Figure 2-15.

A mechanism is required to trigger a laser pulse. This usually requires continual monitoring of the target's power consumption in the same way that power analysis attacks monitor the same property. Real-time pattern matching is then required to identify the moment to fire the laser [178]. If the event of interest has no discernable power profile, it may be necessary to insert a precise delay between a recognised pattern and the laser trigger event. Knowledge of the algorithm being executed can help identify patterns of interest within a power trace. For example, a rapid series of 10, 12, or 14 power surges may identify successive AES round operations. Identifiable but nondescript patterns before such land-mark patterns may be related to key loading or data preparation.

Computer control of the X-Y stage enables the automation of a scanning campaign. It is usually necessary to attempt injection on all possible sites to identify the sensitive regions worthy of further detailed investigation. This process can be laborious and time-consuming, so automation is practically unavoidable.

Figure 2-15: Laser workstation

The top side of an IC is built up of multiple layers of metal tracks, insulation and optional protective layers. This shielding can obscure the sites of interest for optical attacks requiring line of sight access to the silicon. For precision attacks on individual transistors, it can render top-side attacks infeasible. Silicon is semi-transparent to light in the Near Infra Red (NIR) part of the spectrum. It is even possible to photograph the circuit from the back-side [30]. A laser with a wavelength greater than approxi-

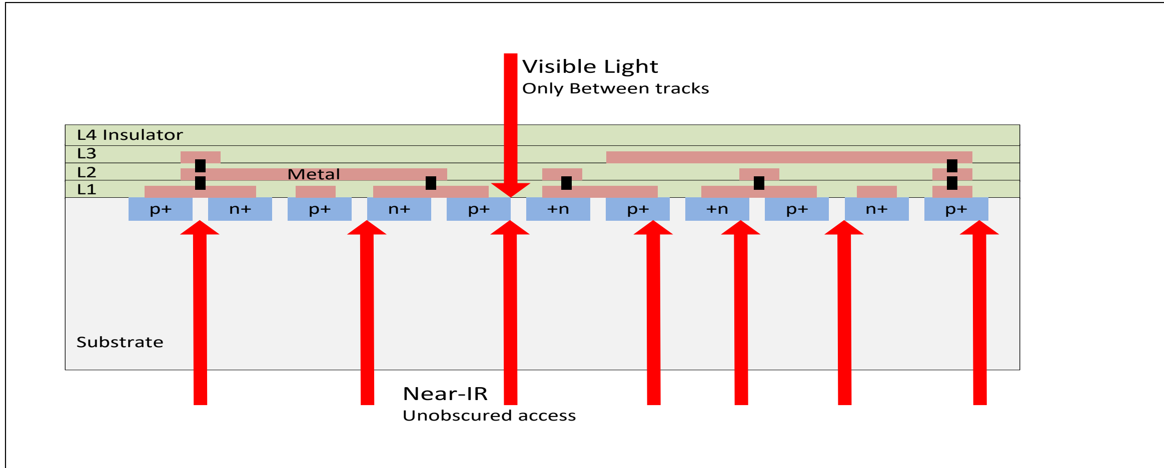


Figure 2-16: Frontside and Rearside Attack

mately 1000 nm will penetrate the silicon and provide unobscured access to all of the ICs components. See Figure 2-16.

In many cases, access to the backside of the IC is less complicated than the front side. For ICs with many connections to external pins, a common manufacturing process is bump-bonding and flip-chip packaging. This involves building up a small conductive *bump* in the terminal pads, flipping the chip over and pressing it against the carrier body. No wire bonding is required, and the back of the IC is left facing outwards. This process makes it easier to attack the chip without needing to re-bond connections but is only practical using NIR [179].

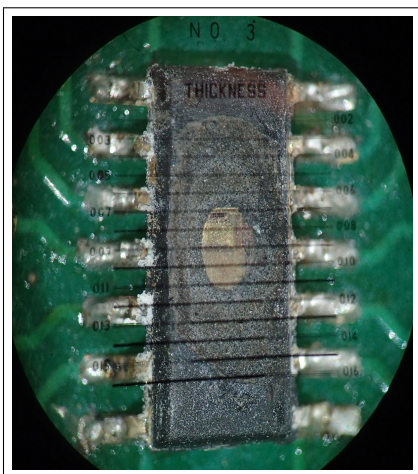


Figure 2-17: Etched Package

It is necessary to expose the surface of the IC to perform laser fault injection. The widely used technique is to dissolve the packaging with hot fuming nitric acid ($> 98\% NHO_3$) and wash it with acetone to remove any residues. This removes the plastic without damaging the IC [156]. For ICs in the familiar black plastic packaging, it is more efficient to mechanically grind away some of the plastic to form a well above the site of the IC. Repeated etch and clean cycles can be used to expose the IC while leaving it supported in the package and connected to the package's pins, Figure 2-17.

We have also observed that some devices use copper bond wires to connect the IC to the pins. For such samples, it is necessary to stop NHO_3 etching as soon as these wires are exposed because they quickly corrode. We found it possible to resume etching at this point with concentrated sulphuric acid ($95-99\% H_2SO_4$). It is slower and requires more etch and rinse cycles, but it leaves the copper undamaged. Others recommend diluting the NHO_3 with H_2SO_4 as this reduces the risk of damaging aluminium structures [29]. The alternative is to entirely remove the IC and re-bond it to new contact points, as shown in Figure 2-18.

When access to the rear side of the IC is required, the whole process of decapsulation can be performed mechanically by grinding away any packaging until the chip is exposed. There is often a copper or aluminium plate protecting the rear side of the chip. This metal can be removed with tweezers and a little bit of force without damaging the device. A problematic aspect of rear-side attacks is aligning the laser to areas of interest. Without recognisable surface features, it is necessary to rely on a visible mark or blemish on the chip and precise X-Y stage movement to reacquire targets.

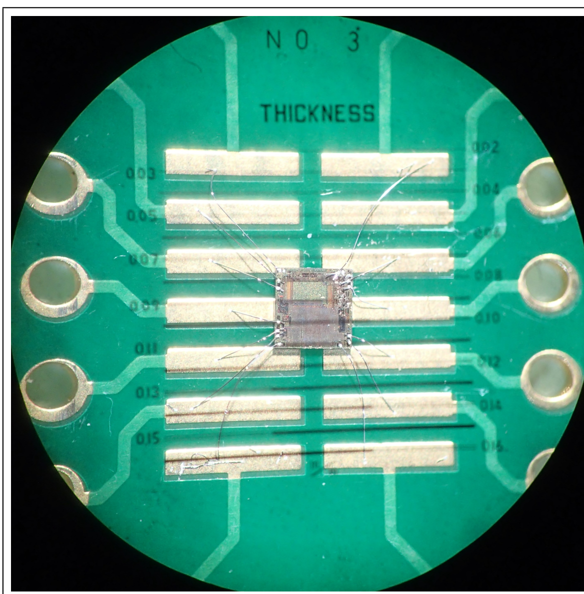


Figure 2-18: Rebonded IC

Unless the microscope supporting the laser is equipped with a filter to protect an observer's eyes from laser light reflected out through the eyepiece, it is advisable to replace an eyepiece with a CCD camera and observe the target remotely via a monitor screen. Likewise, the area surrounding the target should be screened off to avoid accidental exposure to the light source.

2.3.4.2 Static - Bit-Flip

The first reported effects of pin-point laser attacks on an IC related to changing its memory contents. By targeting the transistors making up an SRAM cell, it is possible to set or reset its stored data bit [166].

A typical CMOS SRAM cell comprises 6 transistors, and activating any of them can lead to a state change. In Figure 2-19, transistors P1, N1, P2, N2 form a simple bistable circuit, or flip-flop. Activation of P1, or N2 will set the state to 1, while activation of P2 or N1 will set the state to 0. Using WL to simultaneously activate N3 and N4 will make the component adopt the state currently asserted on the BL lines.

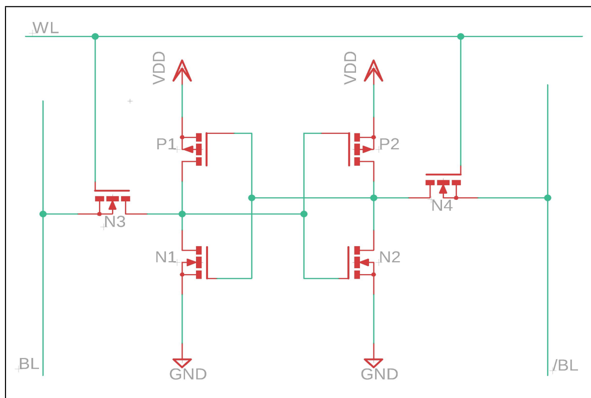


Figure 2-19: Static RAM, Single Bit

Using a highly focussed laser to stimulate individual transistors enables editing of the memory contents. This has been termed '*surgical fault injection*' [64]. Setting and resetting are possible, but logically inverting a bit would appear to be infeasible [151]. However, reproducible bit inversion has also been demonstrated by Agoyan [3]. It has

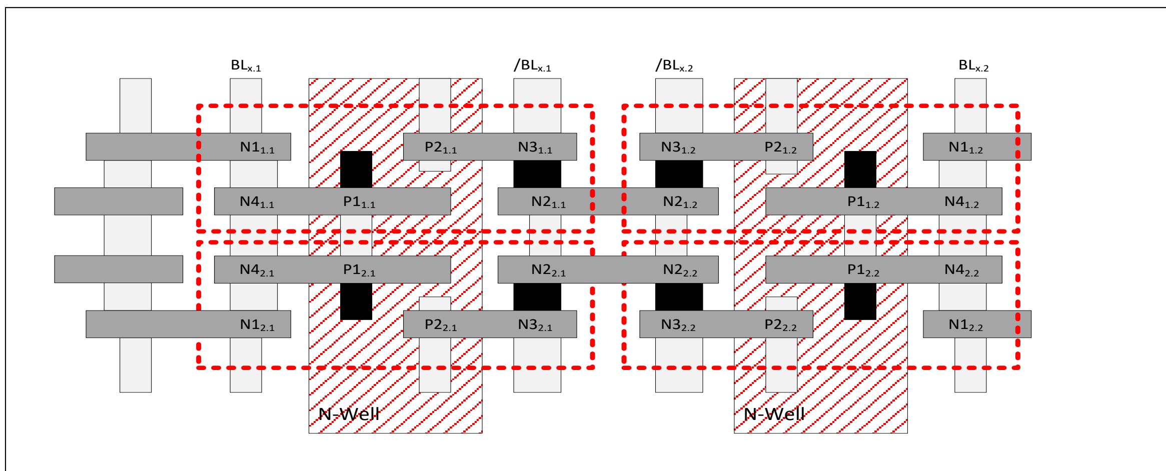


Figure 2-20: Simplified SRAM Layout

been postulated that the effect arises as a consequence of exciting just one of a transistor's wells, with the effect of inverting an inverter. Regardless of the mechanism, it is an interesting result as it guarantees data corruption and increases the efficiency of a DFA attack.

RAM is one of the most readily recognisable structures in an IC. Using 6 transistors per bit, and 8 bits per bytes, a typical μC's RAM or register bank shows a highly repeated pattern. Figure 2-20 shows how high density and regularity is achievable by making each cell a reflection of its near neighbours. This gives a 4 bit repeating tile that is easily recognisable in Figure 2-21 High density and regularised surface metal features mean it is usually difficult to attack RAM from the top side. Instead, for accurate bit editing, a rear-side NIR attack is required.

Figure 2-21: AVR Registers

By using a highly focussed laser beam, Courbon has mapped the locations on an IC where the laser will set or clear individual memory bits [55], effectively mapping the individual transistors in the RAM. These maps demonstrate the alternating pattern of cell layout and the relative sizes of the PMOS and NMOS transistors. Control over the energy delivered by the laser pulse has also demonstrated that on his target IC, $1 \rightarrow 0$ transitions require significantly less power than $0 \rightarrow 1$. This observation raises the possibility of setting whole registers or memory regions to 0 with a single precisely powered light flash over a specific area.

2.3.4.3 Laser Error Induction in a Running Program

Corrupting the execution of a program is the logical extension of tampering with data in memory. Corruption of the individual instructions will lead to program execution errors, and inducing corruptions while the instructions are being processed or in the pipeline means the effect can be temporary. This gives the possibility of injecting

errors into individual rounds of a loop without affecting every round as a memory-resident corruption would do.

Unlike memory editing, timing is critical, requiring synchronisation between the executing program and the laser pulse trigger. In much the same way as with glitch errors, the reading of data or program instructions is momentarily disrupted and synchronising such events with critical operations leads to the possibility of manipulating a program's flow. The significant advantage given by using a laser is that the error location, and presumably, its effect, can be specifically chosen.

Program flow has received less attention than memory editing [148], but the consensus is that the effects are reliably repeatable. The effect has been demonstrated on multiple CPU architectures, the most common being AVR and ARM. The AVR has a one instruction pipeline, and the dominant effect has been characterised as instruction skipping [40, 96]. The ARM7m has a 32-bit instruction fetch, collecting a block of up to four instructions. Here the effect appears to be the repetition of the preceding block of instructions [148]. Execution errors have differing effects on different μC architectures, and in this respect, they differ from memory editing induced errors.

2.4 Defence Techniques

Defence techniques fall into two categories; preventing errors by physically obstructing attacks and detecting errors to enable an attacked device to make a defensive response.

Placing a physical barrier layer on the top surface of an IC is a relatively simple physical defence for a μC. In its most basic form, it is relatively easy to etch and remove. An alternative approach is the *active shield*, where a grid of wires is placed over the surface, and simple electronics then detect broken wires or the short-circuits between wires. Regularities in, or predictability of, the track layout has been recognised by some attackers and subsequently bypassed using a FIB to cut and connect individual tracks selectively [42]. These defences significantly increase the cost of attack but do not stop a determined, well-funded attacker.

While shielding the top side of an IC can protect a device from top-side attack by all but the best-equipped laboratories, it still leaves the backside vulnerable. Physical barriers on this side can be mechanically ground and polished away without risking damage to the active components on the top side.

Constructing transistors on top of a layer of insulator has two beneficial effects, Figure 2-22. Performance is improved as the substrate's parasitic capacitance is reduced, and light attacks are made more difficult. The most common insulator is silicon dioxide, SiO_2. It has a significantly different refractive index to silicon (≈ 1.46 vs ≈ 4.5).

Light injected from the back is more likely to be internally reflected, and any light through the silicon-insulator-silicon sandwich will also be significantly defracted [50]. Other insulators include zirconium dioxide (ZrO_2) and saphire (Al_2O_3). All increase the manufacturing cost of the devices, and none of them prevent rear-side laser attacks. They only make attacks more difficult to achieve [179].

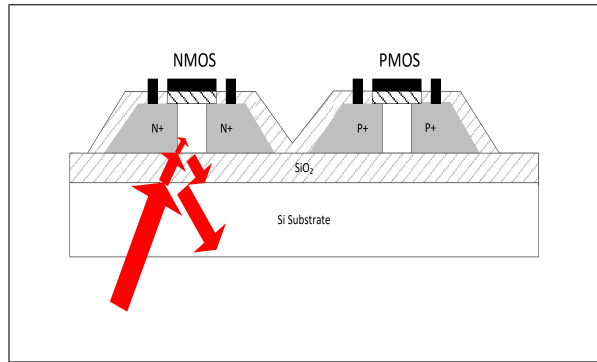


Figure 2-22: Silicon on Insulator

Defences against non-invasive glitch attacks again add expense and complication to an attack. Voltage and temperature sensors can identify when the μC operates outside of the specified parameters and place the chip into a non-functioning hibernation mode until it is reset. Frequency detectors can similarly recognise momentary over-clocking. An additional common defence for clock manipulation is to provide an internal clock or oscillator uncoupled to the externally supplied signals. Collectively these defences prevent non-invasive error injection attacks but are ineffective against semi-invasive attacks.

The same physical properties of a semiconductor junction that cause errors in integrated circuits can detect light. These *photodiodes* can then signal the presence of light and indicate that the chip is not operating in its expected environment, i.e. it is probably under attack. A photodiode can operate in either of two modes. If

a reverse bias is applied to the diode junction, it will conduct in the presence of light; this is the photoconductive mode. In the absence of a bias voltage, the diode acts in photovoltaic mode. The photovoltaic mode has a lower *dark current* and is more efficient in normal operating conditions, while photoconductive diodes have faster response times at the expense of a higher dark current [9]. Variations on the photodiode are the phototransistor and photoFET. These configurations can increase the photodiode's sensitivity but have a larger surface footprint, occupying more of the silicon surface, ultimately increasing the device's size. While these devices are good at detecting simple light-flash glitch attacks, a well-focused laser beam that can target a single transistor [55] can also avoid exciting such detectors.

Duplicating hardware is another approach used by μC manufacturers targeting the high-security market [88]. Running both instances of a device in parallel and halting if ever one device's state differs from the other's, detects an attack on either of the twined devices. This approach is expensive in terms of silicon use and increases the device's power requirements. The repeatability witnessed for many error effects means duplicated hardware is vulnerable to a synchronised attack targeting the same feature on both components [159].

Laser-induced bit-flipping of memory can be defended against by including parity bits in each stored word. The technique is widely used in desktop computer memory, where the devices are typically run at very close to their maximum speed and are implemented as high-density DRAM, which is potentially more error-prone. This technique only detects errors when the corrupted word is accessed. It defends executing programs against real-time induced memory errors when a single laser is used to inject one fault at a time. An attacker, aware of the parity defence, could maintain the parity by accurately manipulating multiple bits in the same word, either by using two targeted pulses or by adjusting the focus to impact multiple bits per pulse [55].

The last, and most versatile line of defence, lies within the executing software. Code that can verify its state, and the results of its calculations, is needed. This defensive code requires discipline on the programmer's part, who must verify calculations and only disclose results when there is no evidence of faults during execution.

Currently, the only way to verify this has been appropriately implemented is via independent code review, where a knowledgable third party reviews code and confirms the appropriate application of defensive code. This code-review methodology is the approach taken for Common Criteria certification [53] and EMV scheme approval [66]. The process is both expensive and error-prone. It requires programmers and reviewers with rare skill sets and introduces delays into projects while reviews are performed.

2.5 Observations

Defences against side-channel attacks rely on intelligent implementation to ensure the execution path is uninfluenced by the processed data and ensure auxiliary hardware defences are appropriately activated. Techniques such as data blinding require software to perform data transformations that disguise the data values when they are stored in memory or their hamming weight while in transit. These defences require additional code, and additional code provides additional opportunities to introduce errors.

There are many ways to induce errors in a μC, and these techniques have evolved as quickly as the underlying hardware. A stand-alone μC is vulnerable to both the static changes of its stored data and dynamic changes that momentarily affect an executing program. Unfortunately, as the underlying technology has advanced, so too has its susceptibility to induced errors. Meanwhile, the availability of the tools required to mount an attack has similarly increased. These two factors significantly reduce the cost of mounting an attack and increase the range of devices worth attacking.

Exploitable errors in a μC rely on accurate timing of the error stimulus, and the techniques used in side-channel analysis provide this. Timing observations and SPA in particular are well suited to assist an attacker in recognising the appropriate moment to inject an error.

Physical defences have been shown to be expensive and, to a degree, ineffective. At best, they serve as an obstacle, increasing the cost of an attack; at worst, they offer

a false sense of security. In short, we must be resigned to the fact that an attacker can inject errors and choose the timing of those errors with some degree of accuracy.

The most basic software defences against errors involve repeating calculations, but this has the undesirable effect of increasing the device's vulnerability to side-channel analysis. The alternative of re-computation via an alternative algorithm is inefficient and provides an additional potential source of leaks. Whatever approach is taken, there is no point in performing repeat computations if the act of comparing the outcomes can itself be compromised.

To be exploitable, errors must be survivable. That is to say, the μC must continue to execute instructions and, in doing so, deliver information to an attacker. If the μC is executing after an error injection, then there is the possibility that programmed defences could recognise anomalous behaviour and react appropriately. This makes the software the last and most versatile line of defence; but, can the executing program be trusted to verify its own behaviour?

This is not a new problem.

"Quis custodiet ipsos custodes?" †

— Juvenal, Satire VI, 2nd century AD.

† Who will guard the guards themselves?

Chapter 3

Categorising Errors

Contents

3.1 Fault Models **96**

3.1.1 Target Specific Factors 96

3.1.2 Simulation vs. Physical Results 97

3.2 Test Strategy **99**

3.2.1 Choice of Target 99

3.2.2 Attack Mechanism 103

3.2.3 Synchronization 103

3.2.4 Bespoke Equipment 104

3.2.5 Specialist Tools 104

3.3 Experiments **106**

3.3.1 Familiarisation 107

3.3.2 Dedicated Tool Development 109

3.3.3 First Results and Tool Revision 114

3.3.4 Revised Controller Board 116

3.3.5 Instruction Behaviour Under Attack 118

3.4 Data and Interpretation **136**

3.5 Summary **136**

Identifying the characteristics of induced errors is the first step in developing effective defences. Here we refine our techniques for categorising errors and identify the main factors that influence them.

Part of the certification requirement for EMV or Common Criteria (CC), is that code reviewers are satisfied with the level of defences implemented in the software. Their conclusions are then tested with a limited amount of penetration testing [187]. This *pen-testing* comprises a set of regular tests, usually automated versions of historically successful attacks, and specific tests inspired by observations made during the code review. The software defences are, as ever, a compromise between many factors, such as code volume on a resource-constrained μC, execution speed, and financial costs relating to development and testing. Likewise, pen-testing is constrained by time, budget, available tools, and the reviewers' knowledge and experience.

For a developer, the question is, which software defences are most cost-effective when considering their engineering costs and the potential degradation of performance? We have seen above in Section 2.2 that, given limitless resources, it is possible to reverse engineer a μC, edit its memory and control each transistor within it. Also, with infinite patience, it is possible to edit the RAM and register contents. Therefore the perfect defence will not be possible. We can, however, identify the nature of inducable errors, their implementation cost in terms of equipment, and the effort required to exploit them. This gives us the cost of an attack. Defence strategy will then be dictated by the value of a defended asset and the attack cost. The tools and techniques used to obtain this information will also be reusable for testing those defences.

Academic papers describing attacks generally concentrate on the outcome of an attack in terms of exploitable results, paying less attention to the cost or practicality of the attack. It is almost a right-of-passage that new or modified attacks are demonstrated by re-implementing the classic Bellcore [36] CRT attack [15, 17, 20, 156, 166, 175]. The notable exception is Balasch et al. [16], who attempted to characterise the nature of faults generated by glitching the clock while a program executed on an Atmel-ATmega μC. They used an unsecured μC packaged as a smartcard. This form factor presented some obstacles to collecting and interpreting the recovered data. They noted that the induced errors were often repeatable and that multi-cycle operations provided data that was easier to interpret. They inferred

this was due to the required interruption of the instruction pipeline and the absence of simultaneous fetch and store operations in this Harvard architecture CPU.

3.1 Fault Models

Device programmers need to make assumptions about both the errors that can be induced in the μC they are programming and the consequences of those errors on program execution. These effects are the micro-scale impact of errors instead of the macro consequences so frequently exploited in the Bellcore attack and its successors. If a program can recognise it is in an anomalous state, it can react and avoid returning faulty results. This framework of what could happen and what the consequences may be is referred to as the *Fault Model*. A fault model may emphasise protecting the μC's program-counter in the belief that glitches may disturb its update and consequently may start executing code from unexpected locations. In this case, a programmer would emphasise defences that track a program's execution path. Alternatively, a model may predict reliable update of the program-counter, but that faulty data may be fetched from memory; in this case, alternative defences would be required. Such a distinction is meaningful for all types of perturbation attacks and can usefully focus the limited defence budget on effective defences.

3.1.1 Target Specific Factors

Hardware defences against glitches are becoming more common; indeed, they are universally implemented on μCs targeting security applications and increasingly common in general-purpose devices. The component manufacturers have effectively neutralised the threat from glitches by implementing hardware for clock and voltage monitoring within the μC. As a result, attackers are now pushed towards the next least complicated attack vector, and device programmers need to consider more complex fault models. The ability to localise fault injection suggests that the bias in the nature of errors witnessed through glitch attacks may no longer be relevant. Instead, the easily exposed and readily attackable structures of a μC will have a disproportionate

influence on the nature of the inducible errors. The exploitable faults will most likely relate to specific components within the DUT, and manufacturing features of the IC's layout will influence the nature of the exploitable errors. What is needed is a way to characterise a μC by identifying the nature of injectable errors, the ease with which they can be triggered, and the repeatability of these effects. With this knowledge, programmers can then marshal their defences to the best effect.

3.1.2 Simulation vs. Physical Results

Simulating the behaviour of a circuit is a well-established method of design verification, widely used in hardware development. Tools exist to simulate all aspects of devices and circuits, from the behaviour of individual atoms within a transistor [162], to the logic level functional behaviour with Verilog and VHDL. The latter encapsulates a methodology akin to software programming with a rich syntax to describe a behaviour along with simulators and debuggers to simplify development and reduce the potential for unforeseen implementation errors. Such tools have been used to model the behaviour of errors within devices, enabling the relative strengths and weaknesses attack methodologies to be compared [101].

Transistor level simulation has been used to model laser stimulation of a RAM cell [151]. While it confirms observed behaviour, the computational requirements make it impractical to scale this modelling to Large Scale Integration (LSI) circuits. Circuit level simulation relies on behavioural models of the sub-components and therefore cannot directly model the physical effects of error inducing stimuli. Instead, failure modes need to be assumed for subcomponents, and their consequences are then modelled through the larger system with manageable computation effort. Such simulation has been used to identify critical logic paths within devices [24] and to predict the behaviour of the cryptographic components such as AES co-processors [170, 74, 133]. Whole circuit simulation becomes computationally impractical when long executions runs are required, and a more abstract simulation is required. Functional simulation of micro-controllers and peripherals are commonplace in many software development environments, and they can also be augmented to simulate errors. Such simulators

have been used to test and categorise the efficacy of various defensive coding structures [173], or to propose combinations of instructions that are immune to localised errors [118]. These approaches rely on a fault model that the simulator can implement while simulating the execution of test programs. The fault model itself is a simplification of observed behaviour and lower level simulations [62].

Furthermore, simulation of errors relies upon assumptions about how, where and when errors can be injected. Radiation hardening experiments have shown that, for a specific type of error — single bit faults, only 15% of errors result in observable computational errors. Such observations rely on a random event component to the fault model, whereas in reality, an attacker will control both the time and location of an error stimulus. When considering the number of possible error types and combinations thereof, exhaustive simulation is impractical.

The accuracy and relevance of a simulation decreases as the abstraction level increases from the localised, transistor-level effects of error injection to the abstract system-level emulation. This accuracy decreases to the point that it can be misleading. Consider Moro et al's *'formally proven ... fault tolerance'* [118]. Here ingenious combinations of instructions are used to ensure that a program can survive skipping of individual instructions, a phenomenon reported by multiple observers on multiple chip architectures [16, 59, 93, 155, 176]. Unfortunately, others have shown that, for their chosen Central Processing Unit (CPU) — the ARM Cortex-M3, instruction skipping is related to the word-sized instruction fetch operation [148, 187] and blocks of four adjacent instructions are skipped or repeated. The oversimplified fault model had led to a considerable amount of effort being wasted.

Where software defences are required, but resources are constrained, attention must be paid to the most readily attackable features and those with the highest value to an attacker if they were to be compromised. Simulation cannot predict this reliably, and what is needed is a mechanism to observe physical attacks, characterise their effects, repeatability, and ease of implementation. Ideally, such a mechanism could characterise a specific device, guide a programmer to appropriate defences and provide a test workbench to evaluate the efficacy of those defences. As a methodology,

it is almost back full circle to the original use of lasers on silicon devices, where physical attacks were used to simulate random radiation-induced errors [81]. Half a century later, we are back where we started as the complexity of modern devices exceeds our ability to simulate their relevant error response behaviour accurately.

3.2 Test Strategy

Our first goal was to categorise the errors that could be invoked on a sample μC.

Multiple independent obstacles need to be overcome when inducing errors, and these obstacles add complications to the implementation of an attack. Each is, to a degree, a defence in its own right, adding cost to the implementation of an attack. However, most of these obstacles are unrelated to the effect we aimed to study, so, where possible, they were eliminated to ensure simplified access to the features we intend to observe. In addition, it was important to simplify the process as much as possible because many experiments were required, and repetition of experiments needed to be automated.

3.2.1 Choice of Target

Multiple factors influenced the choice of target μC for this study.

Security hardened ICs were ruled out for multiple reasons. First of all, the physical protection would add expense and cause a delay in sample preparation; this was considered to be a distraction from the primary aim of characterising the underlying μC. We anticipated the accidental destruction of DUTs while developing our techniques, so a prolonged preparation process needed to be avoided. Secondly, smartcard chips typically lack additional peripherals, such as General Purpose Input Output (GPIO) pins, and we intended to use such features to simplify synchronisation between the DUT and the test harness. Finally, the examples of such devices that we had access to were covered by development and Non-disclosure Agreements (NDA) that precluded any reverse engineering and publication of results.

General-purpose μCs designed for the home automation and IoT market were

of particular interest. As discussed in the introduction (Section 1), they are the Achilles heel in many practical security breaches, and this study's aim is to simplify the adoption of secure programming techniques, particularly in this deployment scenario. These ICs are readily available through multiple distribution channels and are unencumbered with restrictive NDAs.

The choice, therefore, was between the popular device architectures in the μC marketplace. The leading candidates were,

- 8051 derivatives. This design originated in the 1970s and has been popular ever since. Many enhanced derivatives are still available today. Infineon's high-security Sle77 and Sle78 product range is descended from this device family. However, it is an old design and could be considered close to end-of-life.

- ARM Architecture. ARM derivatives are perhaps the single most common μCs on the market. It is widely used in smartphones, laptops and many general-purpose μCs. This architecture has many variants with differing performance enhancements - Single Instruction Multiple Data (SIMD), pipeline lookahead, branch prediction, and out of order execution. Free development tools are available, but the device profiles required for specific devices are often only supported by the commercial toolchains from IAR or Kiel.

- AVR based devices are commonplace in both the smartcard and generic μC markets. Atmel, and latterly Inside-Secure, produce high-security versions of this part, and Microchip Technology Inc. produce a range of generic μCs. The devices have a Harvard architecture, most instructions operate in a single cycle, and the design has a short, one stage pipeline. High-quality development tools are available and free to use.

- Other relatively common architectures are also available. For example, Microchip Technology Inc. has a range of 16 & 32-bit devices. These are widely associated with the hobby electronics market, and they were used in early experiments in the security field [7]. The RISC-V architecture has emerged more

recently as a rival to ARM; it is an open-source, open-standard. It was relatively unknown when this study started and now has many features in common with ARM architecture. Finally, many of the more prominent silicon manufacturers have their own range of μCs; Hitachi/Renesas H8, ST-Microelectronics have their STM8 range, Samsung has their Calm16 & Calm32 devices.

The intention was not to be device-specific, but the investment required to prepare samples and construct test tools meant we needed to concentrate on a representative device. We chose the AVR family of chips to study and, in particular, the ATtiny841 [12]. We anticipated that its relatively simple architecture would provide results that were easier to interpret. While trying to develop techniques to identify patterns in fault behaviour, we considered it wiser to start with the least complicated relevant device. The AVR core has also been studied in relation to fault injection in other academic studies; this provided a context to cross-check our results and conclusions.

Atmel uses the AVR core in a wide range of micro-controllers targeted at the IoT market. It is also widely used in smartcards and similar security devices with a range of variants manufactured by Inside-Secure*.

The ATtiny841 has a few differences from its smartcard targeted siblings.

- The memories are smaller - This would have no material effect on this study.
- There is no metal shielding - The secured variants of the AVR have a top layer of metal that obstructs physical probing and blocks light injection. These defences can, with a significant degree of difficulty, be overcome. NIR lasers can still reach critical components via the reverse side of the chip [178] as silicon is transparent to near-infrared light. The absence of the shield simplified our study without affecting the behaviour we aimed to observe.
- Security monitors - The smartcard variants have peripherals designed to detect glitch attacks. These peripherals typically reset the chip when it is under attack. Techniques exist for bypassing these defences [178] and choosing a target that does not have such defences would not affect the data we aimed to collect.

*Inside-Secure rebranded as Verimatrix in July-2019 and was sold to Rambus in December-2019.

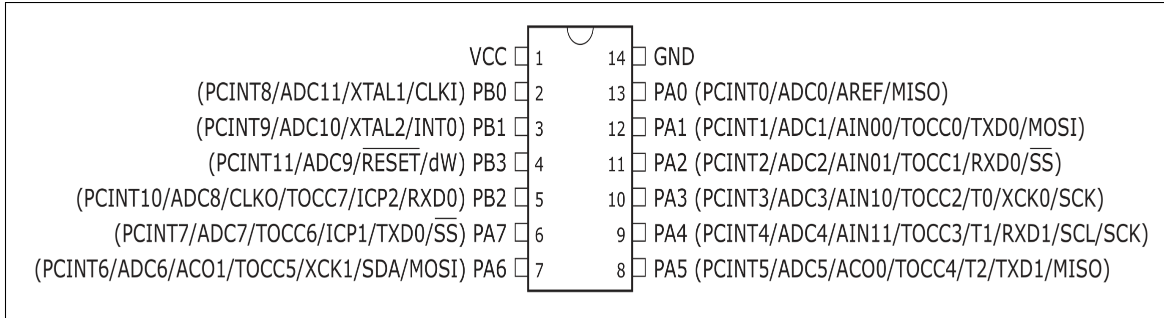


Figure 3-1: ATTiny851 Pin-Out

- Additional peripherals - in particular, GPIO pins enabled us to communicate with a test harness. It would not have been possible to do this with the smart-card variants.

- It was readily available and unencumbered with restrictive licence conditions that would otherwise make our results unpublishable.

Table 3.1: ATTiny841 Features

CPU	
Instructions	120 (mostly single cycle)
Registers	32×8
Clock	16 MHz
Memories (bytes)	
PROM	8192
EEPROM	512
RAM	512
Peripherals	
Serial	$2 \times$ Full-duplex
Timers	2×16 bit
GPIO	12 configurable I/O pins

The majority of the work in this study was performed on this device. It is available in Small Outline Integrated Circuit (SOIC) packaging with the pin configuration shown here in Figure 3-1. The device can be programmed in-circuit, making it relatively easy to reprogram it without disturbing its alignment under the microscope.

3.2.2 Attack Mechanism

Having a target accessible via its top-side enabled us to use visible light (a laser) to induce errors. Light has the advantage that the spot size can be varied, so can the power delivered; this provides greater control of the attack parameters. In particular, visible light has the advantage that standard optics will efficiently aim and focus the beam. We could observe the laser with a CCD/camera to see where the beam was focussed. This greatly simplified the experimental setup.

The recognised downside is that some of the chip's features are obscured by metal tracks, rendering them inaccessible and shielded from attack. We would expect this to be a significant problem for micro-surgical control of individual transistors. However, with our strategy of using laser spot sizes of comparable geometry to the metal tracks, we would expect to be hitting multiple transistors at a time and getting diffraction effects from track edges. As we show in the following chapters, metal track masking was not a problem.

3.2.3 Synchronization

The strategy involved executing and attacking specific instructions, controlling the state of the μC before executing the instruction and then capturing the state of the device after it had completed. While executing, each targeted instruction was subjected to a laser pulse attack. This process was repeated with the laser pulse synchronised with different phases of each machine cycle associated with the instruction under investigation. It was also performed with the laser focussed at different areas on the device's exposed surface and with differing spot sizes and power intensities.

Timing the application of a laser pulse to coincide with specific instructions is a complicated but not insurmountable obstacle; it is discussed here in Section 2.2.2.4.1. We simplified the process by applying an external clock to the DUT and using the same clock signal to drive the timer circuitry responsible for firing the laser. We then used one of the DUT's GPIO pins to indicate to the laser timer when execution of the experimental code started. This signal ensured timing consistency when synchronising

the laser pulse with executing code.

We repeated each test multiple times to provide information relating to the consistency of the results. We could locate and identify the most exploitable error effects and their timings and power requirements by analysing the retrieved data. The intention was to identify the obstacles to harvesting characterisation data and develop reusable techniques for future experiments.

It was clear that this approach would involve many time-consuming and tedious repetitions of relatively simple code fragments. Automation was essential for running tests and processing the resulting data.

3.2.4 Bespoke Equipment

A custom made Printed Circuit Board (PCB) was made to hold the DUT and supply all of the electrical signals needed to run it. It controlled the firing of the laser, programming the DUT, and initiating execution of the test programs. The PCB was connected to a host PC via a serial data link that delivered the results of each experiment. A computer-controlled X-Y stage supported the PCB to enable a controlling PC to position the DUT under the laser. This general configuration evolved as this study progressed, and descriptions of individual configurations are described below in the sections detailing the specific experiments.

3.2.5 Specialist Tools

We were fortunate to have access to a range of tools courtesy of SiVenture Ltd.

The Laser workstation used in our initial experiments was a QuickLaze-50ST from NewWave Research. It is an industrial tool for micro-machining, primarily used for cutting and editing flaws on devices such as LCD displays. The unit consists of a water cooled Nd:YAG[†] Laser mounted on a microscope. The output is selectable as one of three different wavelengths, Infrared 1064 nm, Green 532 nm, and Ultra-violet 266 nm. All of these wavelengths cause electron excitation in silicon and, if suitably

[†]neodymium-doped yttrium aluminum garnet; $Nd : Y_3 A_{l5} O_{12}$

aimed, will induce errors. It has two main power settings, Low and High delivering up to 0.2 mJ and 0.6 mJ per pulse, respectively. Each power band is further controllable between 0% & 100% within each band. The spot size can be varied via a variable aperture. This mechanism obscures parts of the laser beam and, in doing so, reduces the delivered power.

Yttrium Aluminium Garnet (YAG) lasers work by using a high-intensity lamp or gas discharge tube as an *optical pump* to excite the Neodymium atoms in the crystal. When an electron in the excited Neodymium returns to its normal band, it emits a photon with wavelength 1064 nm. The rod-shaped crystal is approximately 15 cm long; it has a mirror at one end and a Q-Switch at the other. When the Q-Switch is closed, it behaves like a mirror, and the energy injected into the rod by the lamp builds up as the light remains trapped in the crystal. Opening the Q-switch releases the light as a very intense short pulse lasting approximately 3 ns.

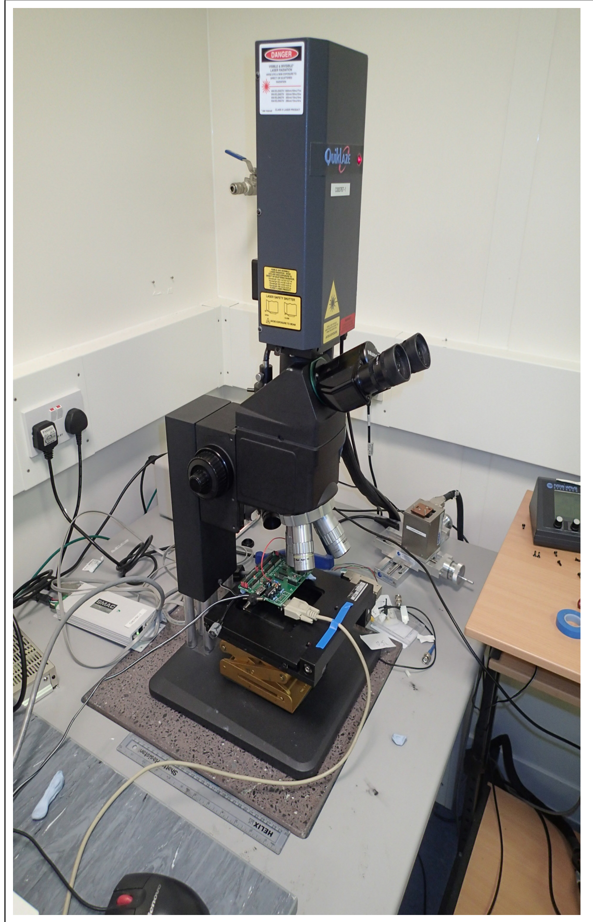


Figure 3-2: Nd:YAG Laser Workstation

The laser is mounted on the camera port of a trinocular microscope. The laser pulse is focussed through the objective lens onto the target. The apparatus is mounted on a stable table and is shown here in Figure 3-2.

The laser driver unit requires two control signals; one to run the lamp that charges the YAG crystal, and another to open the Q-Switch. The latter signal needs to be synchronised with the program running in the DUT to ensure the light pulse strikes at the intended moment.

The DUT was placed on a computer-controlled XY-Stage to enable the target to be positioned with 1 µm accuracy under computer control. Each experiment could then be performed with the laser targeting precisely controlled sites on the DUT's surface without manual intervention.

We developed a PC program to control the stage and repeat experiments as part of this study. It is described in more detail in Appendix A

3.3 Experiments

The first phase of this study was to identify suitable tools, equipment and techniques to capture data from a μC under attack and to identify the effects of the attack. We had already chosen a strategy of running small code fragments and subjecting the μC to laser attack while those fragments were executing; also, we would need to build a device to support the DUT and synchronise the laser attack. We set ourselves the goal of answering the question posed by the contradictory advice received during different certification reviews of an EMV application. Namely, do induced errors have repeatable characteristics that should be considered by application programmers when defining defences? To do this, we performed a series of experiments of increasing complexity.

Step one involved familiarising ourselves with the available tools, confirming that we could successfully induce errors in our DUT, and identifying any complications that may influence the design of the test harness. Step two was to build a test harness that would permit us to synchronise laser pulses with the execution of specific instructions and to gather the resulting state of the DUT. Besides the timing of laser pulses, we wanted control of the site of the attack on the DUT's surface as well as the size and power of the laser spot. We then planned to look at the effects of attacking specific instructions to identify characteristic modes of failure before finally testing realistic sequences of instructions implementing a typical software defence. By the end of this series of experiments, we intend to demonstrate a methodology for recognising the dominant modes of failure.

Our first boards were built with Dual In-Line package (DIL) components to make them easy to patch, fix and modify. As work progressed, increasingly small surface mounted components were required. The only practical way to handle them is to place them directly on a PCB. They are exceedingly tricky to manually solder and too fragile to be attached to freely movable wires. We mounted the DUTs on carriers that could be plugged into the boards, making replacement quick and accessible whenever we destroyed a sample. The other advantage of the PCBs was the inclusion of readily accessible test points to facilitate the attachment of oscilloscope probes and other test devices. This feature was particularly useful when the whole arrangement was placed on a moving stage under the microscope. Reliable attachment of monitoring probes is a fundamental requirement when comparing results between different regions of an IC.

3.3.1 Familiarisation

The first test board was deliberately simple. It also served as a familiarisation exercise, providing rudimentary control of the features required in the intended set of experiments.

The hardware details are described in Appendix B. This board supported in-circuit reprogramming for the DUT, RS232 serial communications with a host PC and the debugger connection (JTAG).

The DUT was programmed to perform an infinite loop, delivering the message "Hello World.", as seen in Figure 3-3. The unused program memory was filled with the value 0x40 (ASCII '@'), and the unused RAM was filled with 0x21 (ASCII '!'). The output from the serial port was displayed on a serial terminal, and the running device was placed directly under the objective lens of the laser workstation.

The laser workstation was configured to deliver a light pulse every half second automatically. This pulse injection was deliberately unsynchronised with the DUT. While the apparatus was running, we were able to vary the position of the DUT, the power of the laser pulse, the spot size of the pulse, all while observing the text output on a monitor screen.

```
1   #define MSG_LEN 12
2   char Message[] = "Hello World.";
3
4   void main(void) {
5
6       Rs232_Setup();
7
8       for (;;) {
9           char *pC = Message;
10          int  i;
11
12          for (i=0; i < MSG_LEN; i++) {
13              Rs232_Send(*pC++);
14          }
15
16          Rs232_Send(13);      // CR
17          Rs232_Send(10);      // LF
18      }
19  }
```

Figure 3-3: Basic Test Program

From this experiment, we made the following observation.

Some locations were sensitive to error injection, while others, sometimes very close by, were seemingly immune to error injection. This result was unsurprising but enabled us to experiment with the power settings and spot sizes in known sensitive areas. We established that, for the 'low-power' setting of the laser cutter, just 15% power with 70% of that obscured by the aperture was enough to cause frequent examples of errors. Higher powers and larger apertures did not noticeably increase the error rate. Therefore there was no need to experiment with the laser in high power mode and risk damaging the DUT.

The errors we observed fell into four common categories.

1. Occasionally, we observed an unrecoverable crash necessitating manual intervention and a power cycle to restart the test message delivery.

2. On some occasions, the message delivery froze mid-sentence. Such events usually lasted multiple seconds before restarting normal execution. The pause was long enough for the stalled program to be subjected to multiple additional laser strikes. This stuttering effect suggested the crashed program was being re-crashed, ultimately resetting the device.

3. A common occurrence was a message of expected length but displaying the '@'

character, suggesting a pointer corruption in the message reading loop.

4. A similar error occurred where the length of the delivered text increased, resulting in the delivery of additional '!' characters. This additional output can be explained as a corruption of the loop counter or the loop test condition, causing a buffer overrun. The additional '!' characters indicate that the data was retrieved from RAM.

This exercise demonstrated multiple modes of failure, and that timing was a significant factor in determining failure mode, as evidenced by the fact that some pulses invoked no observable effect. In contrast, others caused miss-execution at the same target location. This timing dependency was what we expected, given the volume of pre-existing reports of such behaviour.

More importantly, we identified the appropriate range of power settings for the tools and demonstrated that we could control our DUT, attack it, and reprogram it in situ. Furthermore, we noted that observation 1 (above) did not occur if the JTAG debug connection was powered. We could not replicate the error with the debugger attached, and basic attempts to trace or debug the behaviour failed. Presumably, the JTAG cable was delivering power from the debugger to the DUT and this affected sensitivity of some parts of the DUT to laser pulses. Knowing that the debug connection could influence the error response, we ensured that future experiments were performed in the non-debuggable state.

3.3.2 Dedicated Tool Development

We now had a set of requirements for our first support PCB. The DUT needed to be physically reset between experiments rather than relying on its soft rest capabilities. We needed a mechanism to initiate a test and recover its result that did not require manual intervention or use the DUT's debug interface. It was also vital that the laser could be precisely synchronised with the executing test program. Large numbers of tests were to be performed because many code samples needed to be run, and they needed to be repeated with the DUT repositioned under the objective lens.

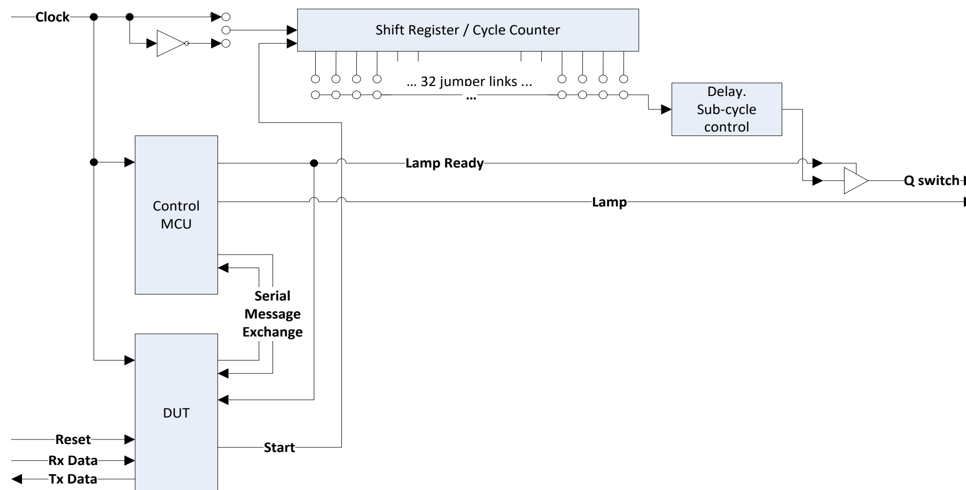


Figure 3-4: DUT Support, Board 1

3.3.2.1 DUT life support PCB

The first solution we adopted is shown in Figure 3-4. The DUT communicated directly with the host PC via a serial link. A second μC was responsible for generating the laser priming *Lamp* signal and for indicating to the DUT when the laser was primed and ready to fire. The DUT then generated a pulse on the *Start* signal line that rippled through a 32 stage shift register. This pulse triggered the laser's Q-Switch, and the event timing was controlled by placing a jumper on any of the 32 shifter outputs. Meanwhile, the DUT would be executing the test code. By carefully calculating the instruction cycle counts and positioning the jumper on the corresponding pins, we could ensure the laser pulse coincided with the execution of a specific instruction within the test sample. This delegation of duty between μCs ensured that if the DUT crashed as a result of the laser injection, the laser itself could still be switched off by the support μC. Both μCs and the shifter were synchronised by a shared clock source. The clock to the shifter was selectable as being in phase or anti-phase with the mainboard clock. This feature provided the opportunity to synchronise the laser pulse with either the falling or rising edge of the clock, i.e. two distinct instances per instruction cycle. The control sequencing for this board is shown in Figure 3-5 and more details about this board are available in Appendix B.2.

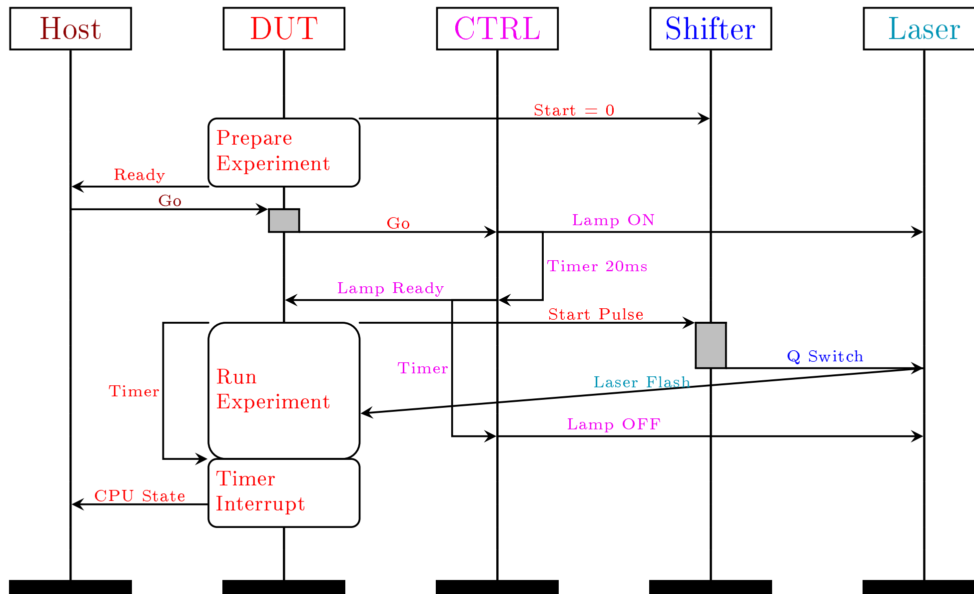


Figure 3-5: Board 1 Control Logic

The test programs in the DUT all followed a common pattern.

1. First, all of the memory and working registers were initialised to a known state. This ensured we had a known starting state and could identify any changes at the end of the experiment.

2. An internal timer was used to trigger an interrupt two cycles after the expected time of the laser pulse. By using an interrupt, we could take control of the CPU even if it had crashed.

3. The interrupt service routine would then report the current state of the μC to the host PC. This information comprised of the CPU's registers, the program counter, stack pointer and status flags. The state of the RAM contents was delivered as a series of checksums — one for each 32-byte block. The non-volatile memory was treated the same way as a series of checksums of 256-byte blocks. The use of checksums enabled us to detect corruption in the memory without the time delay of delivering the entire contents and the storage issue of recording the contents for such a large number of experiments.

4. The DUT was then halted and waited for a reset from the host PC to start the subsequent execution of the same test.

3.3.2.2 Experiment Invocation

Each test was performed by executing a small PC program, `Zap.Exe`, on the Host PC. Command-line parameters indicated the current coordinates of the experiment and the name of a log file where the results were placed. This program reset the DUT, initiated the experiment and collected the response from the board. A single line of Comma-Separated Values (CSV) data was appended to the log each time the program ran. Using CSV, a universal data format, enabled the results to be viewed in a wide variety of applications, such as spreadsheets, databases or even simple text editors. `Zap.Exe` was a small easy to modify program that could be easily customised whenever the programmed behaviour of the DUT was changed.

3.3.2.3 Test Campaign

A larger, more sophisticated program was developed to coordinate multiple invocations of `Zap.Exe` while controlling the microscope stage position. This application is described in more detail in Appendix A. The DUT could be observed via a video camera placed in one of the microscope's eye-pieces. Features on the DUT, observable through the limited field of view of the microscope, were paired with features on a reference image of the IC. The movable stage was then calibrated by cross-referencing landmarks from the current XY-position (crosshairs in Figure 3-6) with the same features in the reference image's coordinate system. The user could then identify a target area of interest on the reference image (green rectangle in Figure 3-6) and direct the stage to align the same feature on the IC. The coordinate system of the reference image was used when recording position within the experimental results for the DUT. Using the image as the base coordinate system simplified the interpretation of the results as laser spot locations could then be directly mapped to this image.

A grid defined for the reference image identifies the coordinates of each experiment. This grid spacing is configurable, and was set to the same size as the laser spot.

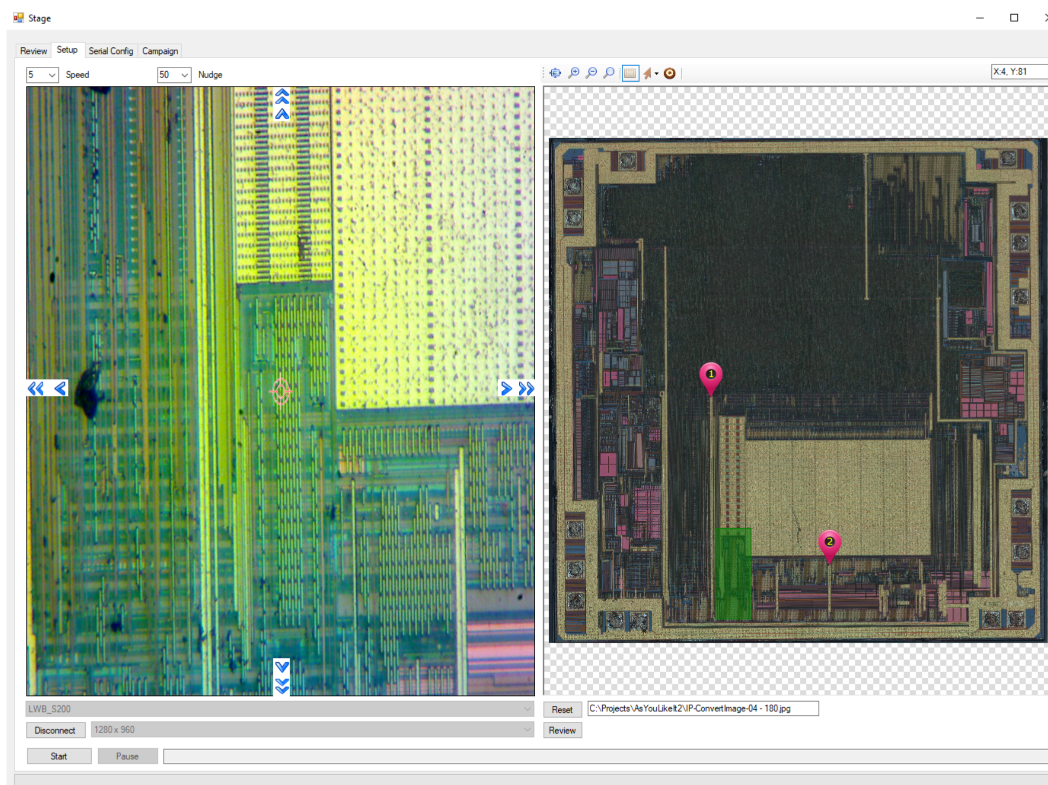


Figure 3-6: Test Campaign Configuration

Each campaign involved the configuration of the following parameters.

1. The DUT was programmed with the appropriate sample code.
2. The pulse timing was configured via the jumpers on the control board. Alternatively, the test code could be synchronised with the board settings by inserting NOP operations in the preamble to a tested instruction.
3. The laser power and spot size were selected.
4. The area to process was identified, and the number of repetitions to perform was indicated.
5. Finally, the microscope lamp was switched off, leaving the laser as the only (intense) light source directed at the DUT.

The test manager then executed the whole campaign. It sequentially positioned the DUT to direct the laser onto each grid position within the identified campaign region before executing `Zap.Exe` at each of these positions.

3.3.2.4 Results Analysis

The analysis of collected data was highly dependent on the nature of the test performed. It invariably involved writing small utility programs to analyse, aggregate and present the results.

3.3.3 First Results and Tool Revision

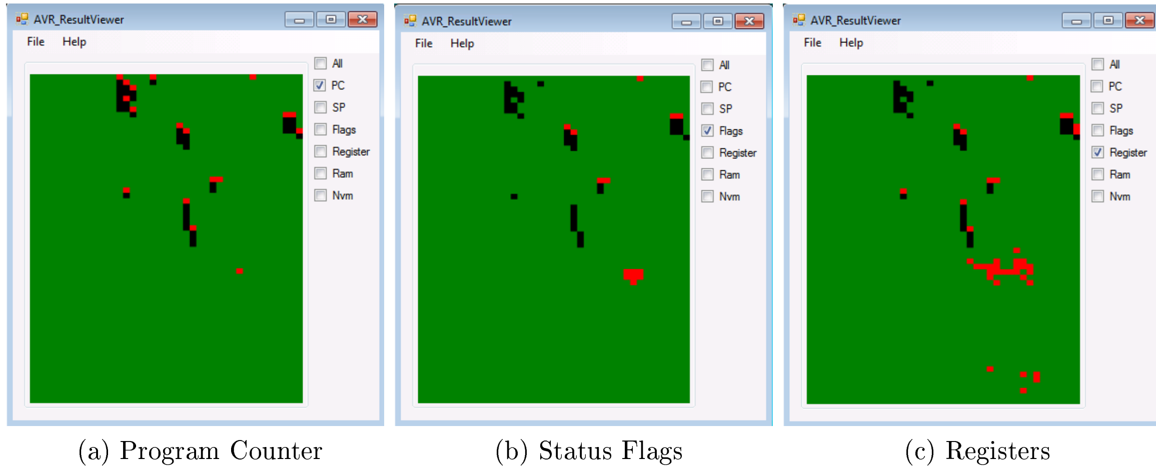
To test the equipment, we devised a basic test that repeatedly incremented a register; see Figure 3-7.

```
void Task01(void) {
    PORTB &= ~TRIG;                 // Start = 0
    UartTx0(0x55);                  // Msg Ctrl to start
    while ((PINB & READY) == 0);    // wait for LAMP READY

    PORTB |= TRIG;                  //      shifter = 1
    PORTB &= ~TRIG;                 //   0  shifter = 0
    __asm (
        "eor␣r0,␣r0\n"              //   1  R0 <= 0
        "inc␣r0\n"                  //   2
        "inc␣r0\n"                  //   3
        "inc␣r0\n"                  //   4
        "inc␣r0\n"                  //   5

            ...

        "inc␣r0\n"                  // 30
        "inc␣r0\n"                  // 31
        "inc␣r0\n"                  // 32
        "inc␣r0\n"                  // 33
        "inc␣r0\n"                  // 34
        "inc␣r0\n"                  // 35
        "inc␣r0\n"                  // 35
    );
    return;
}
```

Figure 3-7: Test Program 1

We expected register R0 to display a faulty result if an execution error could be induced. The whole surface of the DUT was scanned as a 30×30 grid. The board was configured to fire the laser at cycle 4, and the state reporting interrupt was expected at cycle 8.

A result viewing program was written to demonstrate the locations of any errors that occurred graphically. Sample images can be seen in Figure 3-8. Green indicates no error, i.e. that the recovered data were identical to a reference sample taken when the laser did not fire. Red indicates locations where the results differ, and black



(a) Program Counter (b) Status Flags (c) Registers

Figure 3-8: Error Distribution Plots

indicates the DUT failed to deliver any results, i.e. it crashed. Image 3-8a shows the locations where program counter errors occurred due to a laser pulse. 3-8b shows where the flags register ended up with unexpected values and 3-8c where register corruption was observed.

The process demonstrated more or less what was expected; a background of scattered locations that caused crashes and clusters of locations containing related error conditions. When the Program Counter was wrong, indicating unexpected code was being executed, there was also a high probability that a register was corrupt, as most instructions modify registers. When a register is modified, there is also a good chance that one or more flags will be altered.

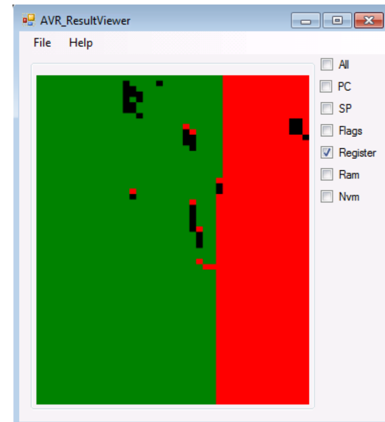


Figure 3-9: Lock-up Error

We intended to perform a large number of specific experiments using a finer grid spacing. Unfortunately, we discovered a relatively common error side-effect that prevented the automation of many tests. Occasionally we would see a result, demonstrated in Figure 3-9. The μC was running and returning data; however, one register was permanently corrupted. The error condition did not clear when the Host-PC toggled the reset even though the reset had the expected effect of preparing the DUT

for the next run of the experiment. Power consumption of the device went up by approximately 10 mA and remained high until a full power cycle reset was performed. This hard reset also had the effect of clearing the error condition.

In light of this, we modified the test board to do a complete power cycle whenever the Host PC toggled the reset line[‡]. It solved the DUT reset problem, but the frequent power cycles had an undesirable effect on the laser's controller with unexpected glitches on the lamp and Q-Switch inputs. It occasionally generated laser pulses while the DUT was performing its bootstrap and, on other occasions, failed to fire at all.

While the device was good enough for short supervised tests, it could not be relied upon for prolonged unsupervised campaigns; we, therefore, took the opportunity to revise the whole control board.

3.3.4 Revised Controller Board

The primary reason for this board upgrade was to enable the temporary electrical isolation of the DUT. This isolation enabled it to be reset via a power cycle without affecting any support components.

The shift register was also replaced with a programmable counter. The control μC programmed this and it ran from 40 MHz clock. This arrangement gave quarter cycle resolution for laser pulse injection into the μC's instruction cycles.

The end of the countdown then generated an interrupt signal to the DUT which activated one quarter-cycle before the laser triggered. The DUT's interrupt latency of three CPU cycles resulted in a short delay before the instruction pipeline froze and the service routine vector was fetched. The influence of this delay can be seen in Figure 3-18's PCINT and IRQ signal lines. This feature removed the need to configure a timer interrupt within the DUT in advance of an experiment. In turn, this reduced the opportunity for human error and simplified test preparation.

More details of this board can be found in Appendix B, and the control logic is shown in Figure 3-11.

[‡]This modification can be seen in Figure B-1; it is the small appendage on the left.

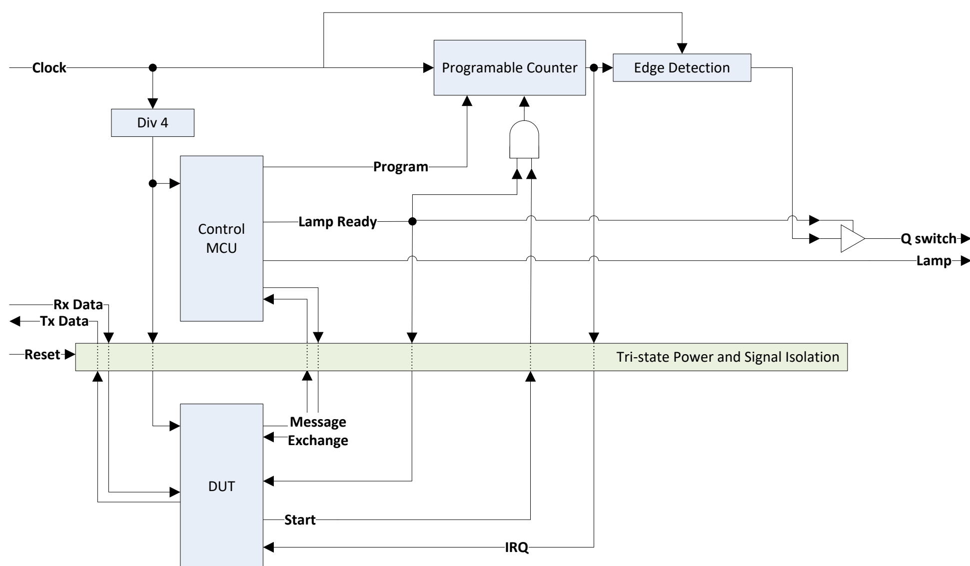


Figure 3-10: DUT support, Board 2

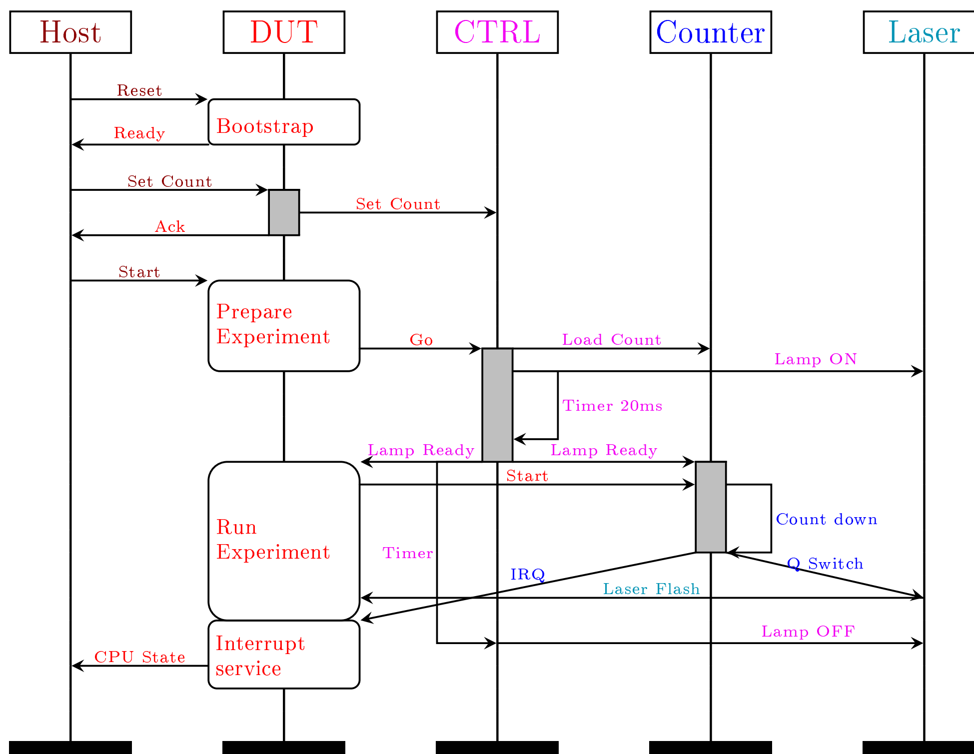


Figure 3-11: Board 2 Control Logic

3.3.5 Instruction Behaviour Under Attack

3.3.5.1 Targeting Individual Instructions

The first step was to subject instructions of different classes to an extensive campaign of laser strikes. Each instruction test involved three stages. The test program set up relevant start conditions and sent the *Start* signal to the delay counter. It then executed a sequence of NOP instruction followed by the instruction under investigation and then executed a further sequence of NOPS. This sequence of NOPS, or whatever crash state had been induced, was then interrupted by the state reporting mechanism.

```
 1  .global Expt
 2  Expt:
 3          rcall PrimeTheTest
 4  E1_1:   sbis  0x16,2        // while (!LAMP_RDY);
 5          rjmp  E1_1          //
 6
 7      <SETUP here>
 8
 9          sbi   0x18,1        // PORTB  |= START
10                              //   Cycle       Laser time
11          nop                 //   0           0.2 - 1.1
12          nop                 //   1           1.2 - 2.1
13          nop                 //   2           2.2 - 3.1
14      <INSTRUCTION here>      //   3  -zap-    3.2 - 4.1
15          nop                 //   4      |    4.2 - 5.1
16          nop                 //   5      |    5.2 -
17          nop                 //   6      |
18          nop                 //   7 <---- ISR after here
19          nop                 //   8
20          nop                 //   9
21          nop                 //  10
22          nop                 //  11
23          nop                 //  12
24          nop                 //  13
25          ret
```

Figure 3-12: Instruction Test

The target instruction was surrounded by NOPS in the expectation that the only changes in the μC's state would relate to the instruction being attacked or the μC's instruction scheduling logic. See Figure 3-12. The 'setup' code initialized the required state for the chosen experiment, for example, setting the carry flag. This action corresponds to the initialisation state listed in the result tables; Tables 3.2, 3.3, 3.4, 3.5 & 3.6.

Each test code sample was executed with laser pulses timed to precede, cover, and follow the tested instruction. As shown in Figure 3-13. For single cycle instructions,

this required 6 runs with the pulse timed accordingly. Two-cycle instructions required 10 runs per experiment to ensure all execution phases were potentially disturbed. Each experiment was also executed 4 times to test for repeatability. All of this was repeated for all points in a 100×100 grid over the μC's surface. Testing a single cycle instruction therefore took $6 \times 4 \times 10,000 (240,000)$ execution runs and generated a $180 Mbyte$ result file.

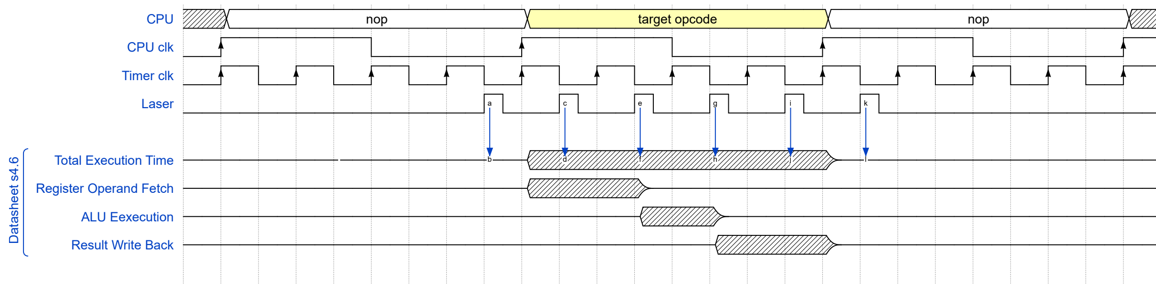


Figure 3-13: Laser Firing

The first three columns of the tables that follow here show the start state, the tested instruction, and the expected effect of that instruction, assuming execution was error-free. The remaining columns show the total number of samples, the number of errors detected and the number of those errors that were unique. Uniqueness was measured for each repetition of four tests performed at the same location and with the same timing for the laser pulse. An error was considered unique if the μC's recorded state differed from the other three samples. Four tests terminating with identical errors would register as four *Errors* but zero *Unique* results; four tests where all results differ would show four *Unique* results.

$$\therefore \quad Errors \geq Unique \geq 0.$$

Smaller values of *Unique*, within these bounds, indicates more repeatable errors. Conversely, larger values of *Unique* indicates unpredictability in the DUT's error response.

3.3.5.1.1 No Operation Tests.

Testing NOPs provided the opportunity to observe changes to the μC relating purely to the instruction fetch with none of the additional complications relating to data transfers or arithmetic calculations. The NOP tests provided a useful reference to compare other tests against; see Table 3.2.

Table 3.2: Laser induced errors on NOP

Initialisation	Instruction	Effect	Samples	Errors	Unique
CLC[a], CLZ[b]	NOP[e]	No change	240000	2045	787
STC[c], STZ[d]	NOP	—"—	240000	2130	802
			480000	4175	1589

[a] Clear Carry. [b] Clear Zero. [c] Set Carry. [d] Set Zero. [e] No-Operation. No register or flag updates expected.

3.3.5.1.2 Single Register Update Tests.

The instructions tested here were characterised as Read-Modify-Write. They access only one register and update a subset of the status flags; see Table 3.3.

Table 3.3: Laser induced errors on increment and decrement

Initialisation	Instruction	Effect[a]	Samples	Errors	Unique
R0←00H	INC R0[b]		240000	1721	730
R0←01H	—"—		240000	1725	750
R0←7FH	—"—	N	240000	1901	805
R0←FFH	—"—	Z	240000	1837	793
R0←00H	DEC R0[c]	N	240000	1857	789
R0←01H	—"—	Z	240000	1816	764
R0←02H	—"—		240000	1704	744
R0←80H	—"—		240000	1759	800
			1920000	14320	6175

[a] Updates **Z**ero, **N**egative & O**V**erflow flags. [b] Increment Register. [c] Decrement Register.

3.3.5.1.3 Arithmetic Tests.

These arithmetic instructions involve binary operations (e.g. addition) taking input from two sources and overwriting the first with an updated value; see Table 3.4.

Table 3.4: Laser induced errors on multiple register/register and memory access

Initialisation	Instruction	Effect[a]	Samples	Errors	Unique
R16←60H, R17←10H	ADD R16, R17[b]		240000	1869	834
R16←60H, R17←30H	—"—	V N	240000	1899	826
R16←60H, R17←A0H	—"—	C Z	240000	1916	815
R16←60H, R17←A5H	—"—	C	240000	1886	817
R16←20H, R17←15H	SUB R16, R17[c]	H	240000	1922	824
R16←20H, R17←20H	—"—	Z	240000	1949	823
R16←20H, R17←25H	—"—	H S N C	240000	1955	806
R16←20H, R17←B0H	—"—	C	240000	1920	785
R16←7, R17←6	CP R16, R17[d]		240000	2197	942
R16←7, R17←7	—"—	Z	240000	2272	1005
R16←7, R17←8	—"—	H S N C	240000	2320	1019
R16←7	CPI R16, 6[e]		240000	2203	1003
R16←7	CPI R16, 7	Z	240000	2306	1058
R16←7	CPI R16, 8	H S N C	240000	2309	1005
			3360000	28923	12562

[a] All tests update the **H**alf-Carry, **S**ign, O**V**erflow, **N**egative, **Z**ero, & **C**arry flags. [b] Add – R16 ← R16 + R17. [c] Subtract – R16 ← R16 − R17. [d] Compare – R16 − R17. [e] Compare Immediate – R16 − const.

3.3.5.1.4 Memory Access Tests.

The memory access tests observe instructions that perform data transfers, i.e. transfers from registers to memory and memory to registers; see Table 3.5.

Table 3.5: Laser induced errors on load store/memory to register and register to memory exchange

Initialisation[a]	Instruction	Effect[b]	Samples	Errors	Unique
@100←00H	LD R2, X[c]	load zero	400000	2124	900
@100←5AH	—"—	load positive	400000	2172	937
@100←A5H	—"—	load negative	400000	2166	892
R16←00H	ST X, R16[d]	store zero	400000	2135	883
R16←5AH	—"—	store positive	400000	2133	883
R16←A5H	—"—	store negative	400000	2143	873
			2400000	12873	5368

[a] Index register X←100h. [b] No flags expected to change. [c] Load indirect. Register ← memory at address X. [d] Store indirect. Memory at address X ← R16.

3.3.5.1.5 Conditional Branch Tests.

The conditional branch tests were of particular interest to the study as they are critical to defensive coding strategies. When executing conditional branch instructions, the program flow will continue to the next instruction or jump to a new address, depending on the state of a status word flag. The expectation was that this behaviour could be disrupted by fault insertion, resulting in a change in the control flow of the program; see Table 3.6.

Table 3.6: Laser induced errors on conditional branch instructions

Initialisation	**Instruction**	**Effect**	**Samples**	**Errors**	**Unique**
CLC[a]	BRCC[e]	branch taken	400000	3407	1270
STC[b]	—"—	branch skipped	400000	3359	1248
CLC	BRCS[f]	branch skipped	400000	3414	1320
STC	—"—	branch taken	400000	3455	1283
CLZ[c]	BRNE[g]	branch taken	400000	2200	836
STZ[d]	—"—	branch skipped	400000	2384	911
CLZ	BREQ[h]	branch skipped	400000	2256	901
STZ	—"—	branch taken	400000	2370	914
			3200000	22845	8683

[a] Clear Carry flag. [b] Set Carry flag. [c] Clear Zero flag. [d] Set Zero flag. [e] Branch if Carry flag is clear. [f] Branch if Carry flag is set. [g] Branch if Zero flag is clear. [h] Branch if Zero flag is set.

3.3.5.1.6 Aggregated results..

Table 3.7 shows a summary of the results of these experiments. The nature of the errors is discussed below.

Table 3.7: Laser induced errors

Test Set	Samples	Errors	Unique	Repeat[a]
No Operation	480000	4175	1589	61.9%
Single Register	1920000	14320	6175	56.9%
Arithmetic	3360000	28923	12562	56.6%
Memory	2400000	12873	5368	58.3%
Branching	3200000	22845	8683	62.0%
	11360000	83136	34377	58.6%

[a] $Repeat = (Errors - Unique)/Errors$.

3.3.5.1.7 Error categorization.

For each of the proceeding tests, we generated a map of the sensitive regions of the IC. One such map is shown here in Figure 3-14. All error location images showed a similar distribution of locations where errors had been induced. The utility developed to create these images can generate similar images identifying specific types of error. Many such images were generated and could be visually compared to identify areas where interesting results could be found rapidly.

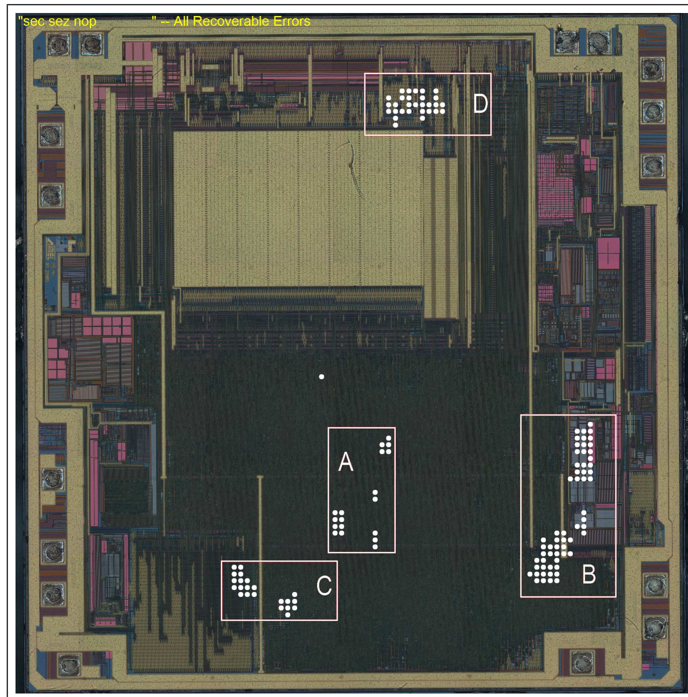


Figure 3-14: Error Locations

The example here is typical of all of the tests; they all show clusters of sensitive sites, and each of these sites exhibits errors with common characteristics. Presumably, circuitry implementing related functionality is physically located in the same region of the IC.

We identified six classes of error.

1. **Crash and no response.** Crashes were comparatively rare; so long as the CPU was executing, then the interrupt-driven state collection mechanism could recover the μC's state. We noted that when they did occur they were associated with the zones marked **A** & **B** in Figure 3-14.

2. **Widespread non-fatal corruption.** Multiple registers became simultaneously corrupted, most frequently in zone **B**. More registers were corrupted than could be achieved via program execution, and we assume the register contents were simultaneously altered. The general behaviour was often repeatable, but the erroneous values assigned to the registers had no apparent pattern.

3. **Program counter corruption.** Program counter corruption was common. Mainly associated with zone **D**.

4. **Working Register Corruption.** The working hypothesis was that if we could corrupt an instruction that manipulated a specific register, that register would end up with corrupted contents. Surprisingly, we did not see any examples of this behaviour. The reasons for this are discussed below.

5. **Status Register.** Here the flags register was corrupted without any other errors induced in the μC. These errors occurred exclusively in Zone **D**. This flag corruption was a surprising result, given the stability of the general-purpose registers and is discussed further below.

6. **Memory corruption.** In some instances, the memory became corrupted, but with no corruption to the μC state. This behaviour was associated with zones **C** and **D**, and appears similar to the effect studied in detail by [165] where

RAM was modified directly. Such corruption is surprising as the IC's tracks in these zones do not have the regular pattern generally associated with memory structures.

These results were not at all what we had expected. Status flags were being corrupted when they were not expected to be updated by the instruction being investigated. Registers that were being manipulated by the instruction under investigation still ended up with the correct results in them. Program counter corruption was frequently observed, and surprisingly it was no more or less common when investigating branch instructions than it was for arithmetic or Nop instructions.

These results appeared to contradict the assumptions underpinning much historical defensive code. Namely, the ability to corrupt arithmetic or data transfer and branching could be exploited to bypass software security features.

What we were observing was a high degree of repeatability. Under the same conditions, the resulting corruption generated the same response. Except for the widespread non-fatal corruption seen in zone **B**, the erroneous data and resulting execution addresses were reproducible.

Another noteworthy observation related to the number of errors recorded for each of the different locations on the IC's surface. Figure 3-15 illustrates this. The heights of the peaks represent the aggregated total number of errors recorded for those locations. Notably zones **B** & **C**, that were associated with corrupted memory, generated most errors. Zone **D** by contrast looked relatively insensitive. However, errors here were strongly correlated with the timing of the laser pulse. This observation supported the idea that for zone **D** we were observing a perturbation in some aspect of the μC's

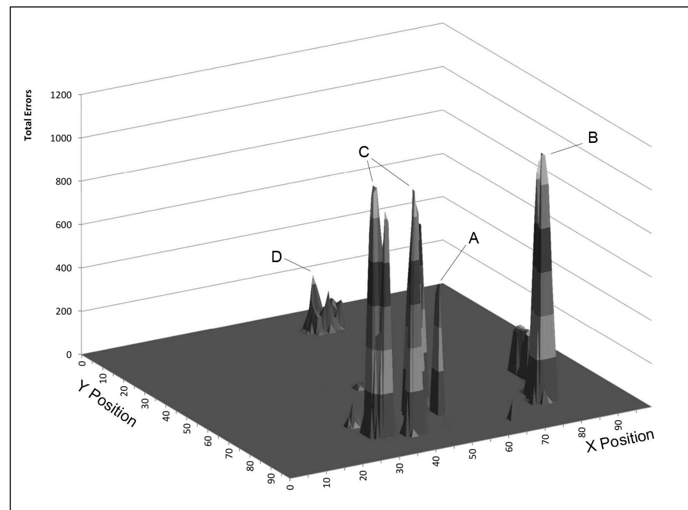


Figure 3-15: Error Sensitivity Map

dynamic behaviour because it was related to the specific timing of the laser pulse. On the other hand, for zones **B** & **C**, we were probably looking at memory editing, where the timing of the pulse is irrelevant.

3.3.5.2 Targeting Running Code

The second set of tests was performed on a realistic code sample, typical of a defensive code. On entry register `r25` contains a value of either `STRUE`, `SFALSE`, or a corrupt value if something had caused an error previously. The sample has four termination states, one of each entry condition and one for errors encountered during its own operation.

```
1                                 // Timing
2             nop                 // 1.0-1.1
3             nop                 // 1.2-2.1
4             cpi  r25,SFALSE     // 2.2-3.1
5             breq L_False        // 3.2-4.1/5.1
6             cpi  r25,STRUE      // 4.2-5.1
7             breq L_True         // 5.2-6.1/7.1
8             rjmp L_Trap         // 6.2-8.1
9             rjmp L_Trap         // 8.2-10.1
10
11  L_Weird:  mov  r0,r25         // Note: mov is used here
12            mov  r1,r25         // because it does not
13            mov  r2,r25         // update any flags.
14            mov  r3,r25         //
15            rjmp L_Weird        //
16
17  L_True:   mov  r10,r25        // 7.2-8.1
18            mov  r11,r25
19            mov  r12,r25
20            mov  r13,r25
21            rjmp L_True
22
23  L_False:  mov  r16,r25        // 5.2-6.1
24            mov  r17,r25
25            mov  r18,r25
26            mov  r19,r25
27            rjmp L_False
28
29  L_Trap:   mov  r4,r25         // 8.2-9.1
30            mov  r5,r25
31            mov  r6,r25
32            mov  r7,r25
33            rjmp L_Trap
```

Figure 3-16: Branch Test Code

Under normal circumstances, the program shown in Figure 4-12 will reach the appropriate end state where it will be trapped in an endless loop waiting to be interrupted by the state reporting mechanism. For these experiments, the interrupt was delayed by 8 cycles to ensure the program had time to execute to one of the end

functions[§]. Good values of `STRUE` & `SFALSE` should reach the appropriate handler, while corrupt input should terminate in the `L_Trap` function. `L_Weird` would only be reached if the data was corrupt and both of the attempts to jump to the trap failed.

We performed three tests, each with `r25` pre-initialised to one of the three possible start conditions. Each of these was performed at all points within the previously identified zone **D**, a grid of 10×7 sample points. A single test involved twenty-four separate execution runs with the laser pulse triggering at each of 24 consecutive quarter cycles, the first of which was synchronised with line 4 of the source sample. Each test was repeated 10 times at each location.

Table 3.8: Branch Test Results

	Start state, R25=					
	SFALSE		STRUE		0	
Final State	n	%	n	%	n	%
Weird	7875	0.5	16625	1.0	48125	2.9
True	0	0.0	1627500	96.9	0	0.0
False	1647625	98.1	0	0.0	0	0.0
Trap	24500	1.5	35875	2.1	1631875	97.1
Total	1680000		1680000		1680000	

The results of this test are shown here in Table 3.8. The interesting result is that we had no examples of branches erroneously taken. All of the outcomes could be explained by skipping instructions and failing to branch.

Closer observation of the results showed that some MOV instructions in the destination routines had also been skipped. Timing of the laser pulse associated with skipped branches and MOVs was such that it had occurred during the execution of the proceeding instruction. This timing strongly suggested that instructions were not miss-executing but that the pre-fetch of the next instruction was being corrupted.

Some arrivals at the `L_Weird` termination appeared to be the result of more than one fault. For example for the `STRUE` start state to arrive at the `L_Weird` end state requires one of the instructions at line 6 or 7 to fail along with both of the instruction

[§]This trigger delay can be seen in Figure B-3, it is the additional, retro-fitted circuit board.

at lines 8 and 9. However, since the jump instructions are double-word sized, it is conceivable that the instruction pointer could become misaligned with the code, and we were witnessing the normal execution of a displaced byte stream.

Analysis of the results from a single location, where faults had been observed, showed that the errors were associated with a laser pulse synchronised with the third quarter machine cycle. Table 3.9 shows the results from one such location where the only laser pulses used were synchronised with quarter three.

Table 3.9: Branch Test Results, fixed location, Third Q-cycle

	R25					
	SFALSE		STRUE		0	
Final State	n	%	n	%	n	%
Weird	9	3.8	19	7.9	55	22.9
True	0	0.0	180	75.0	0	0.0
False	203	84.6	0	0.0	0	0.0
Trap	28	11.7	41	17.1	185	77.1
Total	240		240		240	

These results strongly supported the conclusion that we witnessed instruction skipping rather than in-transit corruption of data. Similar results had been observed by Riviere [148] on an ARM μC where the cache failed to update, resulting in the re-execution of the previous cache contents.

Consequently, our initial assumptions that drove the extensive scanning of the chip surface were flawed. All instructions were examined while they prefetched a Nop. This result suggests many of the observed errors related to the misbehaviour of the subsequent miss-fetched Nop; i.e. not related to the specific instruction under test.

3.3.5.2.1 Testing the Skip Hypothesis

The timings of the laser pulse and the observed effects pointed to a miss-fetched instruction rather than the abnormal execution of an uncorrupted instruction. To confirm this hypothesis, we devised a simple test. We executed a series of Inc in-

structions and subjected them to the same exhaustive attack previously used. A skipped instruction should be observable as a register with unmodified contents. A re-executed instruction would be expected to be shown by a multiply incremented register. The test code is shown in Figure 3-17.

The signal that triggers the laser was used to initiate the data recovery interrupt. No additional post-laser-strike delay was required. As the timer delay increased registers `r2`, then `r3`, then `r4` etc. failed to increment. We observed the missing updates, as expected, but saw no examples of the additional updates that would indicate re-execution of the previously fetched instruction.

```
1             Boot_vec         // Power-on/reset
2             Irq_vec          // vector to ISR
3
4
5   Irq_vec:  rjmp  ISR        // jump to ISR
6
7
8   Expt:     rcall  Setup     // Zero all registers
9
10  E1_1:     sbis   0x16,2    // while (!LAMP_RDY);
11            rjmp   E1_1      //
12            sbi    0x18,1    // Set START signal
13
14                             //   Cycle,   Laser time
15            inc    r0        //       0,   0.2 -  1.1
16            inc    r1        //       1,   1.2 -  2.1
17            inc    r2        //       2,   2.2 -  3.1
18            inc    r3        //       3,   3.2 -  4.1
19            inc    r4        //       4,   4.2 -  5.1
20            inc    r5        //       5,   5.2 -  6.1
21            inc    r6        //       6,   6.2 -  7.1
22            inc    r7        //       7,   7.2 -  8.1
23            inc    r8        //       8,   8.2 -  9.1
24            inc    r9        //       9,   9.2 - 10.1
25            inc    r10       //      10,  10.2 - 11.1
26            inc    r11       //      11,  11.2 - 12.1
27            inc    r12       //      12,  12.2 - 13.1
28            inc    r13       //      13,  13.2 - 14.1
29            ret
30
31
32  ISR:      cbi    0x18,1    // Clear START signal
33            <Report CPU state>
34            ...
```

Figure 3-17: Pre-Fetch Test

This experiment also provided a convenient sanity check for other assumptions made while analysing results. In particular, the timing and sequence of events involved in the data capture.

The delay counter started its countdown after the *Start* signal was asserted. The *Start* signal was then cleared as the first action of the Interrupt Service Routine

(ISR). With the delay counter set to 12 quarter cycles timer would have expired 3 CPU cycles after *Start* was asserted. The *End_ count* signal initiated both the CPU's interrupt triggering logic and tripped the laser's Q-Switch. After the laser's internal delay of 1 µs [124], the Q-Switch would open, enabling the light pulse to emerge. In our test program, this aligned with the execution of INC R4, line 19.

Experimental results showed registers R0 - R4, and R6 were set to 1, whilst registers R5 and R7,... remained unaltered. R5 had clearly been skipped, and the instructions, INC R7 onwards, had not been executed as a result of the non-returning ISR. The *Start* signal was cleared after 14 CPU cycles. These observations match the timing behaviour for interrupts described in the device's datasheet [12] and are shown here in Figure 3-18. It also demonstrated that the CPU maintained its regular timing. I.e. the miss-fetch of INC R5 was not a result of an elongated or skipped CPU cycle, as would be have been expected with an externally applied clock glitch attack.

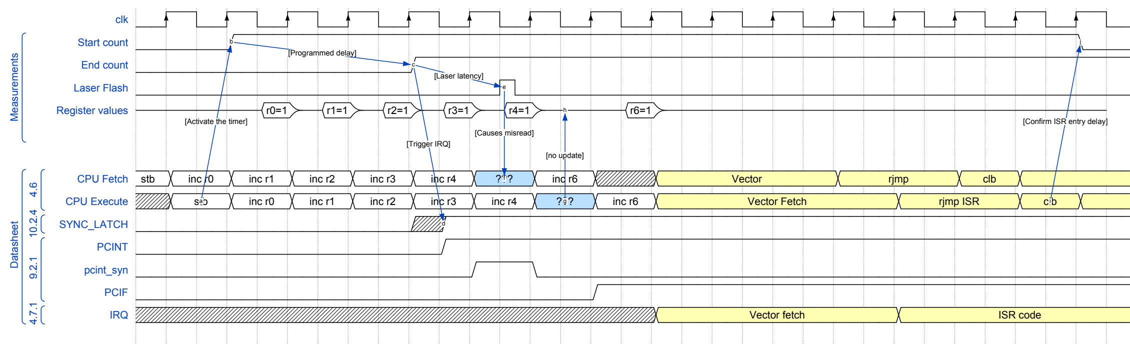


Figure 3-18: Signal Timing

3.3.5.3 Power and Aperture

Two questions remained unanswered. How much power is needed to induce an error, and what is the effect of varying the target area size?

Many factors influence the amount of energy that must be injected into the DUT to induce an error. At the quantum level, we know the energy required to promote electrons into the conduction band (See Section 2.3.3.4), and the appropriate choice

of laser wavelength can control this. In order to make a transistor switch, a critical number of the semiconductor's atoms need to be excited, and this depends on the size of the transistor. At NIR wavelengths, the silicon is semi-transparent, so some of the light may pass through the device without effect. At shorter wavelengths, the silicon is opaque, so light may not penetrate, and only the exposed surface can be influenced. In a top-side attack, metal layers obscure some components and cause light scattering, so power and spot size could conceivably influence the size and shape of the affected zone.

We aimed to show whether error repeatability was affected by either, or both, of the laser spot size and power.

We defined an area 20×18 locations, centred on, and larger than, zone **D**. A test program was run, and the laser fired at 20 time intervals covering a range of different instructions, see Figure 3-19. This program was repeated 25 times at each location. We counted the total number of errors and the number of unique errors for each location. Each experimental run collected $180,000$ samples, and these runs were repeated for all combinations of five different aperture settings and five power settings.

```
1   Expt:   rcall Setup       // Zero all registers
2
3   E1_1:   sbis  0x16,2      // while (!LAMP_RDY);
4           rjmp  E1_1        //
5           sbi   0x18,1      // Set START signal
6
7                             //   Cycle,    Laser time
8           ldi   r0, 7       //       0,    0.2 -  1.1
9           mov   r1, r0      //       1,    1.2 -  2.1
10          add   r0, r1      //       2,    2.2 -  3.1
11          cp    r0, r1      //       3,    3.2 -  4.1
12          inc   r0          //       4,    4.2 -  5.1
13          brne  fwd         //       5+6,  5.2 -  7.1
14          inc   r1          //       ?,
15  fwd:    inc   r0          //       7,    7.2 -  8.1
16          nop               //       8,    8.2 -  9.1
17          nop               //       9,    9.2 - 10.1
18          nop               //      10,   10.2 - 11.1
19          nop               //      11,   11.2 - 12.1
20          nop               //      12,   12.2 - 13.1
21          ...
```

Figure 3-19: Power and Aperture Test

The executed code's only requirement was that errors could be recognised if induced. As per the experiments in Section 3.3.5, we recorded the DUT's state at the

end of each run and used this data to identify corrupted memory and registers.

It was the time required to perform $180,000 \times 5 \times 5$ discrete runs of the code that led to the decision to concentrate on the area surrounding zone **D**, rather than to scan the whole IC. This area includes a known sensitive zone, a boundary zone where scatter effects may be expected and a previously identified insensitive area.

The laser pulse from the YAG laser lasts approximately 4 ns. In a low power setting, it emits up to 2 mJ per pulse [124], and we varied the power from 10% through to 30% in 5% increments. The full aperture of the laser cutter was 40 µm × 40 µm, and we varied it between 20% and 60%. The microscope lens had a 35% transmission ratio at our chosen wavelength of 532 nm (green). These parameters enabled us to calculate an approximation[¶] to the actual power delivered in each experiment. The results are shown here in Table 3.10.

The most notable feature within Table 3.10 is the high degree of repeatability. The initial experiments targeting individual instructions, see Section 3.3.5.1.6, suggested repeatability of approximately 60%. Those figures were based on sets of four samples per test. This data set had 25 samples per test and showed that results that appeared to be unique within a small set of 4 samples were probably examples of repeatable but less common errors.

This table also shows that significant errors are obtained from relatively low power pulses. We see that 4 µJ in a small focussed area reliably induces repeatable errors, as demonstrated at Aperture:20% and Power:15%. The YAG was unnecessarily powerful. All of the errors demonstrated here were induced by pulses using less than one-tenth of the YAG's low power setting.

[¶]It is approximate because the laser power declines with age, and we have no direct way to measure this. These figures will probably overestimate the actual power delivered.

Table 3.10: Errors by Power and Aperture.

	Aperture[a]														
	20%			**30%**			**40%**			**50%**			**60%**		
Power[b]	µJ[c]	N[d]	R[e]	µJ	N	R	µJ	N	R	µJ	N	R	µJ	N	R
10%	2.8	0	0%	6.3	101	85%	11.2	1842	93%	17.5	8142	97%	25.2	9020	95%
15%	4.2	2642	95%	9.5	3729	95%	16.8	6934	96%	26.3	7848	96%	37.8	10219	95%
20%	5.6	3640	96%	12.6	4952	92%	22.4	6604	95%	35.0	5518	95%	50.4	8362	95%
25%	7.0	4127	95%	15.8	5803	96%	28.0	5676	95%	43.8	8208	96%	63.0	10773	96%
30%	8.4	4543	96%	18.9	5899	96%	33.6	4797	94%	52.5	8738	96%	75.6	9690	97%

[a] Aperture diameter as a percentage of 40 µm
[b] Laser power setting as a percentage of 2 mJ per pulse.
[c] Energy delivered to the chip surface after masking by the aperture and losses in the microscope optics.
[d] Total count of errors from 180,000 samples taken within Zone D.
[e] Percentage of errors that are duplicates for a single location and stimulus timing. Where $Repeatability = (TotalErrors - UniqueErrors)/TotalErrors$.

3.4 Data and Interpretation

The results from zone **D** suggest a failure of the prefetch of the next instruction. Therefore, our earlier attempts to categorise the effects of targeting a specific instruction had been flawed for this zone in particular. It is likely that the intended instruction was unaffected, and the observed errors were due to the misbehaviour of the subsequent miss-fetched NOP. This behaviour probably also accounts for the observation that, for arithmetic tests, we regularly saw the same pattern of flag corruption without seeing register corruption. It appears likely that the misread NOP performed a repeatable but unidentified flag modifying operation.

Repeating tests using other samples yielded the same pattern of results. This comparative testing was performed using a 50×50 grid while keeping the laser spot size consistent with the equivalent 100×100 scans. The laser was also only fired during the third quarter-cycle, where earlier tests had shown a higher percentage of induced errors within the collected samples. This reduced coverage saved a significant amount of time but still showed the same pattern of sensitive areas, and the nature of the errors was consistent with the more intensive scans.

For a top-side focussed laser attack on this device, the two dominant failure modes are timing-sensitive instruction skipping and time-insensitive memory editing/corruption.

3.5 Summary

Through this series of experiments, we identified behaviour that obstructs accurate data collection, and we revised our test environment to naturalise this effect. We demonstrated a practical approach to characterising a device, and that characterisation results are transferable to other samples of the same device. Weaknesses identified in the test equipment have been overcome to enable more sophisticated and powerful attacks to be performed.

The test environment affects the results and needs to be considered when automat-

ing large numbers of tests. Individual errors are similar, but failure to reset cleanly before subsequent tests means the data gathered may represent the aggregate effects of multiple errors. This result in itself is interesting as it suggests multiple pulse attacks may have effects beyond those expected by simply combining the effects of two discreet tests. For these cases, it appears that one plus one does not necessarily equal two. The effect is common enough to invalidate data recovered by automated test campaigns unless special measures have been taken to ensure a complete reset of the DUT.

The automated approach for collecting error responses by executing sample code and scanning the whole chip is also practical. More importantly, the results from one sample are transferable to other samples of the same device. Thus a μC component, with its development kit, can be obtained via normal distribution channels and profiled. The knowledge gained can be used to attack third-party devices that utilise the same component. High-security devices such as smartcards usually have restricted access to development tools and engineering samples, so the ability to characterise a component and apply the knowledge to an unrelated device has limited practical value. However, it is a potent threat in the IoT environment, where devices regularly use generic components that are readily available through traditional distribution channels.

The most notable observation is that errors are repeatable. Above a minimum power threshold, the ratio of repeatable errors is both high and remarkably consistent. This effect does not appear to change significantly as the power increases, first reported in, Kelly [96].

Repeatable errors are easier to exploit from an attacker's point of view. Knowing what effect to expect from an attack on a particular site assists in interpreting recovered data. This knowledge, combined with SPA specific features, enables code features such as loop counters to be targeted and manipulated. Similarly, knowing the vulnerabilities and the relative ease of inducing them assists developers when specifying countermeasures. Defensive code can then be deployed efficiently in appropriate proportion to the threat.

We could not readily identify specific sites where a laser pulse affected particular operations, e.g. corrupting arithmetic results without corrupting branching or memory update. For this device, the dominant effects are instruction skipping and memory/register corruption. Recognising such dominant effects is highly valuable when planning defensive code.

This characterisation of error responses assists attackers by enabling them to understand the effects they are inducing while also reducing the time needed to collect and interpret samples. It is equally valuable to a defender who can design defences that fail safely in the dominant identified error modes.

It is also clear that the YAG laser workstation has many drawbacks. It is unnecessarily powerful. Designed as a micro-machining tool, it is as capable of destroying the DUT as it is of inducing errors. We have seen that modest levels of power can induce errors of interest to an attacker. The YAG laser requires a pre-charge before discharge, limiting the rate at which consecutive pulses are generated. This delay has previously been identified as evidence that multiple-pulse attacks are prohibitively expensive or difficult to implement [23][‖] [118][**].

Here we have evidence that the power required is modest and within the capabilities of laser diode technology. Such devices do not require pre-charging and do not necessarily limit the repetition rate.

[‖] "Such a tight timing is a[sic] very difficult to achieve with the current fault injection techniques, which require a non-negligible amount of time to reset the fault inducing means ..." — Barenghi 2010

[**] "... it requires a much more costly fault injection equipment and very high synchronisation capabilities. It is then not yet considered as a realistic threat." — Moro 2015

Chapter 4

A New Laser Workstation

Contents

4.1 Components **141**

4.1.1 Laser Diode 142

4.1.2 Refinements 144

4.1.3 Validation 146

4.1.4 Integration 148

4.1.5 Functional Testing 150

4.2 Multi-Pulse Proof of Capability **152**

4.3 Creeping Barrage - Blind Attack on Known Code **158**

4.4 Summary **161**

Limitations within our original laser pulse injector prevented us from confirming our speculations relating to the exploitability of repeatable errors. Other observations relating to the required power and focus of a laser pulse suggested we could build a superior replacement laser injector. This exercise confirmed our earlier speculations and demonstrated that sophisticated laser attacks could be implemented for a surprisingly low equipment cost.

In this phase of the study, we aimed to confirm the observation that a well-timed and well-aimed laser pulse could cause instruction skipping and discover if the effect can be repeated at short time intervals. While the YAG laser had been used to demonstrate one-off instruction skipping, its long recharge time made multi-pulse attacks impossible, and we found no other identifiable reports demonstrating sequential targeted instruction skipping. This recharge characteristic of industrial lasers has in the past been cited as evidence that multiple-pulse attacks are prohibitively expensive or difficult to implement [23, 118].

A homemade workstation would demonstrate a more realistic, heightened threat of attack. It would also show that defence characterisation and testing could be incorporated into modest development plans, potentially increasing security at this vulnerable, budget end of the market.

We set ourselves three goals,

1. Demonstrate the selective skipping of instructions executing within a small number of machine cycles of each other, using multiple discrete laser pulses.

2. Show a white-box attack on a known code sample without the need to know precisely when the code was being executed.

3. And, the component cost of the solution should be affordable to an amateur attacker or low-budget laboratory.

4.1 Components

The primary requirements of the device were simple and were based on observations from the preceding experiments. We need a suitably powered laser that could be switched on and off at approximately 40 MHz. We need a mechanism to turn the laser on and off that could be synchronised with the DUT. Finally, we need a way to focus the laser onto a spot approximately 20 μm^2.

4.1.1 Laser Diode

Upon initial review, this looked like an impossible goal. We needed a diode that could be switched on and off at up to 40 MHz; i.e. within 25 ns. Furthermore, the data in Table 3.10 suggested that we would need close to 5 µJ per pulse to induce errors. Delivering the equivalent amount of energy in a 25 ns pulse would require a 200 W laser with no losses during transmission through the focusing optics. Such high power would be unachievable with readily available diodes. However, properties of the YAG's pulsed output and the properties of silicon semiconductors (described in Section 2.3.3.4) suggested that longer pulses would generate the same effects at significantly lower power.

The YAG laser delivered this energy in a 4 ns pulse, and this six times faster than our anticipated 25 ns diode pulse. We speculated that to achieve a switching effect in the DUT the electrons in a transistor's gate would need to be excited across the band gap (See Figure 2-14) for a significant fraction of a machine cycle and that the energy delivered by the pulse would dissipate very quickly through the semiconductor and local metal tracks. We had seen this effect when testing the laser response times; see Figure 4-6. Thus we anticipated a similar error inducing effect could be achieved with significantly less power applied steadily for a longer period. The first task was to confirm this; otherwise, plans would need to be changed.

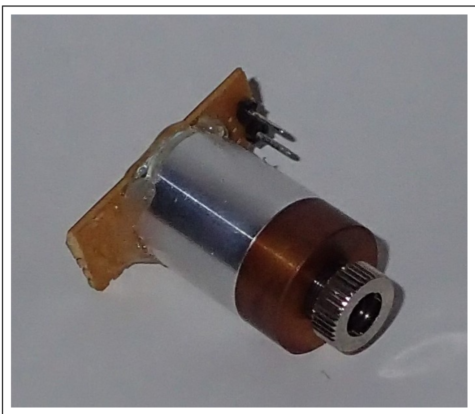


Figure 4-1: Laser Diode with Heatsink and Lens

Readily available laser diodes fall into three main categories.

i) High power and fast switching for telecommunications are unsuitable. They are designed to run continuously at medium power with high-frequency data signal modulating between medium and high power modes [131]; from low power to medium power, these devices typically take several ms.

ii) Continuous-Wave mode diodes, as the name

suggests, are designed to be reliable when driven continuously. They are limited by their packaging's ability to dissipate heat and consequently need to run at modest power. Continuous-wave diodes typically switch on and off relatively slowly, and this property precludes the option of over-powering them and operating in pulse mode.
iii) Pulsed-Mode diodes operate at a low duty cycle. Therefore, if used correctly, they do not generate much heat, and as a consequence, can tolerate operation with higher momentary power output. Because of this, they are also designed to reach peak output relatively quickly. For our application, only pulse-mode capable diodes would be suitable.

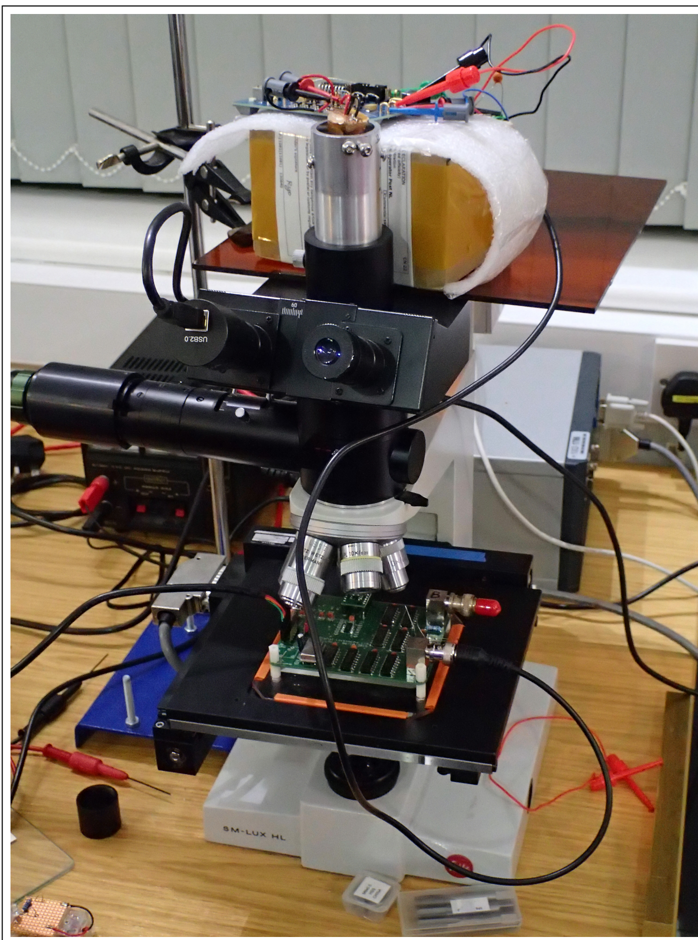


Figure 4-2: Laser Station

We required a diode that could operate in pulsed mode and deliver multiple Watts of power. The choice was between NIR or blue/violet, approx. 450 nm. At other wavelengths, the options available offered insufficient power or slow response times. We chose a 455 nm laser diode salvaged from a high-intensity Nichia NUMB80 laser diode bank [125]. It offers up to 4.3 W of power and is capable of switching at 200 MHz. Available on eBay at the time for less than £30. The other advantage is that working with visible light is significantly easier than NIR. Besides the obvious advantage of seeing the light, readily available microscope optics are optimised for this part of the spectrum. If the DUT is visibly in focus through the eyepiece, it will be in focus for the laser

too.

We obtained an old trinocular Leitz SM-LUX HL microscope, again via e-bay, at the cost of about £200. We mounted the laser on the vertical camera port, a Charge-Coupled Device (CCD) camera in one of the eye-pieces holders and left the third eyepiece in situ for direct observations. For focussing and positioning the DUT, the eyepiece was most helpful but obviously unusable when the laser was powered up. Safety measures involving screens and goggles were taken to avoid the risk of eye damage from reflected laser light.

Driving the laser-diode at full power requires a circuit capable of switching a 3 A current without creating excessive spikes or reverse voltages which can be fatal to a diode. We used a dedicated laser controller device, the iCHaus iCHG 3A Laser switch [86], which is typically used for LiDAR* and data transmission. It is conveniently available through distributors on an evaluation board for £70 (2017 prices).

4.1.2 Refinements

The initial experiments used the controller board developed for the categorisation experiments described in Section 3. This controller triggered the YAG laser to fire upon the rising edge of a Transistor-Transistor Logic (TTL) signal delivered through a coax cable to the laser's controller module. The signal pulse indicated the firing time and its duration with this arrangement. Activation of our laser proved to be unreliable, with the diode frequently failing to fire. The reasons for this can be seen in the oscilloscope trace, Figure 4-3. The blue trace shows the DUT controller's 5 V signal sent to the laser controller. The red trace shows the signal received by the laser controller. The received signal varies between 0.4 and 0.5 V and this is insufficient to reliably switch TTL logic.

We also attempted to use the DUT to send a signal to a stand-alone signal generator. The signal generator was configured to deliver a pre-programmed pulse pattern to the diode controller. In this configuration, the DUT controller provided the synchronisation signal as the rising edge of a longer pulse, in the same way as had previously

*Laser Imaging, Detection, and Ranging (LIDAR)

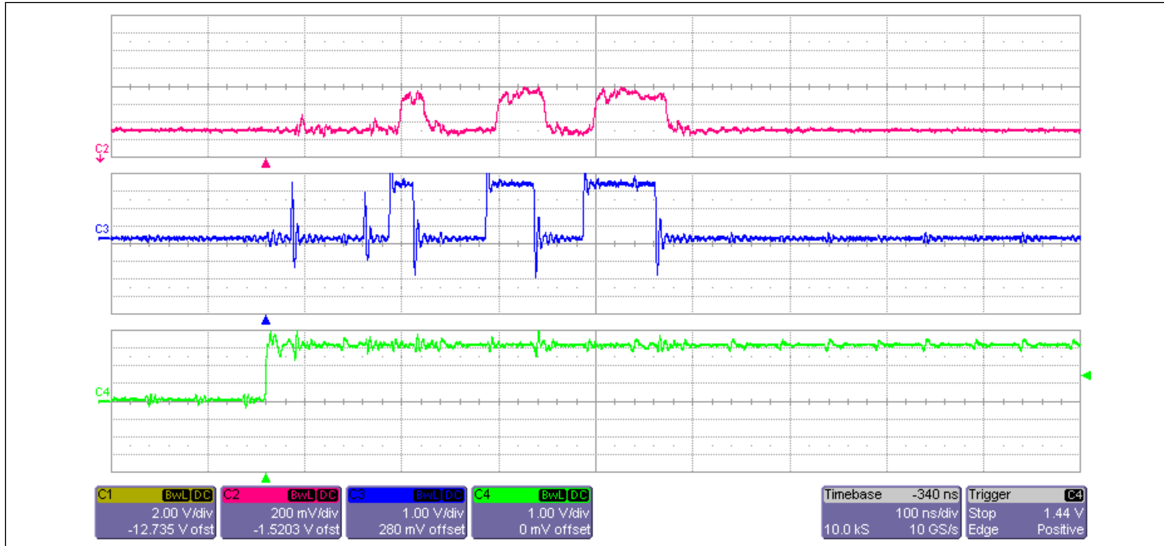


Figure 4-3: Trigger Signal Propagation

Figure 4-4: Laser Control After Signal Propagation

worked with the YAG. This mechanism generated reliable laser activation but with an unacceptable amount of timing jitter. It became clear that a new DUT controller board would be required to overcome the shortcomings of the laser trigger mechanism. The details of this new board are described in Appendix B.4.

The replacement DUT controller board had two significant improvements. First of

all, it delivered the laser firing and timing signal via Low-Voltage Differential Signaling (LVDS). LVDS is used for high-speed digital signalling in Universal Serial Bus (USB) and ethernet. It uses a twisted pair of wires for each signal. This mechanism proved to be highly reliable, as can be seen in Figure 4-4; here the DUT controller generates a 25 ns pulse. The image shows the voltages on both the anode (red) and the cathode (yellow) of the laser diode.

The second significant change to the controller board was the incorporation of a Field-Programmable Gate Array (FPGA) to replace the secondary μC and the timer circuitry used in its predecessors. We programmed the FPGA to provide multiple high-speed counters, giving us control of both the timing and duration of multiple laser pulses. All counters were synchronised with a synthesised 200 MHz clock signal, which was also used to derive a 10 MHz clock signal to drive the DUT. We used an evaluation board [128] supporting a Xilinx Spartan-6 gate array [186]. Such boards are widely available and cost less than £30.

4.1.3 Validation

We now had a DUT-controller that could generate laser control signals with 40 MHz resolution. The control signal signals could be measured and verified for timing accuracy at the connections to the laser diode. We still needed to confirm that the laser's light output matched its control signals.

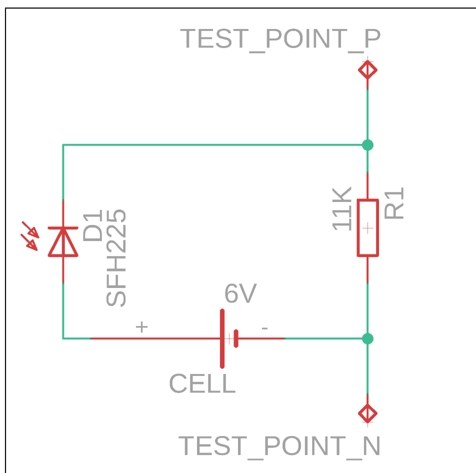


Figure 4-5: Laser Test Rig

We constructed a simple light detector circuit, shown in Figure 4-5. A smooth power supply, provided by an battery of alkali cells, was placed in series with a photodiode and a resistor. The voltage across the resistor indicated the current flow and, therefore, the open/closed state of the photodiode. When driven by a 100 ns pulse, repeated at 10 kHz we observed the trace shown in Figure 4-6. The laser responded quickly to the 'on' signal, but it remained unclear whether the

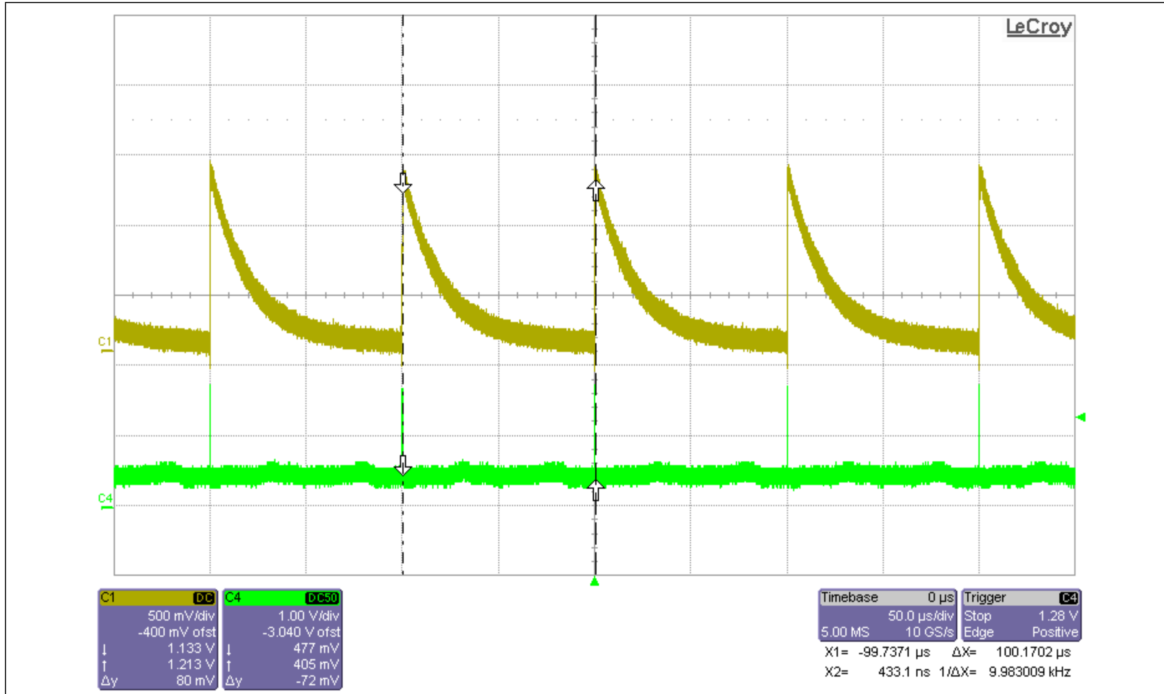


Figure 4-6: Regular Pulse

laser continued to emit light after the signal had ended.

Figure 4-7 shows three 25 ns pulses delivered in quick succession, again demonstrating that the 'on' response is both fast and distinct but is equally ambiguous about the 'off' behaviour.

Two possibilities were considered. It could be capacitance and exponential discharge within the test circuit or an after-glow effect where the laser emits light after the driving signal ends. We reduced the size of the resistor, R1 in Figure 4-5, and saw that the rate of decay increased; this was indicative of the capacitance hypothesis. However, the signal also became very noisy with 'ringing' effects, and the distinction between closely spaced pulses could no longer be identified. In a final test, we left the laser on high power and directed it at the sensor that was itself obscured by a spinning disk. The spinning disk, a standard 60 mm radius Compact Disk (CD), had a 0.1 mm hole drilled close to the circumference. When spun at 50 Hz the hole's transition time was 5.3 µs. At higher rotational speeds, this apparatus became mechanically unstable. The received power trace showed an asymmetric profile, with a steep rise and an extended decay time. Whilst not proving the absence of a laser after-glow, it

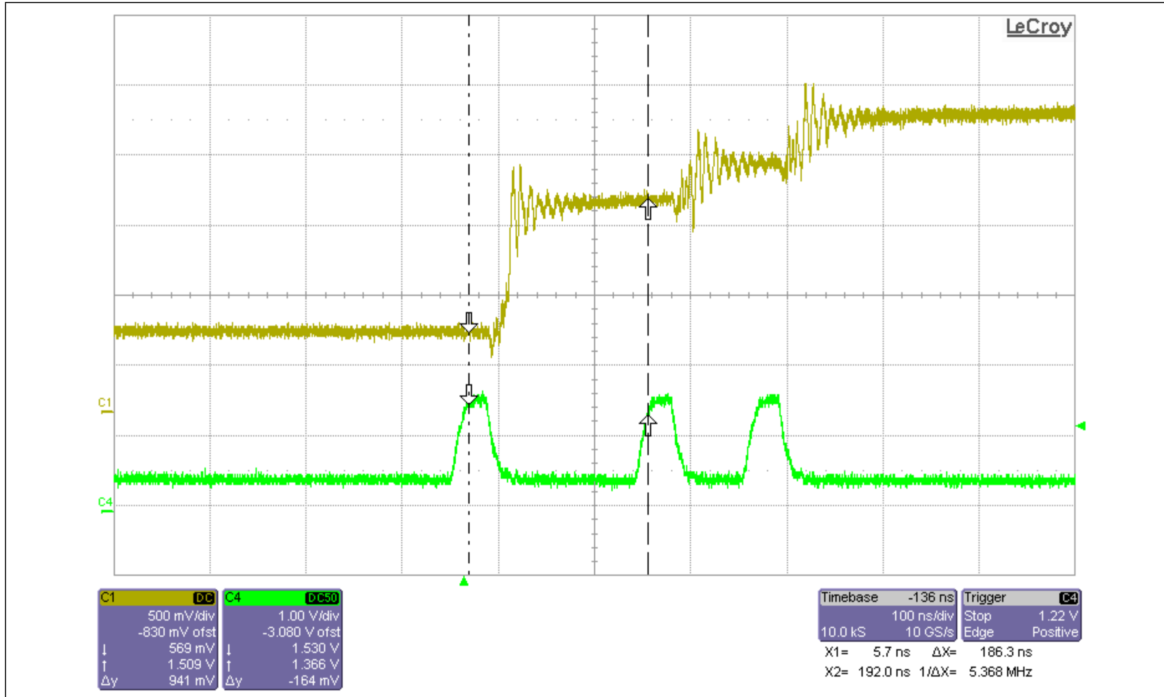


Figure 4-7: Laser Tripple Pulse

was indicative of significant capacitance in the detector circuit. Demonstrable capacitance in the detector and a demonstrable absence of power input to the diode gave us confidence that the laser was switching off reasonably quickly.

4.1.4 Integration

Figure 4-8: Alignment Basic

The next challenge involved integrating the sub-components into a usable workstation.

The laser diode (seen in Figure 4-1) needed to be mounted so that it could be focussed directly through the microscope's objective lens. This alignment was achieved by placing the diode at one end of a pipe and mounting the pipe in the microscope's camera port; see Figure 4-2. The microscope's camera port provided a 38 mm aperture, and aluminium tube is readily available at exactly this size. The diode was held in place with four pinch screws, North-South at the top of the diode's heat sink and East-West at the bottom. Adjustments to these screws en-

abled the fine[†] adjustments to be made to the position and orientation of the diode, Figure 4-8. Crude alignment was first achieved by placing a target paper screen at the mouth of the tube, running the laser at low power to make a visible spot, and adjusting the screws to align the spot on the centre of the target.

The laser spot, visible via the CCD camera in the microscope, is a reflection seen on internal optics within the microscope's body tube. When the laser brightness was increased to see the spot projected onto the target, this reflection became so bright that it saturated the CCD. Figure 4-9 shows the beam, seemingly aligned with the centre of the image. However, the scorch-mark to the right of the image shows the point on the target where the beam actually focussed.

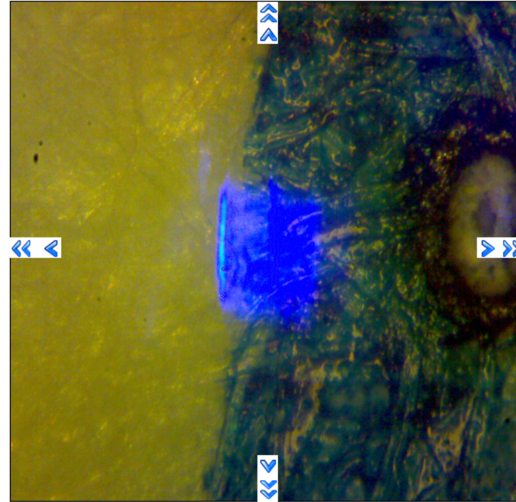


Figure 4-9: Alignment Parallax

Figure 4-10 shows the virtual image and the target aligned but displaced. The final accurate alignment was achieved by increasing the length of the diode's support tube. While this made the adjustments marginally more sensitive, it proved easier to align the reflection and scorch-mark with the centre of the image. This alignment with the centre of the lens was essential; otherwise, realignment would be necessary each time the objective lens is changed to a different magnification.

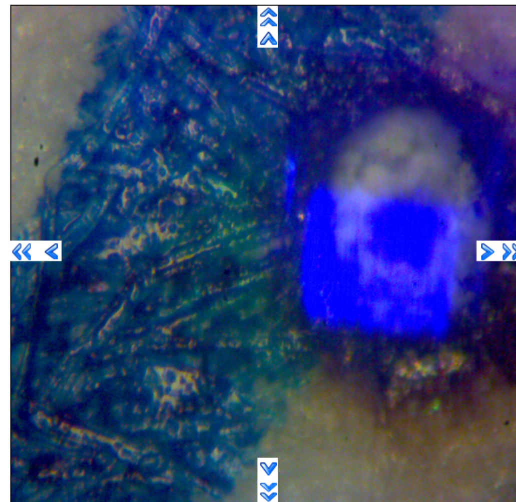


Figure 4-10: Alignment Offset

The laser controller board was mounted next to the diode to minimise the wire

[†] An M4 threaded bolt gives 0.7 mm of travel per 360°of rotation.

lengths and improve the signal quality for the fast switching high current laser supply. The FPGA was mounted directly below the DUT on the support circuit board to ensure clean signals between the DUT and the timer controller. The two parts of the system were connected via a standard 1 GHz ethernet patch cable.

4.1.5 Functional Testing

We tested the new workstation by rerunning the experiments used to test the instruction skipping hypothesis, described earlier in Section 3.3.5.2.1. With a 2 A current driving the laser, we could reliably repeat the results previously obtained using the YAG laser. From the datasheet [125] describing the laser diode, a 2 A current should deliver 2.5 W and a 25 ns pulse would therefore deliver 630 nJ. This value was approximately 1% of the energy delivered by the YAG to achieve the same effect. It

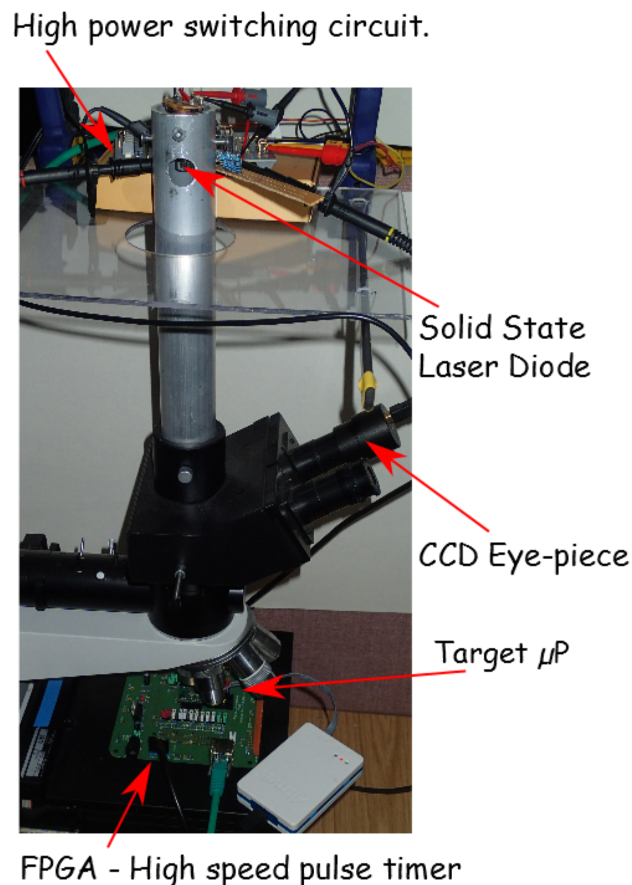


Figure 4-11: Workstation

demonstrated that sustained low power for a significant fraction of a machine cycle was equally effective as a very short pulse of high power. This result supported our speculation made in Section 4.1.1; removing the last of the uncertainty about the viability of our approach.

We now had a fully functional, programmable laser workstation; Figure 4-11. This budget device was capable of generating multiple pulses with 40 MHz resolution and of inducing errors with the same properties as the expensive YAG laser that it replaced. We had also confirmed an earlier observation that at a particular *Sweet-Spot* on the DUT's surface, we could reliably induce instruction skipping behaviour with a laser pulse synchronised with the third quarter cycle of the device's instruction clock.

4.2 Multi-Pulse Proof of Capability

The primary driver for building the new laser was to be able to generate rapid consecutive laser pulses. Such quickfire pulses would enable us to test the predictions of earlier work. Firstly, it would test the hypothesis that multiple suitably timed laser pulses would induce multiple repeatable error effects and that each induced error was independent of preceding errors. Secondly, it would dispel the myth that attacks requiring multiple closely timed laser pulses would be prohibitively expensive to implement.

To do this, we devised the short code fragment shown in Figure 4-12. It consists of a sequence of conditional branch instructions with each decision path leading to another similar conditional branch. After four such branches, the program flow terminated at one of sixteen possible end states, each identified by loading a register with a unique value.

```
10:          breq   H
11:  L:      breq   LH
12:  LL:     breq   LLH
13:  LLL:    breq   LLLH
14:  LLLL:   nop
15:          nop
16:          nop
17:          nop
18:          ldi    r24,'0'
19:          ret
20:  LLLH:   nop
21:          nop
22:          nop
23:          ldi    r24,'1'
24:          ret
25:  LLH:    breq   LLHH
26:  LLHL:   nop
27:          nop
28:          nop
29:          ldi    r24,'2'
30:          ret
31:  LLHH:   nop
32:          nop
33:          ldi    r24,'3'
34:          ret
35:  LH:     breq   LHH
36:  LHL:    breq   LHLH
37:  LHLL:   nop
38:          nop
39:          nop
40:          ldi    r24,'4'
41:          ret
42:  LHLH:   nop
43:          nop
44:          ldi    r24,'5'
45:          ret
46:  LHH:    breq   LHHH
47:  LHHL:   nop
48:          nop
49:          ldi    r24,'6'
50:          ret
51:  LHHH:   nop
52:          ldi    r24,'7'
53:          ret
54:  H:      breq   HH
55:  HL:     breq   HLH
56:  HLL:    breq   HLLH
57:  HLLL:   nop
58:          nop
59:          nop
60:          ldi    r24,'8'
61:          ret
62:  HLLH:   nop
63:          nop
64:          ldi    r24,'9'
65:          ret
66:  HLH:    breq   HLHH
67:  HLHL:   nop
68:          nop
69:          ldi    r24,'A'
70:          ret
71:  HLHH:   nop
72:          ldi    r24,'B'
73:          ret
74:  HH:     breq   HHH
75:  HHL:    breq   HHLH
76:  HHLL:   nop
77:          nop
78:          ldi    r24,'C'
79:          ret
80:  HHLH:   nop
81:          ldi    r24,'D'
82:          ret
83:  HHH:    breq   HHHH
84:  HHHL:   nop
85:          ldi    r24,'E'
86:          ret
87:  HHHH:   ldi    r24,'F'
88:          ret
```

Figure 4-12: Branch Tests

Figure 4-13 shows the possible paths through the code. The tree assumes an error results in an instruction skip rather than a branch to an indeterminate location. This behaviour had been observed in the previous characterisation experiments.

When a branch is taken, it takes two machine cycles on this μC. Therefore an unperturbed path down the tree's branches should take 8 machine cycles, giving 32 discrete opportunities to inject errors. Branches not taken, and most other instructions take a single machine cycle to execute. Thus erroneous execution could reach

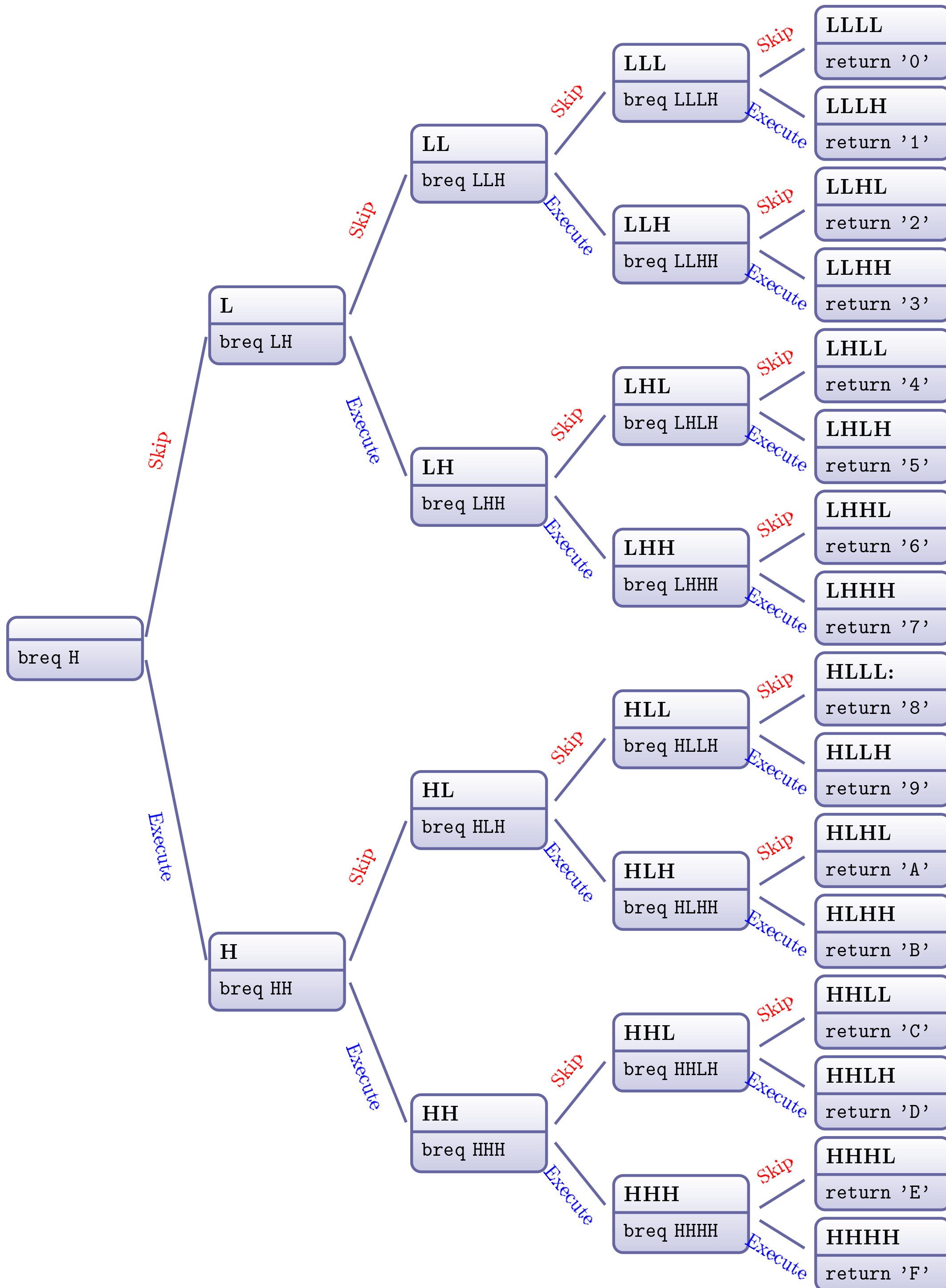


Figure 4-13: Branch Test Execution Paths

a termination node in just 4 cycles. To avoid the possibility of firing the laser at the termination node's code and potentially confusing the identifying response, we added

NOP instruction to ensure all paths through the tree took 8 cycles to reach the end state. The laser could then be fired at all 32 relevant time intervals, as shown by the 'Pulse clk' signal in Figure 4-14, without risk of hitting the code that reported the outcome.

Figure 4-14 also shows the execution paths and the pulse timings required to reach the 16 possible outcomes. It also demonstrates the significance of the NOP instruction and their role to delay the execution of RET instruction until after the moment the final pulse may be delivered. The presumption here is that a NOP, whether skipped or not, will be immediately followed by the same instruction. Therefore we do not need to consider the effect of skipped NOPs; i.e. the instruction pointer or instruction timing are unaffected by skipping. In this diagram, instructions arrived at as a result of a skip are shaded blue. The red lines link skippable instructions with the resulting 'blue' out-of-sequence instruction. The code identifying termination nodes is orange and was not attacked.

In each test run, the ZERO flag was set to state **H**igh, and under normal circumstances, the execution path would take each branch it encountered, ultimately ending up at node HHHH, and return the value 'F'.

For the first real test of the equipment, we generated all patterns of $1-4$ pulses within the 32 quarter cycle window and repeated it at 9 target sites, on and surrounding in the previously identified *Sweet-Spot*. Each test generated 41 thousand response samples and took three hours to complete. The device operated continuously for a day. We found examples of all 16 possible outcomes within the results, but the sheer volume of result data proved difficult to interpret. By far, the vast majority of pulses had no effect. An error induced by a single pulse would be repeated for all ineffective permutations of the other 3 pulses.

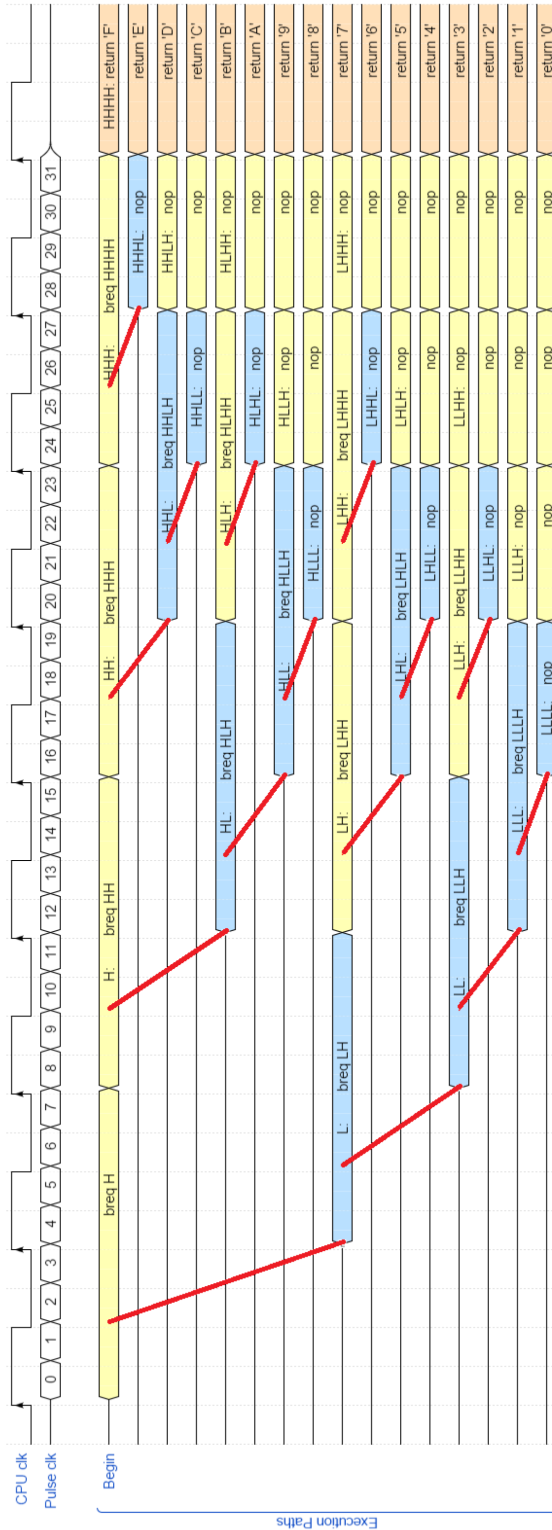


Figure 4-14: Branch Test Execution Paths

The test time rises geometrically with the number of execution cycles being examined. For a code fragment offering n intervals in which to inject pulses, the number of samples to collect when using up to X pulses is given by,

$$Samples = \sum_{p=1}^{X} \binom{n}{p} \qquad \text{where } \binom{n}{p} = \frac{n!}{p!(n-p)!} \tag{4.1}$$

Equation (4.1) demonstrates an important feature, the time to perform exhaustive searches soon becomes prohibitively long, the volume of results becomes large and difficult to interpret. Table 4.1 shows figures based on the above experiment. The apparatus executed the experiment 4 times per second, and each run offered $n = 32$ pulse injection opportunities.

Table 4.1: Test Pulses, Permutations and Time.
($n = 32, at\, 4\, Samples\, s^{-1}$)

Pulses (X)	**Samples**	**Time**	
1	32	8.0	Seconds
2	528	2.2	Minutes
3	5,488	22.9	
4	41,448	2.9	Hours
5	242,824	16.9	
6	1,149,016	3.3	Days
7	4,514,872	13.1	
8	15,033,172	43.5	

Armed with the knowledge that, at the *Sweet-Spot*, the skipping effect was associated with pulses coinciding with the third quarter cycle, we repeated the test. This pruning reduced the time to test a single site from 2.9 hours to 40 seconds, enabling us to repeat each test multiple times to compare the consistency of the outcomes. Once again, we found examples of all 16 outcomes within the collected data.

Table 4.2: Jump Matrix Termination States

Outcome	**Pulses**[a]	**Patterns**[b]	**Samples**[c]
0	4	1	4
1	3	5	20
2	3	5	20
3	2	11	44
4	3	5	20
5	2	11	44
6	2	11	44
7	1	15	60
8	3	5	20
9	2	11	44
A	2	11	44
B	1	15	60
C	2	11	44
D	1	15	60
E	1	15	60
F	0	15	60
corrupt		0	0
Total		162	648

[a] Minimum number of errors required to reach this outcome. [b] Unique pulse patterns that yield this outcome. [c] Samples collected after repeating each pulse pattern 4 times.

Table 4.2 shows the results for the test at the sweet spot. As expected, where pulses coincided with the fetch cycle for a branch instruction, we saw branch skipping, exactly as predicted. Where pulses coincided with the second CPU-cycle of a branch instruction or with one of the NOPs, we saw no effect. For example, the outcome '0' was reached consistently with the pulse timings of 3, 7, 11 & 15. We saw five paths reached outcome '1', requiring 3 well-timed errors — 1 trace of 3 pulses got there as well as 4 traces of 4 pulses, where 3 of these pulses were well-timed, while the fourth had no effect.

4.3 Creeping Barrage - Blind Attack on Known Code

We have seen that exhaustive searches for pulse patterns that induce errors are practical for small code fragments. However, in more extensive programs, it is not always possible to identify a short enough region of interest that could be searched within a reasonable time frame.

If a pulse pattern could be identified that would compromise a small defensive code structure, then could that pattern be applied to a larger code sample where the time at which the defence is applied is unknown? Furthermore, particularly relevant for the instruction skipping effect; if we knew the source code, could we predict the appropriate pulse pattern and test this pattern over a wider time window? The same way a single pulse attack can search for a weakness in a time frame proportional to the search window size, a multi-pulse pattern could be used with no time penalties.

We chose to attack an implementation of Speck [28]. We used an AVR version freely available on Github [10]. This implementation had previously been studied by Breier [39] for DFA via static code analysis and simulation of the effects of errors.

The code implementing the encryption rounds is shown in Figure 4-15. We looked at the code and quickly identified an obvious weakness; the loop counter and loop termination test; seen in lines 36 and 37. Since this would be vulnerable to a single pulse attack, we duplicated the test; lines 40 and 41. This style of double testing is typical of defended code. The test to terminate and deliver a result is repeated and, if the conclusions differ, a `Trap()` function is entered from which there is no return.

Knowing the code, and in particular the defence, we could predict that skipping the instructions at line 38 and line 41 would result in a premature exit from the loop. These instructions execute 2 cycles apart if the first of them is skipped. Thus, we would expect to bypass the test and the defence by presenting a pattern of two pulses, 8 quarter cycles apart, to the DUT at all timing intervals where we suspect the round body is being executed. If failure could be triggered at the earliest opportunity, the resulting data would be the input data after round one using $Subkey_0$.

This code was invoked from a simple 'C' program, part of which is shown in Figure

```
1   loop:   // Enter with Z = @Subkey[0],  r0..7 = Plaintext
2           ld   r12, z+            // r12-15 = Sub Key[round]
3           ld   r13, z+
4           ld   r14, z+
5           ld   r15, z+
6           add  r5,  r0            //  Xh += Ror8(Xl)
7           adc  r6,  r1
8           adc  r7,  r2
9           adc  r4,  r3
10          eor  r12, r5            // Subkey ^= Xh
11          eor  r13, r6
12          eor  r14, r7
13          eor  r15, r4
14          lsl  r0                 // Xl = rol3(Xl)
15          rol  r1
16          rol  r2
17          rol  r3
18          adc  r0,  zero
19          lsl  r0
20          rol  r1
21          rol  r2
22          rol  r3
23          adc  r0,  zero
24          lsl  r0
25          rol  r1
26          rol  r2
27          rol  r3
28          adc  r0,  zero
29          eor  r0,  r12           // Xl ^= Subkey
30          eor  r1,  r13
31          eor  r2,  r14
32          eor  r3,  r15
33          movw r4,  r12           // Xh = Subkey
34          movw r6,  r14
35
36          inc  currentRound       // next round
37          cpi  currentRound, 26   //
38          brne loop               //
39
40          cpi  currentRound, 26   // basic defence against
41          brne trap               // premature exit
42
43  exit:   // Result CipherText = r0-r7
44          ...
45          ret
46
47  trap:   // execution black-hole.
48          ...
```

Figure 4-15: Speck Encryption

4-16. For the tests, we used a *GO* signal (line 15) generated by the DUT to indicate when the encryption rounds were being executed. Clearing the *GO* signal (line 17) simplified the process of working out how long the round processing took. This control could have been provided by an oscilloscope tuned to trigger on identifiable power trace landmarks or even just the DUT's I/O command exchange. The plain text and key data are the reference samples provided by Beaulieu [28] for Speck 64/96.

The twin pulse pattern was delivered at each of the third quarter cycle opportunities, i.e. starting at quarter cycle 3, 7, 11, 15, Etc. The results of this experiment

```
1                                          // Test vectors Speck 64/96.
2   unsigned char initialKeys[12] = { 0x13, 0x12, 0x11, 0x10,
3                                      0x0B, 0x0A, 0x09, 0x08,
4                                      0x03, 0x02, 0x01, 0x00 };
5
6   unsigned char plainText[8]    = { 0x74, 0x61, 0x46, 0x20,
7                                      0x73, 0x6E, 0x61, 0x65 };
8
9   unsigned char cipherText[8];    // 0x9F, 0x79, 0x52, 0xEC,
10                                  // 0x41, 0x75, 0x94, 0x6C
11
12    ...
13   speckKeySetup();               // Generate subKey array
14
15   PORTA |= GO;                   // Signal Encrypt Starting
16   speckEncrypt();                //                Active
17   PORTA &= ~GO;                  //                Ended
18   ...
```

Figure 4-16: Speck Test Control

are summarised in Table 4.3.

Normal unperturbed execution took 111 µs and generated the expected result. The vast majority of tests took precisely this time and generated erroneous results. This outcome was expected because all the instructions in the loop body execute in a single cycle and perform arithmetic that affects the final result. The laser pulses in most experiments would almost certainly cause one or more of these instructions to be skipped.

The shortest execution time corresponds to a pulse pair on quarter cycles 259 and $259 + 8$; this result also matches the expected cypher output after one round. Likewise, the second and third shortest execution times correspond to the execution and premature result delivery of two and three rounds, respectively.

There were multiple examples and a 116 µs execution time. This timing corre-

Table 4.3: Speck Round Attack

	Time (µs)	First pulse cycle No.	Result
Unperturbed execution	111	none	9f 79 52 ec 41 75 94 6c
Most common	111	various	many different
Shortest	9	259	90 e0 c3 ab 0b 93 c8 80
Second shortest	13	423	a4 39 aa 4a f8 a7 ee 4a
Third shortest	17	587	f8 94 2a a7 3d ab 58 f0
Longest	116	various	many different

sponds to an additional round of the algorithm and can be attributed to skipping the loop counter increment instruction.

The choice of the Speck cypher, and a short-key, small-data variant at that, was just for convenience to prove the viability of the attack mechanism. In practice, any loop-based algorithm would be similarly vulnerable. The key feature was the use of knowledge of the implemented defence to prescribe a multiple pulse pattern which could then be tested against the algorithm. The time required to perform the test was linear with respect to the search window size, whereas, in the previous experiments, the search time increased geometrically with the window size.

4.4 Summary

In this phase of the study, we have demonstrated that a functionally powerful laser workstation can be constructed for just a few hundred pounds, and this can be achieved without needing access to anything more than Do-It-Yourself tools. Therefore, a laser workstation is within the budget of any suitably motivated hacker, and the capital cost of the equipment is not a defence in its own right. With a low cost of attack, even low-value targets become worth attacking.

The laser diode can be switched on and off quickly enough to target consecutive instructions, and the effects appear to be independent; i.e. the effect on an instruction appears to be the same whether targeted individually or shortly after a previously targeted instruction. We demonstrated the ability to navigate a path through a multiply branching chain of instructions and guide execution to a chosen termination state. This capability was impossible with the expensive and powerful YAG laser previously used, and this observation had presumably led others to speculate that multiple pulse attacks would be both difficult and infeasibly expensive to implement. This low-cost solution proves that it is inexpensive and straightforward and appears to be the first published demonstration of this capability, Kelly [95].

The ability to selectively execute individual instructions means that a template of pulses can be defined to manipulate an execution outcome. This is particularly

powerful when an attacker knows the code being attacked. In such a scenario, the definition of a multi-pulse template is straightforward, and templates can be tested at all relevant time intervals far more quickly than exhaustive searching of pulse combinations.

As far as pulse timings and repetition rate are concerned, our laser workstation exceeds the capabilities of the equipment it was designed to replace and does so for a fraction of the cost. Though we chose to attack a single μC with visible light from the topside, more recent results from Guillen [80] indicate that similar behaviour can be expected on differing architectures.

The frequently used defensive strategy of repeating tests leads to frequent duplication of simple code structures. In practice, this is often achieved by placing this repetition in macros to ensure consistent application of defences throughout an application. Consequently, there is frequent and predictable use of identical defensive structures. The ease with which multi-pulse templates can be used means that some defended code is almost as vulnerable as its undefended equivalent.

Besides the offensive capability of this workstation, it also has a defensive role. Code developers can use it to characterise and test a variety of defences and identify appropriate defensive techniques for their chosen target μC and application. We explore this defence testing capability in the following section of this report, Chapter 5.

The primary weakness exposed here is the ability to skip instructions, resulting in inaccurate arithmetic or out of sequence execution of code. Forearmed with this knowledge, it should be possible to write code that is designed to fail-safe. The practicalities of such an approach are pursued here later in Chapter 6.

Chapter 5

Testing Security Defences

Contents

5.1 Practicalities **166**

5.1.1 Defensive Coding 166

5.1.2 Fault Model 167

5.1.3 Testing Defences 167

5.2 Defences **169**

5.2.1 Unprotected 171

5.2.2 Double Test 171

5.2.3 Retest at Destination 172

5.2.4 Inverse Test 173

5.2.5 Double Data 174

5.2.6 Data Inverse 174

5.2.7 Checksum 175

5.2.8 Redundant Representation 176

5.2.9 Repeat Calculation 176

5.2.10 Modified Compensated 177

5.2.11 Alternative Algorithm 177
5.2.12 Inverse Calculation 178
5.2.13 Jump Id 179
5.2.14 Waymark - Late Test 179
5.2.15 Waymark - On the Fly 180
5.3 Results and Analysis **181**
5.3.1 Results 183
5.3.2 Normal Termination 186
5.3.3 Trapping 186
5.3.4 Crashing 187
5.3.5 Out-of-Order Processing 188
5.4 Application **188**
5.5 Summary **190**

We use the new laser workstation to reevaluate work that has previously been performed, by others, in simulation. In doing so, we demonstrate that physical attacks provide a practical approach to testing the efficacy of defensive code structures. The exercise also demonstrates that subtle differences in source code can lead to radically different outcomes for defence efficacy. The results and techniques described here are valuable to both attackers and defenders of secure μC *systems.*

One of the initial objectives for this study was to physically evaluate defences that hitherto had been based on speculation or tested through simulation. The suspicion was that simulations overlook many of the real-world practicalities of implementing attacks. The typical approach taken in earlier work on categorising the efficacy of various defences, typified by Theißing [173], was based on random errors injected into a large number of simulation runs. In Chapter 3 we have shown that error injection is far from random and, more importantly, that repeatable errors can be injected. This observation invalidates the Monte-Carlo simulation approach used by others when testing defences. These reliable and repeatable properties opened up the realistic possibility of exhaustive probing of a defence at a previously identified vulnerable site using multiple carefully timed error injection pulses.

The weaknesses of simulation are becoming apparent. Moro [118] produced a formal verification of combinations of instructions showing that immunity from single skipped instructions was both possible and practical. However, for their chosen CPU architecture, the single skip fault model was flawed [148, 187]. Others, such as Laurent [106] and Proy [139], argue that a deep understanding of the device's Instruction Set Architecture (ISA) is required here. Such detailed knowledge will enable a simulation to predict the effect of a faulty instruction based on the details of the pipeline and instruction interpretation hardware. This approach is interesting and relevant for micro-surgical error injection, in much the same way that, Skorobogatov [166], Courbon [55] and others demonstrate bit-level static memory editing. This micro-surgical error injection is, to a large extent, irrelevant when large laser spots induce multiple examples of such errors. The resulting error effect will become dominated by the DUT's circuit layout, a feature that will be beyond all but the most detailed simulations to predict.

While we can be confident about getting multiple, repeatable errors, it is still possible that the individual error effects may differ subtly. For example, the dominant effect we have witnessed is instruction skipping. In reality, the μC is still executing an instruction, and that unknown instruction will probably affect some aspect of the DUT's state. In a simple simulation, we would probably model this behaviour by

assuming the unknown instruction is a NOP, while in reality, it remains unknown.

From a practical perspective, for software developers wanting to test the resilience of their defences or for hackers tinkering with vulnerable consumer electronics, the most straightforward approach is to test the target device physically. An academic archaeologist would approach a dig with a trowel and a brush and ultimately understand the whole context of their site. In contrast, a treasure hunter will go in with a shovel. It is the treasure hunters whom we need to defend against in practice.

This phase of the study used the new laser station to evaluate a range of defences. Such an evaluation would be a practical first step when embarking upon an application development. Defences have a cost in both performance and code volume. Therefore, identifying the most effective defences will minimise these costs.

5.1 Practicalities

The μCs deployed in the consumer electronics world are usually readily available to developers. Ready availability means a would-be attacker has almost unlimited access to samples for testing code and locating appropriately sensitive regions on which to focus the laser. Locating such a *Sweet-Spot* is the first step in executing an attack.

It is unlikely that most attackers would have in-depth knowledge of the DUT's internal layout or understanding of the physical nature of the induced faults. Whilst these details are interesting, Proy [139], such knowledge is not required when considering the consequences of an attack, Schellenberg [154], and these attacks are therefore accessible to a wide range of attackers, including amateurs and those with limited resources.

5.1.1 Defensive Coding

Programmers need to add redundant operations to their code in anticipation of errors to detect and respond to their effects. For example, when considering the consequences of memory corruption, multiple defensive strategies exist. Simple bound checking can ensure the data is within a set of expected values. Alternatively, cor-

ruption can be detected by maintaining duplicate copies of data and confirming they match. There are many defences to many potential errors, and each defence comes at a cost, additional code, degraded performance and extended development times. The choice of defence will ultimately be a compromise, considering these factors; the likelihood of the specific error being ineducable and the impact on the application if the error goes unchecked.

Therefore, defensive coding can be seen as a self-performed software sanity check to detect erroneous behaviour and respond appropriately to prevent errors from propagating.

5.1.2 Fault Model

A Fault Model describes the likely error behaviour of a device under attack. Different devices will have different models, and those models will vary per deployment, depending on the opportunities for attack and the commercial value of any assets those devices protect. The range of errors a μC may exhibit depends on the physical properties of the device. A remote device will not be physically attacked, for example, but could be probed with malformed, badly sequenced or badly timed command requests. On the other hand, electrical consumer goods can be subjected to the most intrusive physical attacks, unpoliced by their creators.

5.1.3 Testing Defences

Historically, programmers have defined fault models based on knowledge of the target device, experience, and to a large degree, intuition. A set of appropriate defences is then prescribed and implemented throughout the application during development. The cost of the defences is easily measurable, but its appropriateness is difficult to test because the defensive response can only be triggered by an error. If this error behaviour is simulated, the simulation's definition will most likely be influenced by the same factors and logic that prescribed the defences. Where multiple defence mechanisms exist for a particular error category, the relative efficacy of each of them

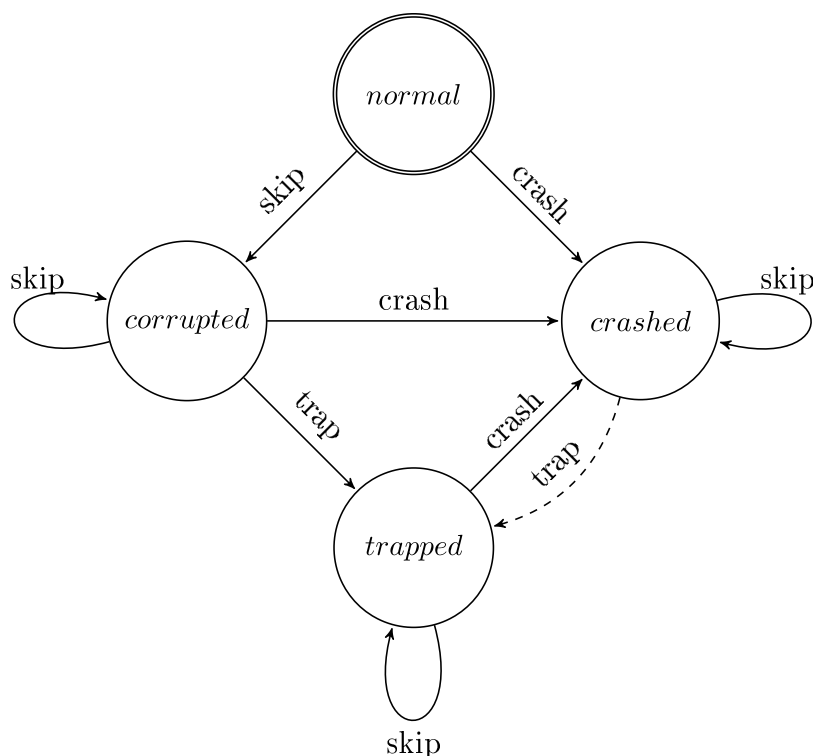


Figure 5-1: Execution States

may also be difficult to observe in simulation*. Anything but the most detailed hardware simulation will be testing the features for which defences have been defined.

The new laser workstation permitted us to physically test various defensive structures in the device for which the program is being developed. To do this, we defined a minimalistic fault model. We considered an executing program to be in one of four states as shown in Figure 5-1.

1. ***normal***: Execution as expected. Unaffected by error injection.
2. ***corrupted***: Execution continues within our program but some instructions may have been skipped and some data values may be incorrect.
3. ***trapped***: The executing code has recognised it was in the *corrupted* state and has deliberately entered a trap. Here the program would be expected to erase, or otherwise protect, valuable assets before freezing execution.

*"...there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns — the ones we don't know we don't know." —Donald Rumsfeld, United States Secretary of Defense, 12/02/2002 :

4. ***crashed***: The executing code is out of our program's control.

Transitions between the states occur as a consequence of one of three events.

1. ***crash***: may occur as a consequence of a single catastrophic error such as a skipped RET or as a result of continued execution with corrupt data for example RET from a function when the stack or frame pointers were previously corrupted.
2. ***skip***: occurs when an instruction fails to execute correctly but the program continues. There is a high likelihood that some aspect of the program's state will be corrupt. It is also possible that a fetched data value had been corrupted as noted by Proy [139]. Here we treat errors such as failure to execute an ADD or, adding the wrong value, as equivalent outcomes.
3. ***trap***: the executing program detects its own *corrupted* state. This transition is driven by the defensive code, and the efficacy and efficiency of this detection process is the primary driver behind this study.

Ideally programs will terminate in either the *normal* or the *trapped* states. The risks associated with termination in the *corrupted* state are well-documented [32, 37]. Attempts to cause a crash during data delivery is a long-established technique aimed at obtaining snapshots of a DUT's memory contents. As such, termination in either *corrupted* or *crashed* states is undesirable.

5.2 Defences

Our catalogue of defences was compiled from examples published in studies such as Theißing [173], Barenghi [23], Malkin [109]; manufacturer prescribed defences from Infineon [87], Samsung [152] and InsideSecure [89]; Payment scheme security recommendations from MasterCard [111], Visa [182]; and a back catalogue of secure applications for EMV, e-Passport and JavaCard operating systems developed by Digital Locksmiths Ltd. over many years that have all been subjected to independent security review. We identified 14 basic defences from these sources, each representing a distinct defence strategy.

The exhaustive testing of all pulse patterns is prohibitively time-consuming. Thus, some compromises needed to be made. As noted in Section 4.2, for multiple-pulse attacks, the test time rises geometrically with the number of execution cycles to be examined. Because of this, the test code has been stripped down to the bare minimum. This pruning exercise reduced the predicted test execution time from many months to a few days for some tests.

It has been observed that defensive code is frequently employed to protect small, focused operations such as double testing within a comparison. Thus these short code samples remain representative of real-world defences. It is also worth noting that many of the defences become more effective when used in bigger, more realistic modes. For example, a checksum over a small data block hardly differs from a duplicate data value. In contrast, when used over a larger block, it is likely to be vulnerable to a miscalculation step. An attack is then signalled even when the data it protects is uncorrupted.

Careful attention was paid to code output from the compiler. It was noted that, even with optimisations disabled, code was frequently in-lined and repeated, apparently redundant code was often omitted. These compiler-generated optimisations can totally remove defences.

The code was also structured by placing `SecretOp` code after the defended code that selectively accessed it. This arrangement was intended to expose the vulnerability of skipping the final RET operation and falling through into the protected code. It also engineered the scenario required for testing out-of-sequence execution detection.

The complete set of defences examined is described below. Each code fragment was invoked from a test routine that provided a synchronisation signal to the laser timer. This signal ensured the test timing was accurate and consistent for all execution runs. The same routine disabled the laser timer as soon as the test code returned. Disabling the timer reduced the risk of inducing errors in the result delivery code if the sample returned sooner than expected.

5.2.1 Unprotected

```
1   // Unprotected
2
3   var Flag = FALSE
4
5   func test()
6
7       ...
8       if (Flag == TRUE)
9           return SecretOp()
10      else
11          return EXPECTED
12  endfunc
13
14  func SecretOp()
15
16      ...
17      return SECRET
18  endfunc
```

Figure 5-2: Unprotected

This sample acts as the reference behaviour for the other test samples. The basic logic is that a secret value is returned if a flag variable in RAM has a specific value. Under normal circumstances, the secret value should not be returned. We would expect the secret value to be erroneously returned if either the pre-call test of `Flag` was corrupted or if the function return failed and execution was to 'fall through' into the `SecretOp` code.

5.2.2 Double Test

```
1   // Double Test
2
3   var Flag = FALSE
4
5       ...
6       if (Flag)
7           if (Flag)
8               return SecretOp()
9           else
10              TRAPPED
11      else
12          return EXPECTED
13
14
15  func SecretOp ()
16      ...
17      return SECRET
18  endfunc
```

Figure 5-3: Double Test

Here the test is repeated. This defensive construct is frequently used in the EMV and JavaCard sample code that we reviewed. The technique is recommended in [111] with caveats. The rationale is that if a test is skipped, the repeat should detect the inconsistency and invoke the trap mechanism. Under error-free conditions, the expectation is that the program should return the EXPECTED value and that returning SECRET would indicate an error had been induced, and it had been undetected by the defence. The cost of this defence is trivial; it is one extra fetch and one extra conditional branch operation per decision point.

Despite its apparent simplicity, this defence mechanism test requires meticulous coding. In 'C', for example, using the manufacturer supplied GNU Compiler Collection (GCC) toolchain, the second test is recognised as redundant and eliminated. This code elimination can only be identified by closely examining the compiler's output. Reducing the optimisation level of the compiler retains the prescribed double test; however, the `Flag` variable is copied to register, and that register's contents are then compared twice. The solution lies in marking the variable as `volatile`, ensuring the variable is re-fetched from memory for the second test.

5.2.3 Retest at Destination

```
1   //  Retest at Destination
2
3   var Flag = FALSE
4
5       ...
6       if (Flag)
7           return SecretOp()
8       else
9           return EXPECTED
10
11
12  func SecretOp()
13      if (Flag)
14          ...
15          return SECRET
16      else
17          TRAPPED
18  endfunc
```

Figure 5-4: Retest at Destination

This is a slightly more sophisticated variant of *Double Test*. Here the second test is repeated within the called function. As above, it tests a property twice. Functions

that involve different numbers of parameters will incur differing timing overheads between the first and second test of the state variable `Flag`. When this technique is used to protect multiple functions, it will add a timing variability not seen in *Double Test*. This variation also has the advantage of being able to detect out-of-order invocation of the `SecretOp` code. The execution time cost of this defence is comparable to *Double Test* and, depending on how many times the function is invoked, should lead to a smaller memory footprint.

5.2.4 Inverse Test

```
1   // Inverse Test
2
3   var Flag = FALSE
4
5       ...
6       if (Flag)
7           if (!Flag)
8               TRAPPED
9           else
10              return SecretOp()
11      else
12          return EXPECTED
13
14
15  func SecretOp ()
16      ...
17      return SECRET
18  endfunc
```

Figure 5-5: Inverse Test

The test is repeated in its negative form. Evaluators have recommended this as an improvement to the *Double Test* mechanism. The intention is to ensure that one branch is taken and another not taken to reach the protected code. Thus attacks that force conditional branches to either be taken or fall-through should be resisted when the valid path requires an example of each mode of behaviour. In practice, the resulting assembler output does not reflect the 'C' source code's intention, and we had to handcraft assembler code for this test. The GCC compiler generated code identical to *Double Test*. This makes the strategy impractical for large projects with this toolchain unless the compiler's behaviour can be modified. The cost of this defence is equivalent to *Double Test*.

5.2.5 Double Data

```
1  // Double Data
2
3  var FlagA = FALSE
4  var FlagB = FALSE
5
6      ...
7      if (!FlagA && !FlagB)
8          return EXPECTED
9      elseif (FlagA && FlagB)
10         return SecretOp()
11     else
12         TRAPPED
```

Figure 5-6: Double Data

This mechanism duplicates critical data variables in memory. The simple rationale behind this mechanism is that if a variable becomes corrupted in memory or while being fetched, it is unlikely that its shadow copy will be similarly corrupt. A variable's value is only trusted when both copies match, and if the two instances differ, there must be a problem. The runtime cost of this defence is equivalent to *Double Test* thus, it has minimal impact on performance. The 100% duplication of data, however, is likely to prove impractical in resource-constrained environments typical of μC deployments.

5.2.6 Data Inverse

```
1  // Inverse data
2
3  var FlagA   =  FALSE
4  var InvFlag = ~FALSE
5
6      ...
7      if (Flag != ~InvFlag)
8          TRAPPED
9      elseif (Flag)
10         return SecretOp()
11     else
12         return EXPECTED
```

Figure 5-7: Data Inverse

This is another widely used defence and a theoretical improvement on the *Double Data* mechanism. Here the shadow copy is the logical inverse of the primary data. The state variables are only trusted when the two copies are complementary. The

presumption is that if both copies of a variable can be corrupted in memory or during a fetch, it is less likely that the two corrupted instances will be mutually complementary. The cost of this defence is the same as that for the *Double Data* defence. It is expensive in terms of storage if widely used but comparable to the preceding defences in terms of code volume.

5.2.7 Checksum

```
1   // Checksum over data
2
3   var FirstVar
4       ...
5   var Flag = FALSE
6       ...
7   var LastVar
8   var CheckSum
9
10      ...
11      if (CalculateCrc(FirstVar ... LastVar) != CheckSum)
12          TRAPPED
13
14      ...
15      if (Flag)
16          return SecretOp()
17      else
18          return EXPECTED
```

Figure 5-8: Checksum Data

The 100% redundancy of the *Double Data*, and *Data Inverse* defences can be avoided by maintaining a checksum over a set of variables at the expense of more computation. Here a checksum is generated over a set of variables. This checksum is then verified before the variables are used. The checksum also needs to be recalculated and set whenever the variables are updated. The runtime cost of this defence is very high as verifying and calculating checksums over blocks of data is computationally expensive. However, it has a low demand for resources when used to protect large data blocks. This defence has been seen defending large blocks of critical data as a one-time operation before a complex algorithm proceeds. For example, to verify cryptographic keys or to verify the whole machine state between Application Protocol Data Unit (APDU) commands in smart-card applications.

5.2.8 Redundant Representation

```
1  // Redundant Representation
2
3  #define SECURE_TRUE  0xA5
4  #define SECURE_FALSE 0x50
5
6  var Flag = SEURE_FALSE
7
8      ...
9      if (Flag == SECURE_TRUE)
10         return SecretOp()
11     elseif (Flag == SECURE_FALSE)
12         return EXPECTED
13     else
14         TRAPPED
```

Figure 5-9: Redundant Representation

This technique aims to detect corrupted data by inserting redundant data bits (sometimes referred to as sentinels) within a value's representation. If the sentential bits do not match the expected pattern, then the value as a whole must have been corrupted. The technique can encode multiple flags into a single word but is most frequently deployed to represent a single flag value where the sentinels and value can be tested in a single operation. The cost of this defence is very low. It requires very little storage as sentinels can be encoded within the redundant bits of a variable's storage word. Similarly, the testing of sentinel values can be performed in parallel with the associated data, or in the worst case, after logical masking and comparison operations.

5.2.9 Repeat Calculation

```
1  // Repeated Calculation
2
3  var nTmp1 = SecretOp()
4  var nTmp2 = SecretOp()
5
6      ...
7      nTmp1 = Calculation()
8      nTmp2 = Calculation()
9
10     if (nTmp1 != nTmp2)
11         TRAPPED
12     else
13         return nTmp2;
```

Figure 5-10: Repeat Calculation

Errors in computation can be detected by performing an operation twice and confirming that both calculations yield the same result. The technique is computationally inefficient but may be appropriate when invoking hardware-assisted calculations using peripherals such as co-processors. Repetition has its own drawbacks, and *Inverse Calculation* may be more appropriate.

5.2.10 Modified Compensated

```
1  // Modified & Compensated
2
3     ...
4     Tmp1 = Calculation(Rnd1)
5     Tmp2 = Calculation(Rnd2)
6
7     Result = Clear(Tmp1, Rnd1)
8     if (Result != Clear(Tmp2, Rnd2))
9         TRAPPED
10    else
11        return Result
```

Figure 5-11: Modified Compensated

In this technique, an input parameter affects the result of a calculation in such a way that the caller can easily compensate for it. This enables the caller to invoke a function multiple times, yielding different answers while still being able to confirm the accuracy of the results. If the function is entered accidentally during out-of-order processing, then the returned value is likely to be modified by an unknown input. This provides an additional level of defence beyond the ability to check the accuracy of the calculation. The computational cost of this defence depends on the complexity of removing the input's bias from the result.

5.2.11 Alternative Algorithm

This technique aims to overcome the primary weakness inherent in *Repeat Calculation*. Namely, the power profile of repeat calculations is likely to be similar and therefore recognisable, thus enabling an attacker to synchronise attacks on the same moments in both invocations of a function. Using different algorithms makes it less likely that an attacker could influence both to yield matching erroneous results. This is a very

```
1   // Alternative Algorithm
2
3       ...
4       Tmp1 = Method1()
5       Tmp2 = Method2()
6
7       if (Tmp1 != Tmp2)
8           TRAPPED
9       else
10          return Tmp1
```

Figure 5-12: Alternative Algorithm

costly defence in terms of both code volume and processing time. Performing two calculations and comparing their results must take more than double the time of a single execution of the optimal algorithm.

5.2.12 Inverse Calculation

```
1   // Inverse Calculation
2
3       ...
4       Tmp1 = Method(Input)
5       Tmp2 = InvMethod(Tmp1)
6
7       if (Input != Tmp2)
8           TRAPPED
9       else
10          return Tmp1
```

Figure 5-13: Inverse Calculation

For some algorithms, the inverse calculation can be significantly quicker than the normal calculation. In this situation, it is possible to confirm computational accuracy by ensuring the input data can be recovered and verified from the deliverable result. The confirmation step also avoids the repetition of the primary computation. A surprising result is that the cost of this defence is not always as high as the *Alternative Algorithm* defence. For example, with RSA signature generation, verifying the signature using the public key is significantly faster than repeating the signing calculation. Given the high cost of failure, see Boneh [37], this defence is frequently deployed.

5.2.13 Jump Id

```
1   // Jump ID
2
3   Var CallId
4
5   func ProtectedFn()
6        if (CallId != CALL_F)
7            TRAPPED
8        else
9            ...
10           CallId = RET_F
11           return Result
12  endfunc
13
14       ...
15       CallId = CALL_F;
16       Val = ProtectedFn()
17       if (CallId != RET_F)
18           TRAPPED
19       else
20          return Val
```

Figure 5-14: Jump Id

This technique aims to detect out of order execution. Code for function entry and exit is augmented with additional parameters to demonstrate the caller's intent to invoke the function. If the execution path accidentally enters the function, for example, by skipping a RET and falling through to adjacent code, then the executing function can recognise this is not deliberate and enter the *trapped* state. Similar behaviour at the function exit enables the caller to confirm the return occurred from the correct function. This defence is relatively expensive as each defended function's invocation, entry, exit & return state must be instrumented with various data set, get and comparison operations.

5.2.14 Waymark - Late Test

Whenever execution passes a particular point, a waymark variable is updated. The waymark can later be examined to confirm that all the critical points of the proceeding execution path were executed. This technique is well suited for code containing loops and function calls where different fields within the waymark can be manipulated independently and where the final value is constant and predictable at compile time. The overhead is minimal, and the test is performed at the end of the processing.

```
1  // Waymark - Late Test
2
3     WayMark = IV
4     ...
5     WayMark += M1
6     ...
7     WayMark += Mx
8     ...
9     if (WayMark != IV + M1 + ... Mx)
10        TRAPPED
11    else
12        return nRetVal
```

Figure 5-15: Waymark, Late Test

5.2.15 Waymark - On the Fly

```
1  // Waymark - On the fly
2
3  var nNextWayMark
4
5  func Waymark(n)
6    if (n != nNextWayMark)
7        TRAPPED
8    else
9      nNextWayMark++
10 endfunc
11
12     ...
13     Waymark(10)
14     ...
15     Waymark(11)
16     ...
17     Waymark(12)
18     return Result
```

Figure 5-16: Waymark, On the Fly

An alternative waymark mechanism enables the early detection of unexpected or skipped code. In this variation of the defence, each time a waymark is updated, its current expected state is simultaneously verified. Frequent inline tests throughout the whole code body make this mechanism able to detect out-of-order processing, and it can be used to switch a *crashed* program to a *trapped* state. It is marginally slower than the *Waymark Late Test* variant but has the advantage of noticing the effects of skipped code at the earliest possible opportunity.

5.3 Results and Analysis

Each defensive technique tested addresses a particular side effect of fault injection. These roles are shown in Table 5.1. Most defences address a single issue, although some have a minor overlap of roles. For example, the Retest in Target technique could detect accidental invocation of the target function under some circumstances.

- **Test & Comparison** errors arise as a result of faulty comparisons or failure to correctly execute a conditional branch operation.
- **Data & Arithmetic** errors arise as a consequence of many fault effects. In-situ memory editing, faults during fetch and store operations, or faulty arithmetic operations, all lead to a corrupt result from a calculation.
- **Flow-Control** errors arise when program code is executed inappropriately or when code that should be executed is not executed at all.

The effects of all of these errors can be achieved as the result of skipped instructions, where the skipped instruction fails to either read the data correctly or fails to operate on it correctly. For example, skipping a LOAD instruction has a similar effect as correctly loading a previously corrupted value, and skipping an ADD operation is equivalent to adding faulty data. This property justifies our focus on the *sweet-spot* where instruction skipping is most readily demonstrated.

Each test was designed to return different values depending on the execution state at the end of the run. Three coded return values identified the different termination states.

i. An expected value, returned when the program apparently executes faultlessly.

ii. A recognisable alternative value that is another legally returnable result that would be delivered if state variables protecting it contain appropriate values. Such behaviour is typical of applications that perform protected actions if, and only if, earlier actions have been performed, such as password verification.

Table 5.1: Test Roles

Defence	Test & Comparison	Data & Arithmetic	Flow-Control
Double Test	✓	.	.
Retest in Target	✓	.	✓
Inverse Test	✓	.	.
Double Data	.	✓	.
Data Inverse	.	✓	.
Checksum	.	✓	.
Redundant Representation	✓	✓	.
Repeat Calculation	.	✓	.
Modified Compensated	.	✓	✓
Alternative algorithm	.	✓	.
Inverse Calculation	.	✓	.
Jump Id	.	.	✓
Waymark - Late Test	.	.	✓
Waymark - On the Fly	.	.	✓

iii. An indicator to signal that a trap had been reached. Traps are entered under program control when the application detects an anomaly in its own state.

These three values corresponded to the *normal*, *corrupted* and *trapped* states of our execution state model, Figure 5-1. Unexpected return values were also treated as examples of *corrupted* behaviour. Failure to return or an unexpected volume of returned data were categorised as *crashed* samples.

The ideal outcome is for a test program to terminate in the *normal* or *trapped* state. The *trapped* response indicates that the defence worked and prevented the program from continuing. In contrast, the *normal* response suggests that the program did not suffer any faults or that the faults had no bearing on the outcome. Termination in the *corrupted* state is the least desirable outcome. It indicates that the program is running, but it is potentially making flawed decisions. In some of the experiments, it is equivalent to performing protected operations without performing the pre-requisite authorisation test. In others, it indicates that code execution may have

been skipped or repeated. *crashed* programs that go mute are less of a concern but crashes sometimes return bursts of data, and this is an undesirable and potentially dangerous outcome.

The laser was focused on a previously identified *sweet-spot* that reliably caused the μC to misread memory fetches. This gave us a high probability that each laser pulse would cause some form of error, either an instruction skip or a faulty operand fetch.

We also chose to attack each code fragment with all combinations of up to four laser pulses. Three injected errors would be a reasonable expectation for an attack, one to induce an error and two more to hopefully bypass a defensive step intended to anticipate and trap the initial error. A fourth pulse was added for good measure in case the above assumption was flawed. A fifth pulse would make the execution time of the experiments prohibitively long and could not be considered[†].

Initially, each code fragment was executed to determine the execution time of the algorithm under test. This knowledge was then used to program the laser pulse injector to subject each fragment to all possible time combinations of $1 \ldots 4$ laser pulses within this time window. The actual number of result samples obtained varied depending on the execution time of the code fragment (See Equation 4.1 and Table 4.1). Each experiment was then repeated 40 times[‡].

5.3.1 Results

The aggregated results from all of the experiments on the different test samples and their corresponding terminating states are presented below in Table 5.2.

The **Cycles** column is determined by the execution time of the code fragment being investigated. The **Samples** column indicates the number of unique 1,2,3,&4 pulse combinations tested per experiment. The **Termination State** columns show

[†]One test of Four pulses in a 102 cycle frame took seven weeks to execute. Five pulses would tie the equipment up for over a year.

[‡]The test program was reutilized from earlier experiments, and it repeated each experiment four times. The laser and microscope stage control program then executed the test program ten times to collect sufficient samples to demonstrate error repeatability while keeping the overall experimental time manageable. This resulted in 40 sample results per test site.

Table 5.2: Test Samples

	(per run)		Termination State (after 40 runs)			
Defence	**Cycles**	**Samples**	**Normal**	**Trapped**	**Corrupt**	**Crashed**
Undefended	11	561	3766	0	14209	4465
Double Test	12	793	6296	4524	14519	6381
Retest in Target	22	9108	95423	23631	154962	90304
Inverse Test	11	561	10190	32	6024	6194
Double Data	18	4047	36086	52178	66947	6669
Data Inverse	15	1940	12122	24489	30724	10265
Checksum	33	46937	842127	302284	531388	201681
Redundant Representation	12	793	10887	126	8774	11933
Repeat Calculation	32	41448	329672	700314	345769	282165
Modified Compensated	25	15275	110048	24	102600	398328
Alternative algorithm	31	36456	316146	558530	265387	318177
Inverse Calculation	46	179446	1024362	1530077	822062	3801339
Jump Id	30	31930	224302	594678	304951	153269
Waymark - Late Test	27	20853	191771	403210	163354	75785
Waymark - On the Fly	35	59535	323660	1403506	549331	104903

the total number of experiments terminating in each of the four states. Because the execution time, and hence the number of experiments, varies per sample, this presentation is relatively difficult to interpret.

The same data, normalized, is re-presented in Figure 5-17. Each termination state is shown as a fraction of the number of experiments performed on the code sample. This visual presentation shows the likelihood of an attack pulse pattern terminating in each state, making it possible to cross-compare the efficacy of different defences. Desirable termination states are *Normal* & *Trapped*. *Crashed* is undesirable but difficult to exploit, while *Corrupt* is potentially dangerous.

None of the defences is infallible, but this is unsurprising as we have already demonstrated the selective execution of arbitrarily timed instructions.

In this test environment, it is possible to determine the particular instructions being executed at the time of a laser strike. From this, it is possible to infer the

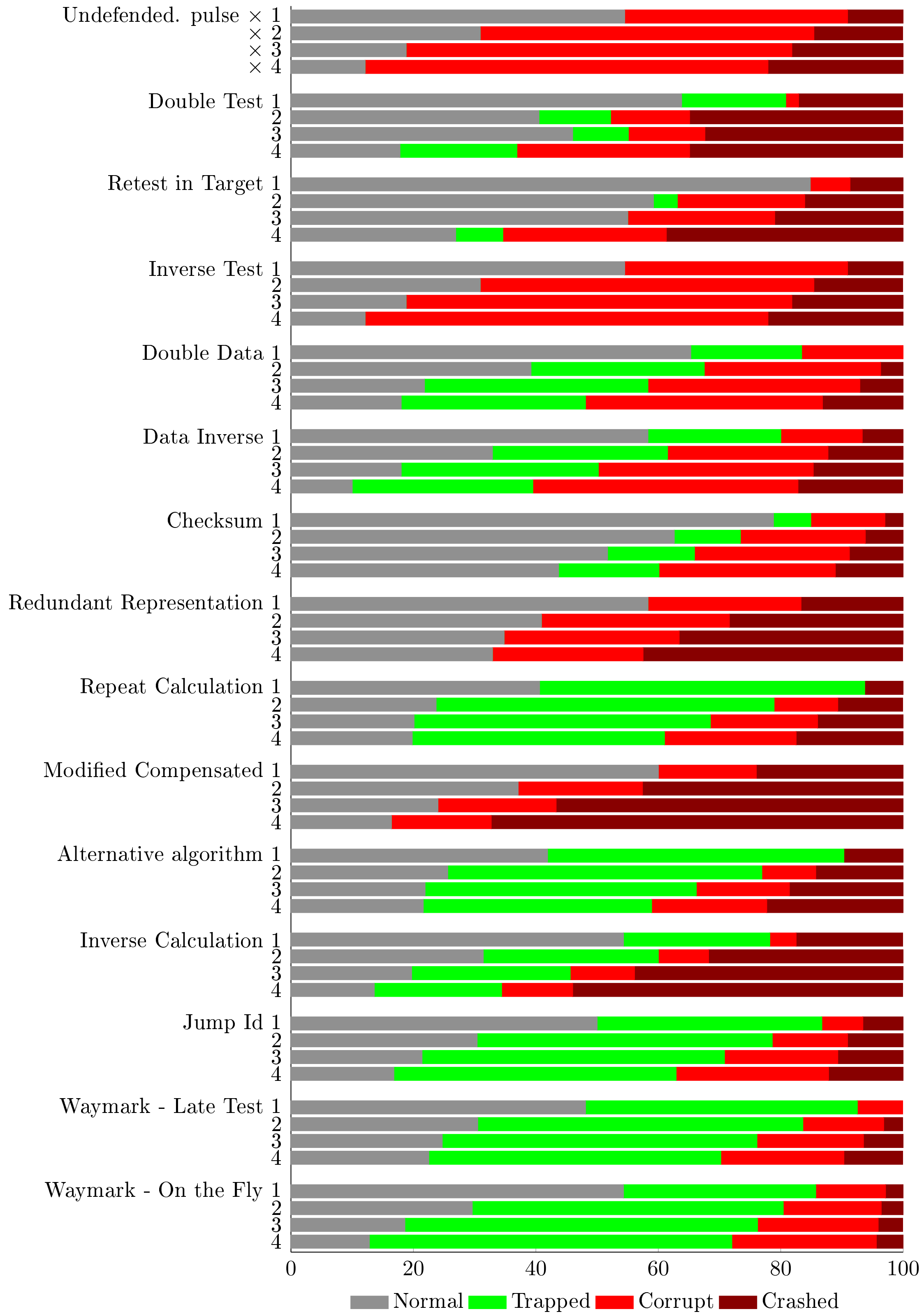


Figure 5-17: Termination States

execution path within a failure. This technique was used to determine probable failure paths and explain why similar defences show markedly different outcomes.

5.3.2 Normal Termination

The large number of *Normal* terminations initially appears to be surprising, given that the chosen target site was reliably susceptible to laser-induced errors. Examination of the instruction sequences in the compiled code and cross-checking this with the pulse times shows that these *Normal* terminations occur when the pulses coincide with instructions that take multiple clock cycles to execute. Here the vulnerable pre-fetch of the next instruction is not performed on every cycle. Similarly, on conditional branch operations, the potentially erroneous pre-fetched fall-through option may be discarded in favour of the calculated branch-taken address.

5.3.3 Trapping

The Inverse, Redundant Representation and Modified Compensated code samples rarely terminated in *Trapped* state. These are examples of where the programmer's intention, as prescribed in the source code, is not realised in the compiled output. This is illustrated when comparing the source code for Double Test (Figure 5-3) and Inverse Test (Figure 5-7). They are both very similar and logically identical, but the executable code is subtly different. Both programs have a standard test followed by a defensive retest.

The assembler code for the two tests is shown in Figures 5-18 and 5-19.

In Figure 5-18 failure to execute lines 10 or 11 has no effect as this is the repeat of the initial test, while failure to execute line 12 will be followed by a fall through into the Trap. This code fails safe or ends in the Trap.

In Figure 5-19, skipping line 12 results in the Secret code being executed, as does corruption of `r0`. This code defaults to calling the protected operation. Reaching the trap code involves misreading `Flag` variable as value zero on line 10, or the call to `SecretOp()` and the following RET, lines 15 & 16, both skipped.

```
1   //   if (Flag)                 // Check
2            ld     r0, Flag       // Fetch Flag from memory
3            or     r0, r0         // update Z flag
4            brne   L1             // Z-flag clear (true)
5
6   //   else EXPECTED
7            rjmp   Expected       // Z-flag Clear so 'else'
8
9   //   if (Flag)                 // Check again
10  L1:      ld     r0, Flag       // Fetch Flag from memory
11           or     r0, r0         // update Z flag
12           brne   L2             // Z-flag is Consistent. OK
13
14  //   else TRAPPED
15           rjmp Trap             // Z-flag is inconsistent. NOK
16
17  //   return SecretOp()
18  L2:      rcall SecretOp        // Called only after double test
19           ret
```

Figure 5-18: Disassembled Double Test

```
1   //   if (Flag)                 // Check
2            ld     r0, Flag       // Fetch Flag from memory
3            or     r0, r0         // update Z flag
4            brne   L1             // Z-flag clear (true)
5
6   //   else EXPECTED
7            rjmp   Expected       // Z-flag Clear so 'else'
8
9   //   if (!Flag)                // Check again
10  L1:      ld     r0, Flag       // Fetch Flag from memory
11           or     r0, r0         // update Z flag
12           breq   L2             // Z-flag is inconsistent. NOK
13
14  //   return SecretOp()
15           rcall SecretOp        // Called only after double test
16           ret
17
18  //   else TRAPPED
19  L2:      rjmp Trap             //
```

Figure 5-19: Disassembled Inverse Test

Despite their similarities, one efficiently identifies injected faults, and the other is surprisingly ineffective. As we see in Figure 5-17 the Inverse Test has more similarities with Undefended code than it does with the Double Test code.

5.3.4 Crashing

There is a noteworthy but unsurprising trend to terminate in the *Crashed* state that increases with the number of injected faults. It is difficult to get out of this state in these test scenarios, and with more faults injected, it is more likely to be entered. An

exit from the *Crashed* state would require a further fault that returned execution to the controlled environment of the test code. Additionally, when entering the *Trapped* state, there is a window of opportunity for another fault to be injected, hijacking the trap and resulting in a crash. In some cases, the result reporting functionality is itself subjected to laser pulses, resulting in erroneous result delivery and consequently being interpreted as *Crashed*.

5.3.5 Out-of-Order Processing

A relatively easy to identify cause of corruption is out-of-order processing. Here, a code section is executed at an unexpected time; for example, by 'fall-through', where, after skipping a RET operation, execution will continue into the neighbouring function. In many cases, the next RET encountered returns execution to the original caller where execution resumes, unaware of the additional processing that had been performed. The reverse effect occurs when a function call is skipped. Here there is a failure to execute code, and the caller is unaware of it.

On other occasions, skipping conditional branches within comparisons leads to a similar effect where an unscheduled code section is executed. This is a specific example of out-of-order processing where execution falls through to conditional code regardless of the state of the condition.

The defences against out-of-order are JumpId and Waymarks, and the best of them is Waymark-on-the-fly. The key feature of Waymark-on-the-fly is the accumulating waymarker and the repeat tests. If either marking or testing is skipped in one instance, the effect can still be noted and trapped by a subsequent mark and test instance.

5.4 Application

The defences tested here individually address particular fault scenarios, and in reality, defences need to be applied in combinations to cover the range of different faults an application may encounter. It is expected that an efficient recipe for defensive coding can be identified by combining defences from each category. Therefore, the question

is, does a hybrid defence inherit the positive properties of its components, or does the compiler generate code that, like the Inverse Test, superficially looks good but is impotent in practice? This exercise demonstrates the practicality of using the new laser workstation as a development tool for refining and testing defences.

Based on earlier observations, we combined the most effective defences to create a testable hybrid defence. Figure 5-20 shows this hybrid combination of *Waymark (Test on the Fly)* for flow control protection, *Repeat Calculation* for arithmetic accuracy, and *Double Test* for conditionals.

```
1   // Hybrid Defence
2
3   var nNextWayMark
4
5   func Waymark(n)                // Waymark
6       if (n != nNextWayMark)     // Confirm caller's prediction
7           TRAPPED                //     Failed.
8       nNextWayMark++             // Set next predictable value
9   endfunc
10
11
12  func Calculation(WP)           // Arithmetic function
13      Waymark(WP)                // Confirm called as expected
14      ...                        //      Do the arithmetic
15      Waymark(WP+1)              // Show we got to the end
16      return EXPECTED            //
17  endfunc
18
19  func TestEQ(WP, V1, V2)        // Comparison with waymark
20      Waymark(WP)                //
21      return (V1 == V2)          //
22  endfunc
23
24    ...
25    A = Calculation(1)           // Calculation
26    B = Calculation(3)           // Repeat Calculation
27    if (TestEQ(5, A, B))         // Comparison
28        if (TestEQ(6, B, A))     // Comparison Doubled
29            Waymark(7)           // Proof of full path
30            return A
31    TRAPPED
```

Figure 5-20: Hybrid Defence

Table 5.3 shows the number of executions terminating in each of the four states. The same data is presented pictorially in Figure 5-21. While the example is contrived, it shows that individual defences can be combined to improve the level of defence within an application, as witnessed by the low incidence of *Corrupt* or *Crashed* results. The proportion of runs that end in the *Trapped* state also increases as the number of pulses increases, demonstrating that this defence improves when an attack intensifies.

Table 5.3: Hybrid Defence Test

102 Cycles		Termination State							
		Normal		Trapped		Corrupt		Crashed	
Pulses	Samples	#	%	#	%	#	%	#	%
1	408	184	45.1%	216	52.9%	0	0.0%	8	2.0%
2	20604	4289	20.8%	15955	77.4%	0	0.0%	360	1.7%
3	686800	67688	9.9%	610653	88.9%	28	0.0%	8431	1.2%
4	16998300	821628	4.8%	16032665	94.3%	2656	0.0%	141350	0.8%
total	17706112	893790	5.0%	16659489	94.1%	2684	0.0%	150149	0.8%

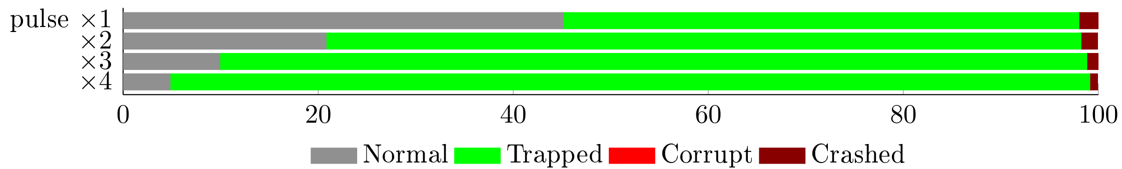


Figure 5-21: Hybrid Defence Termination States

5.5 Summary

Using physical device samples, as opposed to simulation, is a practical method of evaluating software defences. This approach exercises features that are unknowable to all but the most detailed simulations. Others, Brejon [41], Proy [139], . . . , have argued that physical attacks overlook issues relating to the instruction pipeline and the CPU's behaviour while skipping an instruction. They argue that ISA level modelling is the only way to understand the full effects of a faulted instruction. We have shown real physical effects can be observed, and un-modelable subtle side effects will be included in these observations. The results presented here and the fact that they were collected with relative ease demonstrate the counterargument.

It is unlikely that chip manufacturers will share sufficient detail to enable highly accurate modelling of their devices with developers. It is even less likely that they would do so with hackers. Thus detailed modelling will only be a practical option for the device manufacturers themselves or for laboratories with the resources to reverse engineer the silicon chip. Most importantly, from a practical point of view, there is no overriding need to understand a fault so long as it can be recognised and neutralised.

The prevalence of *crashed* data collected from the experiments also highlights the need to freeze the CPU immediately upon detection of a fault. Many crashes were identifiable as a consequence of the trap management being faulted. The calling and processing of a `Trap()` function can itself be faulted and potentially disabled. Ideally, a HALT instruction or reset interrupt should be invoked. This was not done in these experiments because of the requirement to separately identify *crashed* and *trapped* termination states.

When untrapped errors are induced, knowledge of the pulse times and executing code can be used to show what went wrong and hopefully guide a programmer to an improved solution. Alternatively, using the same techniques, a hacker can develop a library of template pulse patterns that defeat known defensive code structures. Applying a multi-pulse template as a Creeping Barrage over a long stretch of unknown executing code is a practical attack mechanism, as described in Section 4.3.

All of the defences ultimately rely on comparison operations to decide whether to proceed or not. This *Trust-or-Trap* decision process is unrelated to the data representation or algorithm used, and disrupting it negates the defence. The surprise result from the *Inverse Test* sample demonstrates this. When a conditional branch is skipped, the fall-through option becomes the default. Testing for a problem and then optionally branching to a trap function is inherently weak. This branch weakness is, in many cases, outside the programmer's control as the compiler ultimately controls the ordering of, and flow between, different blocks of code. The strongest defence comes when a positive test leads to a conditional jump to the appropriate code, with all fall through behaviour resulting in a trap. This arrangement cannot easily be described in the high-level source code. Instead, a compiler will typically treat a binary decision as a conditional jump and a deliberate fall-through. A compiler's behaviour can also change depending on the chosen optimisation level and may change in future versions. A chosen defence for a device will need to be reevaluated whenever such changes occur.

As a test environment, the most significant limitation we encountered was the test repeat rate. When considering large numbers of pulses or large code fragments,

exhaustively checking all pulse combinations can take a considerable time. The *Hybrid Test* program took 102 instruction cycles to execute, resulting in 17,706,112 samples, and that took 7 weeks to collect at 4 tests per second. Despite this frustratingly slow repeat rate, it still compares favourably against simulation. On a powerful desktop PC, the logic level simulation of our test rig's pulse timer mechanism took 15 seconds to model 120 clock cycles (Figure B-10). Simulation of a μC will take longer due to the vastly increased complexity. This logic level simulation cannot possibly model the interaction of multiple simultaneous errors induced in neighbouring transistors because a VHSIC Hardware Description Language (VHDL) description provides no information about the physical layout of the target's subcomponents. Thus a more detailed, and hence slower, simulation would be required. For all practical purposes, physical testing will be quicker than simulation, and as the detail of a simulation increases, so too will the advantage of physical testing.

These results, also presented here (and Kelly [95]), are directly applicable to this specific DUT and the particular version of the GCC toolchain used here. However, the techniques are generically applicable and easily repeatable. This prediction is supported by contemporary observations that the instruction skipping effect has also been noted as the most readily exploitable fault effect on a different μC architecture, Colombier [52]. This also reinforces the idea that the techniques should be transferable, even if the specific defences need modification. So in this respect, the nature, if not the implementation, of effective defence combinations will most likely be similar.

μCs that use clock jittering to disrupt SPA and DPA attacks would require many more samples to be gathered for each candidate pulse pattern. Correlation between the pulse timing and the currently executing instruction will be difficult to perform in this scenario. However, the quantifiable outcomes of *trapped*, *corrupt*, ..., will still provide a measure of the efficacy of a defence for developers or pulse timing templates for attackers.

As a tool for studying error injection attacks, this low-cost laser workstation's capabilities far exceed those of the expensive YAG laser cutter it replaces.

Chapter 6

Automating Defence Generation

Contents

6.1 Background **197**

6.2 Defensive C Compiler **201**

6.2.1 A Defensive Object-code Generator. 202

6.3 Defending Execution Path **203**

6.3.1 Call and Return 207

6.3.2 Branching 216

6.4 Code Efficiency **224**

6.4.1 Large Application 225

6.4.2 Call Intensive Code 226

6.4.3 Branch Intensive Code 227

6.4.4 Code Size 227

6.4.5 Code Performance 227

6.5 Summary **230**

6.5.1 Room for Improvement 231

Not all defences can be described via the syntax of high-level programming languages, and hand-crafted assembly language programming is both time-consuming and error-prone. By integrating defences into a compiler's code generator, we can ensure defences are universally applied and that the compiler's optimiser does not eradicate important but seemingly redundant code structures. The resulting compiler permits experimentation with defences and generates code of comparable efficiency to commercial compilers.

Writing defensive code is both time-consuming and error-prone. As noted in Section 5.3.3, subtly different but logically identical, defences can exhibit markedly different efficacy. These behavioural differences result from the compiler's translation and optimisation and are difficult to influence from the source code.

```
1   extern u8 Function1(void);
2   extern u8 Function2(void);
3
4   u8 bVal;
5
6   u16 Foo(void) {
7       u8 b;
8
9       if (bVal) {
10          if (!bVal) {
11              trap();
12          }
13          else {
14              b = Function1();
15          }
16      }
17      else {
18          if (bVal) {
19              trap();
20          }
21          else {
22              b = Function2();
23          }
24      }
25
26      return (u16)b;
27  }
```

Figure 6-1: Simple Double Test Source

Take, for example, the simple code fragment shown here in Figure 6-1. The functional intention of this code is to call either `Function1()` or `Function2()` depending on the state of a variable, `bVal`. `bVal` is tested twice and if the tests are contradictory a `trap()` function is called. It is also hoped that the ordering of the code statements means that if the test instructions were skipped, fall-through would default to the `trap()`.

```
1   Foo:        lds     R24,bVar
2               tst     R24
3               breq    L1
4               rcall   Function1
5               rjmp    L2:
6   L1:         rcall   Function2
7   L2:         ldi     r25,0
8               ret
```

Figure 6-2: Accidental Optimised Output

The programmer's failure to mark `bVal` as volatile and compiling with default

options results in the code shown in Figure 6-2.

The compiler has identified that second tests of `bVal` are redundant and has deduced that the calls to `trap()` are unreachable and can therefore be eliminated. This highly efficient code has had all of the intended defences removed by an over-helpful compiler and a simple oversight by the programmer.

```
1   // -O3
2   Foo:      lds     R24,bVar
3             tst     R24
4             breq    L2
5             lds     R24,bVar
6             cpse    R24,R1
7             rjmp    L4
8   L1:       rcall   trap
9             rjmp    L3
10  L2:       lds     R24,bVar
11            cpse    R24,R1
12            rjmp    L1
13            rcall   Function2
14  L3:       ldi     r25,0
15            ret
16  L4:       rcall   Function1
17            rjmp    L3
```

Figure 6-3: Optimised Output

After correcting the programmer's error by marking `bVal` as *volatile*, the compiler generates the output shown in Figure 6-3. Here the variable is tested twice as intended, but the compiler has re-ordered the code. Now a single fault injection to skip the function's return (line 15) will result in `Function1()` being called, and the routine will still return to its caller as intended. Code re-ordering is unpredictable and depends on the size of the code blocks and any duplication of code within them.

The code re-ordering can be overcome by reducing the optimisation level as shown in Figure 6-4. This code is closer to the programmer's original intention. Multiple errors must be induced before `Function1()` or `Function2()` can be invoked inappropriately. Unfortunately, lowering the optimisations to achieve this has also eliminated safe optimisation relating to the unnecessary use of register R28. The resulting code is larger and slower than it needs to be.

This section looks at automating code generation with two specific security-related goals. Firstly, making the compiled code easier to verify simplifies the review process and reduces development costs. To achieve this, we want human-readable output that can demonstrate to reviewers that effective defences are present despite their apparent

```
1   // -O1
2               push    R28
3               lds     R24,bVar
4               tst     R24
5               breq    L2
6               lds     R24,bVar
7               cpse    R24,R1
8               rjmp    L1
9               rcall   trap
10              rjmp    L4
11  L1:         rcall   Function1
12              mov     R28,R24
13              rjmp    L4
14  L2:         lds     R24,bVar
15              tst     R24
16              breq    L3
17              rcall   trap
18              rjmp    L4
19  L3:         rcall   Function2
20              mov     R28,R24
21  L4:         mov     R24,R28
22              pop     R28
23              ret
```

Figure 6-4: Unoptimised Output

absence in the source code files. Secondly, the automatic generation of defensive code simplifies implementation and reduces the potential for accidental errors, making this niche skill available to programmers with differing skills and backgrounds.

Finally, if the goals are achievable, will the resulting code be efficient enough for deployment? The key factors are the defence's effectiveness, execution speed, and generated code volume. Determination of efficacy requires small abstract test samples that can be exhaustively tested, while meaningful comparisons of speed and size require larger, more realistic applications. Fortunately, we had access to the source code for several commercially developed smartcard applications that enabled these comparisons to be made.

6.1 Background

Using a compiler to generate code that goes beyond the source code language's semantics to defend against undesirable behaviour is not new.

Automated defences against side-channel leakage, a related security threat, have been investigated. Pre-compilation code transformations to mask leakage relating to the execution-path are described by Molnar [115], and Moss [119] describes transfor-

mations of a compiler's intermediate arithmetic representations to mask data related leakage. They are interesting because they show that features and obscure behaviour that is difficult to implement with traditional development tools can be delegated to the compilers.

Execution error defences have also been investigated. For example, adding canaries into a stack frame provides a run-time mechanism to detect buffer overruns, Cowan [56]. Similar capabilities are built into Microsoft's Visual C and are seen in 'debug' builds of applications but absent from 'release' builds. This demonstrates the common mindset that assumes errors result from flawed programming logic and, once eliminated, the host platform can be trusted. In a similar vein, extensions to GCC have been described by Jones [91] that ensure the results of pointer arithmetic reference meaningful data. These are primarily defences against programming errors and errors triggered by malformed or unanticipated data.

In an extreme example, Reis [144] has demonstrated a compiler that generates two functionally identical versions of code, each using different register and memory allocation strategies. Both are then executed, and the results are compared to identify if errors have occurred. This approach is the software equivalent to Infineon's Integrity Guard [88] where duplication of hardware is used to detect faulty execution.

Of the defences investigated in Section 5 the most complicated to implement, but most effective defensively, protected against out of order execution. This is where a compiler can add the greatest value and has had recent interest from researchers. The interest was triggered by so-called *Stack Smashing* attacks, where a hacker would exploit buffer overruns to insert executable code into the stack frame. By corrupting return addresses within the stack frame, functions could be made to return into the injected code rather than the original caller. The simple defence of prohibiting execution of code within the stack can be defeated by an improvement on the attack, known as Return Oriented Programming (ROP). In ROP manipulation of return addresses on the stack is used to transfer execution to known code fragments. This technique is sometimes known as *return-into-library* or *return-to-libc* attack because an application's runtime library is often the best source of exploitable code fragments.

Out of order execution is addressed by Control Flow Integrity (CFI) as described by Abadi [1] and is effective against buffer overflow, *return-into-library* attacks and pointer subterfuge. As a defence, CFI attempts to verify that each control-flow transfer is required for the correct execution of a program. Called functions perform run-time checks to confirm they were deliberately invoked. A control flow graph is statically generated at compile time; this is combined with run-time verification invoked upon arrival at a destination to ensure the flow transition was expected. The technique recognises when functions are invoked from unexpected call-stack configurations and terminates execution. A recognised disadvantage here is the size of the control flow graph, which may contain many unnecessary edges. Construction of a smaller control flow graph, on-the-fly, has been proposed by Niu [126] but the technique is still too expensive for resource-constrained devices, Burow [49]. Microcontrollers and similar resource-constrained processors need a mechanism to enforce CFI while avoiding the overheads associated with storing and checking a control flow graph.

Other researchers have also identified flow-control verification as an important defensive technique and have offered solutions.

Source Code Level. Automatic additions and modifications to a program's source code are possible. For example, Lalande [103] simulates a program at source code statement-level using a fault model that assumes that glitches can cause random assignments to the program counter. The impacts of simulated glitches are then analysed to identify outcomes that may have security implications. Defensive code in the form of *Waymarks** is then automatically added to the source code to defend these vulnerable sections. This arbitrary-jump fault model has been demonstrated as being realistic by Gratchof [79], but in doing so, it was noted that it is difficult to control. Arbitrary jumps, however, need not necessarily jump between lines of source code and will frequently resume execution partway through code associated with a single source code statement. As such, the categorisation of glitch outcomes may be flawed. The efficiency of Lalande [103]'s technique of strategically placing *Waymarks*

*See Sections 5.2.14 & 5.2.15

at only the most critical locations may overlook many code vulnerabilities and fail to take into account the relative likelihoods of particular outcomes occurring.

Intermediate Representation†. After parsing source code, a typical compiler builds a representation of the program in a collection of data structures. This Intermediate Representation (IR) can then be analysed and manipulated to optimise it. Loop hardening by directly manipulating the IR was attempted by Proy [138]. After recognising the compiler's habit of aggressively removing redundant code, the defences were inserted later in the compilation process. However, at this stage of compilation, loops and their vulnerable control variables become more difficult to identify, and the technique fails to protect all loops. Additionally, it was noted that modern compilers, such as the LLVM [105] used in the study, perform multiple stages of optimisation on the IR and can still recognise and remove the additional redundant code.

Link Time. The final step in creating a program's binary output is linking, and some limited optimisations are possible during this step. Link time code hardening as been considered by deKeulenaer [57]. Recognisable vulnerable code sequences in the linkable code can be substituted for more robust alternatives. E.g. expanding conditional branch instructions to trap accidental fall through. The disadvantage of this technique is that all additional information from the source code relating to the object code has been lost. Defences, when blindly applied, may then have unintended consequences, particularly when memory-mapped special function registers are accessed or where code addresses are calculated at run-time.

The above examples echo the view expressed by deKeulenaer [57] that "programmers should not have to waste effort and time on changing their source code to introduce the redundancy". They also demonstrate that automatic generation of defensive code needs information only available in the source code. As a result, the process is highly suspectable to even the most basic optimisations. Proy [138] concludes that "We do not foresee that such a problem will find a general, compiler-independent

†Aho [4] "Introduction", & Bornat [38] "Phases and Passes", refer to Analysed-programs, Parse-trees or various stages of Intermediate Representation. Here, Intermediate Representation (IR) covers all of them generically

answer".

Defences within source code are suspectable to being eliminated by an over helpful optimiser. Moreover, the best defences are target specific but unreliable when applied at the link phase. We are left with the conclusion that target specific defences should be inserted after the IR optimisations have occurred but before the source code annotations are lost. i.e. in the so-called Translator or Object-code generator.

6.2 Defensive C Compiler

The stages of a typical compilation process are shown in Figure 6-5. A typical C compiler would perform all of the transformations, from source code (stage 1) to object code (stage 6), as a single operation from the user's point of view. Internally it will still follow the transformations shown here (often missing out stage 5) and directly generate object code as the *Code Generator's* output.

The Defensive C Compiler (DCC) is a test environment for the experimental *Code Generator*, but because the generator needs an input IR, it also needs to perform the roles of *Parser* and basic *Optimiser*. The parser uses the ANSI C language YACC grammar presented in Kernighan & Ritchie [43]. It constructs a tree structure of the source code program, and in doing so, it confirms the syntax of the input and identifies missing or undefined functions and variables. The tree structure is the basic-IR and can be traversed and manipulated without the need to anticipate syntactic errors. The *Optimiser* manipulates the basic-IR, e.g., evaluating expressions to remove constants or converting array and structure field access into pointers and displacements. The optimised-IR obeys the same syntax rules as, and is semantically identical to, the basic-IR. This optimisation aims to reduce the code volume by simplifying the IR's structure, ultimately speeding up calculations. No defences are added at this stage. Finally, the optimised-IR is presented to the *Code Generator* and converted into individual, target specific machine instructions. More details of DCC can be found in Appendix C.

The alternative approach would be to take an open-source compiler, such as

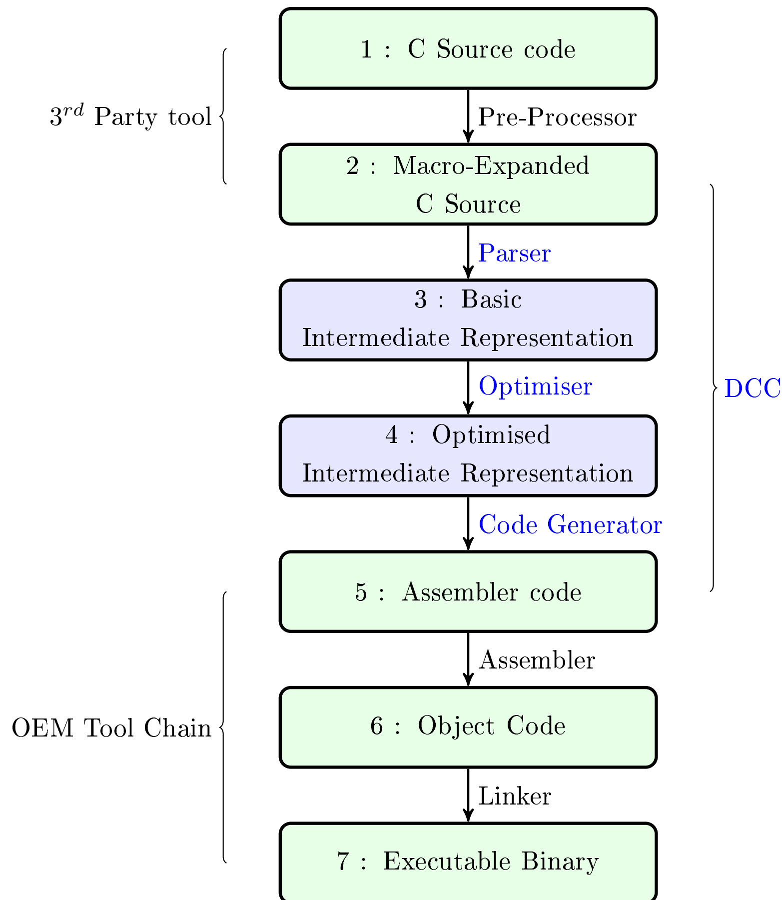


Figure 6-5: Compilation Stages

GCC [169] or LLVM [105], and adapt or replace an existing code generator. The absence of a public domain AVR variant of either compiler and the need to gain familiarity with a highly complex versatile IR before writing a generator led to the decision to implement a basic, C specific, IR. The code generator is the main point of interest for this study, so an imperfectly optimised-IR is acceptable.

6.2.1 A Defensive Object-code Generator.

An additional advantage of inserting defensive behaviour into the code generation step is that the output can be annotated to assist with quality assurance and security review. With access to the symbol tables, detailed knowledge of the origin of each code block, and the logic controlling execution flow between code blocks, it is possible

to indicate where the defences are and what form they take.

Once written, it is relatively simple to substitute an alternative target μC instruction set. The output is a human and machine-readable assembler code file that is generated via a series of `printf()` statements.

The assembly file output is then converted to object code by the original manufacturer's toolchain. DCC's output can therefore be freely mixed with other assembler code modules, and the resulting program can be loaded, executed and debugged via the industrial-strength OEM tools.

The best source code defences against arithmetic errors involve repeating a calculation via an alternative algorithm or reversing it to ensure a result is consistent with its input. This level of intelligence is beyond the capabilities of a simple compiler. The crude and less effective approach of simply repeating calculations is inefficient and has the potential to change the semantics of the source code[‡]. There is also the issue of volatile variables to consider. Here a repeat calculation will differ from the original; or, the compiler may deliberately avoid the recalculation leaving the programmer unaware that a defence has been omitted.

That is not to say that the problem is insoluble. Compiler generated arithmetic transformations by Patrick [134] and inter-instruction redundant operations by Moss [120] have been used to defend cryptographic functions. While these are specific use cases, they demonstrate the approach's feasibility and suggest it is worthy of further investigation.

However, this proof-of-concept study concentrates on the equally important task of CFI, defending the execution path.

6.3 Defending Execution Path

To achieve this, we needed a mechanism that permitted us to execute a code sample and then retrospectively determine the execution path taken. The mechanism also

[‡]E.g., post incremented variables - most C programmers will recognise this issue commonly associated with `max(a,b)` macro.

had to be tolerant of glitch attacks upon itself, as there was no practical mechanism to avoid glitching the μC while it was recording the execution path without first knowing that path and its time of execution.

The solution adopted is shown here in Figure 6-6. Three consecutive counter increments are performed within the body of each code block at strategic points. Limiting the number of glitches to two per run restricts the range of possible modifications to a given triple. Each time a triple increment is executed, it must leave a trace by updating at least one of the counters. By inspecting each triple at the end of the experiment, it is possible to infer whether code was not executed, executed once or multiply executed.

```
1   char baBeenThereDoneThat[N_TESTS*3];
2
3   #define TestTick(n) { ++baBeenThereDoneThat[n*3];   \\
4                         ++baBeenThereDoneThat[n*3+1]; \\
5                         ++baBeenThereDoneThat[n*3+2]; \\
6                       }
7
8
9   void foo(void) {                 // called fn
10      TestTick(1);                 //
11  }                                //
12
13
14  void bar(void) {                 // uncalled fn
15      TestTick(3);                 //
16  }                                //
17
18
19  void TestMain(void) {            // Entry point
20      TestTick(0);                 //
21      foo();                       //
22      TestTick(2);                 //
23  }
```

Figure 6-6: Test Structure

```
1   lds   r20, baBeenThereDoneThat+3    // reg8 <== Global baBeenThereDoneThat[3]
2   inc   r20                           // pre inc 8
3   sts   baBeenThereDoneThat+3, r20    // baBeenThereDoneThat[3] <== reg
4   lds   r20, baBeenThereDoneThat+4    // reg8 <== Global baBeenThereDoneThat[4]
5   inc   r20                           // pre inc 8
6   sts   baBeenThereDoneThat+4, r20    // baBeenThereDoneThat[4] <== reg
7   lds   r20, baBeenThereDoneThat+5    // reg8 <== Global baBeenThereDoneThat[5]
8   inc   r20                           // pre inc 8
9   sts   baBeenThereDoneThat+5, r20    // baBeenThereDoneThat[5] <== reg
```

Figure 6-7: Triple Update

Implementing the `TestTick()` functionality as a macro ensured it resulted in inline code, thus preventing the inclusion of additional control instructions (`Call` and

`Ret`) within the tested code sample. Additionally, using a constant value to identify each macro invocation ensured no run-time arithmetic was required when calculating an address to update. Corruption of run-time arithmetic would seriously confuse the results being gathered. The code generated by both compilers for updating a triple was inspected to confirm this predicted behaviour and is shown here in Figure 6-7.

Table 6.1: Triple Values

Category	**Interpretation**	**Pattern**	**Reasoning**
None	Not executed	000	Expected result if the code is not executed.
Single	Single pass	111	Expected result if all instructions execute correctly.
Multiple	Double pass	222	Executed twice and all instructions execute correctly.
Single	Single pass	112, 121, ?11	One error in either of fetch, increment or store
		011, 101, 110, 001, 010, 100	One error in either of fetch, increment or store, or, Two errors in Increment or Store operations
		012, 021, 102 120, ?01	One fetch error and one increment or store error
		123, ??1, ?21	Two fetch errors
Multiple	Double pass	122	One increment or store operation failed
		223, 232, ?22	One Fetch failed
Ambiguous		212, 221	Single execution with errors on two failed fetch operations, or, Double execution with one increment, or, store error.

At the end of each experiment, the pattern of values within each triple was used to infer the probable execution path. Table 6.1 shows a list of all theoretically possible combinations of values, how they arose and how they were categorised when using a fault model that assumed a skipped instruction was equivalent to a Nop. This set of possible patterns was obtained by simulating execution of the code shown in Figure 6-7, selectively omitting the operations on lines 1...9, and observing the result. It was also assumed that an error would be required to initiate a double execution. Therefore, there could be at most one arithmetic error within a triple that has multiple invocations.

Results for each triple were categorized as *None*, *Single* or *Multiple* to indicate whether code was not run, executed once or multiply executed. To cater for the

Ambiguous results, the analysis was performed twice. Initially defaulting to *Multiple* and probably overestimating the number of times code was executed unexpectedly. Secondly, defaulting to *Single* and probably overestimating the number of experiments that followed the intended code path.

The results obtained from the physical experiments contained a few alternative triples that could only be explained if more than two errors occurred or if the skipped instructions did more than a NOP. These results were rare but repeatable. This observation suggests that the fault simulator was good and that the physical silicon can demonstrate inexplicable but repeatable behaviour. It also highlights the value of physically testing defences rather than trusting simulated results. These additional error triples were categorised manually by considering each on a case by case basis and when in doubt, treating them as *Ambiguous*.

A custom-built test harness executed the code samples. It initialised the triples used to determine the execution path, programmed the pulse timers, signalled the laser that the test had started and was about to invoke the test sample. At the end of a test, the contents of the triples were delivered to the controlling PC. The role of the `Trap()` function was to block further laser pulses before delivering the full set of triples to the host PC. The duration of each test, and hence the number of samples collected, depended upon the expected execution time of the code fragment under test. Each test was repeated four times for each possible timing combination. This ensured that the rare occasional misfire did not materially skew the results. The collected results were first processed to categorise each triple, and from this, to infer the execution path.

This arrangement satisfied three requirements necessary for testing Call, Return and Branch operations: *i*) the only execution flow control instructions executed were those being tested; *ii*) the execution path could be retrospectively determined; *iii*) induction of arithmetic errors during the tests would not compromise the results.

6.3.1 Call and Return

When instructions can be skipped, a significant risk is that a function call may be avoided, and the caller will carry on, unaware that a significant amount of processing has not been performed. Similarly, execution may fall into the following code if a function is called, but its return operation is skipped. A skipped return will probably fall through onto another function, and when that function returns, the original caller may be unaware that additional unrelated processing has been performed.

Flow-control graphs, as proposed by Abadi [1] as a solution for CFI, could potentially be calculated by a compiler. Unfortunately, this will be incomplete in all but the most simple applications because the complete map of flow transitions can only be calculated after linking. Even if it were practical, the storage and run-time processing overheads would preclude this approach as a viable option for μCs.

The alternative approach using *Waymarks* has already been shown to be effective in μCs, as demonstrated here in Section 5.2. However, this mechanism of updating and testing state variables at each waymark has the same compile-time issues relating to completeness as identified for flow control graphs. While *Waymarks* are simple to use within a single function, they are difficult to use effectively when branching and especially difficult to use when invoking subroutines or within loops.

Another technique, usually seen in debug builds of programs, is the use of stack canaries as proposed by Cowan et al. [56]. Here sacrificial data is placed within the call frame, and its corruption indicates anomalous execution has occurred and triggers defensive behaviour.

```
1            ...
2            ldi    r0, Foo-1   // Load immediate
3            call   Foo         // Call subroutine
4         // clz                // Z-flag, Preset to fail
5            subi   r0, Foo+1   // Sub immediate
6            breq   L_ok        // Branch if zero
7            jmp    Trap        // Error detected
8            jmp    Trap        //
9         // jmp    Trap        // in case > 2 errors
10   L_ok:   ...
```

Figure 6-8: Defended Call

We formulated a hybrid of these last two techniques that can be implemented

efficiently within a code generator to defend Call and Return operations. A token with a constant value is added to the invocation parameters of a subroutine (Figure 6-8, line 2). As with a canary, this token is not part of the function's formal parameters, but its corruption indicates anomalous behaviour. The subroutine's entry code then verifies this token's value before proceeding (Figure 6-9, line. 1-3); accidental invocation would not be expected to have a valid token. Similarly, when the subroutine exits, it replaces the token with one that the caller can verify (Figure 6-9, line 10). If execution returns from the wrong subroutine, the caller will be aware that an abnormal execution path has been followed (Figure 6-8, line 5-7).

```
1   Foo: // clz             // Z-flag, Preset to fail
2           subi r0, Foo-1   // Sub immediate
3           breq Foo_ok      // Branch if zero
4           jmp  Trap        // Error detected
5           jmp  Trap        // Error detected
6        // jmp   Trap       // in case > 2 errors
7   Foo_ok: ...              // Foo body code
8                            //
9           ...              //
10          ldi  r0, Foo+1   // Load immediate
11          ret              // Return
```

Figure 6-9: Defended Entry/Exit

There are some simple constraints on the choice of values for a subroutine's tokens. Each subroutine should be allocated different tokens, or at least adjacent subroutines need differing tokens. The call and return tokens need to be different in order to identify the case where the CALL instruction gets skipped. The compiler must automatically generate the token values, and they must remain unique through the compilation and linking processes. This precludes values based on a subroutine's name as it cannot accommodate indirect invocation via pointers.

The mechanism shown here uses the callee's address -1 for the invocation token and the same address $+1$ for the return token. For static calls, the compiler can symbolically describe each token value, and the linker can handle simple arithmetic at fix-up. For an indirect call, efficient operations can calculate a token value from a function pointer.

This mechanism gives compact and efficient code for function invocation, as shown in Figure 6-8. The example is capable of trapping up to two skipped instructions

providing the zero flag is clear before either the CALL or RET operations. The code for subroutine entry and exit is similarly compact and efficient, as shown in Figure 6-9.

Similar but less compact, instruction combinations can be constructed to survive more than two skipped instructions. This may be necessary where the programmer expects a concerted attack or where the CPU architecture may skip more than one instruction at a time in a deterministic pattern, as observed by Riviere [148].

```
1  WORD Fn1(WORD a) {                // Function 1
2      TestTick(N_TICK_1);           // ----------
3      return a | 0x01;
4  }
5
6  WORD Fn2(WORD a) {                // Function 2
7      TestTick(N_TICK_2);           // ----------
8      a = Fn3(a);
9      a = Fn4(a);
10     return a | 0x02;
11 }
12
13 WORD Fn3(WORD a) {                // Function 3
14     TestTick(N_TICK_3);           // ----------
15     return a | 0x04;
16 }
17
18 WORD Fn4(WORD a) {                // Function 4
19     TestTick(N_TICK_4);           // ----------
20     return a | 0x08;
21 }
22
23 WORD Fn5(WORD a) {                // Function 5
24     TestTick(N_TICK_5);           // ----------
25     return a | 0x10;
26 }
27
28 WORD Fn6(WORD a) {                // Function 6
29     TestTick(N_TICK_6);           // ----------
30     return a | 0x20;
31 }
32
33 WORD Fn7(WORD a) {                // Function 7
34     TestTick(N_TICK_7);           // ----------
35     return a | 0x40;
36 }
37
38 void TestEntryPoint(void) {       // Entry Point
39     WORD a;                       // -----------
40     PORTA |= ((u8)1<<3);          // Timer Start
41     TestTick(N_TICK_0);           // first mark
42     a = Fn1(0);                   // call 1
43     a = Fn2(a);                   // call 2,3,4
44     a = Fn5(a);                   // call 5
45     PORTA &= ~((u8)1<<3);         // Timer stop
46     TestTick[N_TICK_9];           // last mark
47 }
48
49 WORD Fn8(WORD a) {                // Function 8
50     TestTick(N_TICK_8);           // ----------
51     return a | 0x80;
52 }
```

Figure 6-10: Call/Return Test Program

A test program (Figure 6-10) was structured to provide the opportunity for errors and the ability to detect them. For example, the uncalled functions Fn6() ... Fn8() may be executed if the preceding functions fail to return.

The `TestTick()` mechanism shown in Figure 6-6 makes it possible to deduce the execution path taken. Normal execution would be expected to indicate single registrations for ticks 0...5 & 9 while not registering ticks 6...8. This is indicated by the blue path shown in Figure 6-11. This figure also shows all of the execution paths possible when two errors may be injected. By comparing each triple with the values shown in Table 6.1 it is possible to identify a sample's execution path, even in the presence of arithmetic errors.

This test program was compiled and executed while being subjected to all possible timing combinations of one and two laser pulses. To demonstrate the defences individually and in combination, we tested six alternative compilations of the code. They were,

- **A_1: GCC**. This was the code built using the GCC toolchain. During compilation, all optimisation had to be disabled to prevent the compiler from eliminating unused code and inlining subroutine calls. This generates an artificially large and relatively inefficient output. However, it is not unrealistic when using defensive code, as these optimisations need to be disabled to ensure defences are not optimised away.

- **B_1: DCC (undefended)**. With all automatically generated defences switched off, this output provides a common reference point for the two compilers. This version is expected to exhibit similar weaknesses as the GCC generated code.

- **C_1: DCC (basic defence)**. This variant generates the code defences shown in Figures 6-8 and 6-9. It is expected to defend against two skipped instructions, where skipping an instruction is assumed to be equivalent to executing a NOP.

- **D_1: DCC (additional CLZ)**. An additional clearing of the zero-flag is inserted before the SUBI operations. As shown commented out in Figures 6-8 line 4, and

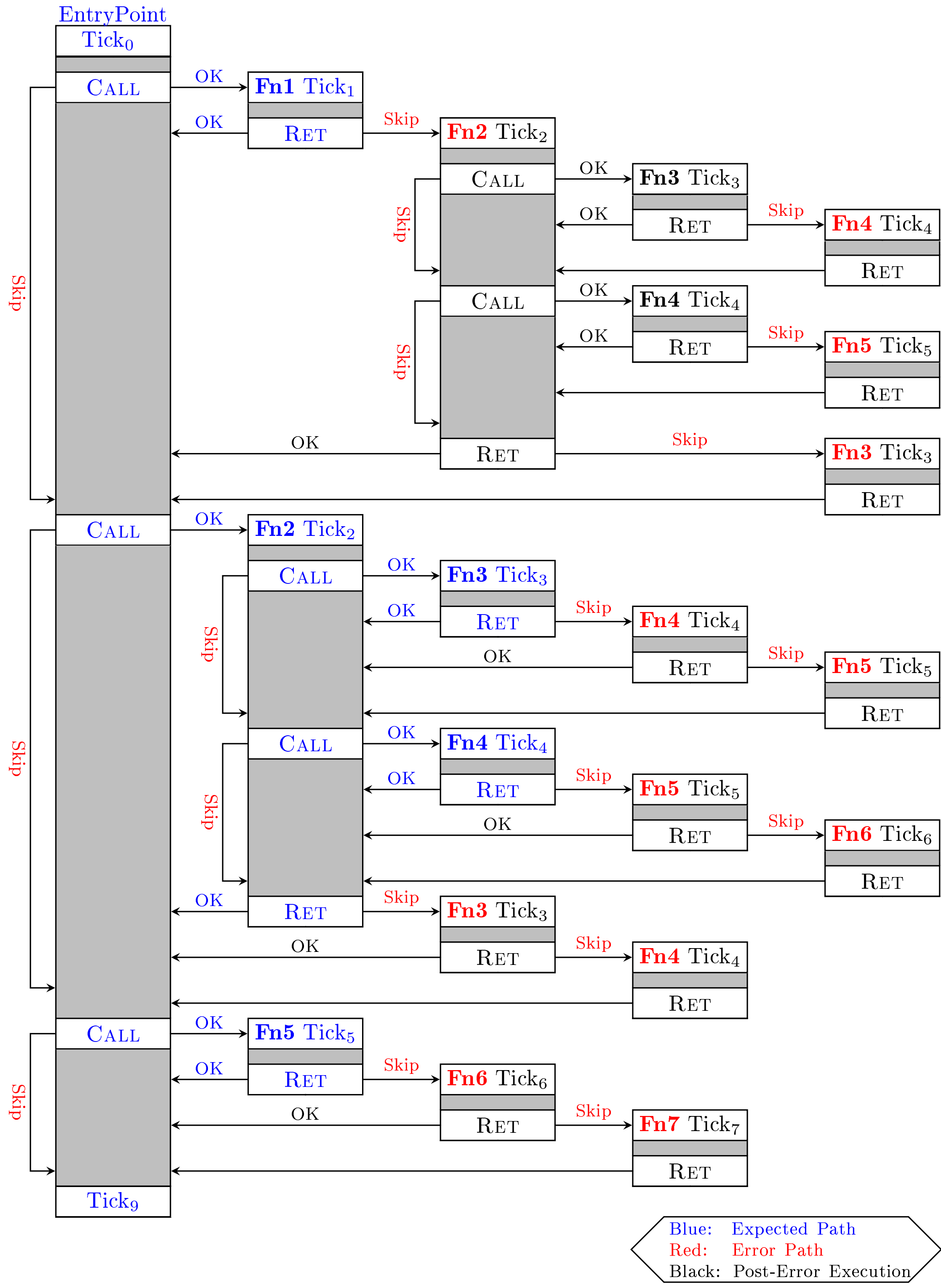


Figure 6-11: Possible Paths with Two Skipped Instructions

6-9 line 1. The reasoning is that if the zero-flag is already set and the SUBI gets skipped, then the test will fail to identify the error. Pre-arming the zero-flag to indicate failure should prevent this scenario.

- **E_1: DCC (additional JMP TRAP)**. An additional unconditional JMP was added to the trap handling. As shown commented out in Figures 6-8 line 9, and 6-9 line 6. This can only have an effect if more than two instructions have been executed erroneously.

- **F_1: DCC (additional CLZ & JMP TRAP)**. Here both of the supplementary defences D_1 & E_1 are added.

The results presented in Table 6.2 show the outcome of executing all six variants of the test program while subjecting each run to a single laser pulse attack. Laser pulses were generated for all possible moments during the program's execution; thus, the total number of samples per experiment differs according to the program's execution time.

A Run was categorized as ***Complete*** if it executed and returned to the calling test harness; ***Trapped*** if it entered the trap handler; and, ***Crashed*** if it failed to return to the test harness. The *Complete* results were further categorised to indicate the path taken during execution. A sample is ***Off-Path*** if code that should not have been invoked is unexpectedly executed or if code on the expected path gets executed more than once. It was ***Skipped*** if code on the expected path does not get executed. Finally, it is categorised as ***Path-OK***, only if the expected path is followed precisely.

- The defended code shows significantly fewer crashes. Identifying and trapping the erroneous states prevents the unwinding of potentially corrupt frames and the associated risk of retrieving a faulty return address.

- For the undefended code, sample sets A_1 & B_1, *Off-Path* or *Skipped* code accounted for approximately 6% of the *Complete* samples. Whereas in the defended code, sample sets C_1 - F_1, No samples identified as *Complete* had executed *Off-Path* or *Skipped* code.

Table 6.2: Call/Return - Single Pulse Attack

Sample Set	Undefended code				Defended code							
	A$_1$		**B$_1$**		**C$_1$**		**D$_1$**		**E$_1$**		**F$_1$**	
	#	%	#	%	#	%	#	%	#	%	#	%
Total	1000		724		884		924		884		924	
Crashed	49	4.9	37	5.1	6	0.7	7	0.8	21	2.4	21	2.3
Trapped	0	0.0	0	0.0	160	18.1	174	18.8	148	16.7	160	17.3
Complete	951	95.1	687	94.9	718	81.2	743	80.4	715	80.9	743	80.4
Complete breakdown												
Off Path & Skipped	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0
Off Path	32	3.4	20	2.9	0	0.0	0	0.0	0	0.0	0	0.0
Skipped	20	2.1	20	2.9	0	0.0	0	0.0	0	0.0	0	0.0
Path OK	899	94.5	647	94.2	718	100.0	743	100.0	715	100.0	743	100.0

- Sample sets D_1, E_1 & F_1 show that the additional defences have no value when considering a single glitch event.

The experiment was repeated for all possible timing combinations of two laser pulses. These results are presented in Table 6.3. With two glitch events, the execution path analysis yielded examples of the ambiguous states shown in Table 6.1. These rare examples could have resulted from either single or double passes of the corresponding code section. The results for *Complete* samples were therefore processed twice, once as ***Strict*** and then again as ***Lax***. The *Strict* interpretation treats ambiguous triples as examples of multiple executions; doing so probably overestimates the number of erroneous executions. The *Lax* interpretation treats the same samples as a single execution of the same triple; this time, possibly underestimating the number of faulty executions.

As with the single pulse attacks, the undefended code from both compilers shows similar ratios of *Complete* executions that executed *Off-Path* or *Skipped* subroutine calls. Approximately 10% of *Complete* runs indicate functions have not been called as expected or have been executed more than once.

The basic defended code sample, C_1, almost entirely eliminates these problems by trapping erroneous execution and reducing the number of crashes. Of the *Complete* runs, 0.1 . . . 0.7% had at some point skipped a function call or executed unprescribed code without detection by the in-built defences. The majority of these errors are for skipped function calls rather than accidental additional function execution. The number of tests performed when such errors occur explains this result. Failure to call a function can be achieved by skipping the initial CALL instruction and then bypassing the return confirmation test. Accidental execution of a function requires many more errors. An error is required to invoke the function; the function's entry test must then be bypassed; finally, the original callers return confirmation test must also be bypassed.

The additional defences tested in samples $D_1 \ldots E_1$ have a measurable effect. This implies that, for a small number of samples, more than two errors occurred, and the simple NOP substitution model of instruction skipping is imperfect.

Table 6.3: Call/Return - Double Pulse Attack

Sample Set	Undefended code				Defended code							
	A_1		**B_1**		**C_1**		**D_1**		**E_1**		**F_1**	
	#	%	#	%	#	%	#	%	#	%	#	%
Total	125500		65884		98124		107184		98124		107184	
Crashed	10478	8.3	2912	4.4	1192	1.2	1313	1.2	3980	4.1	4139	3.9
Trapped	0	0.0	3	0.0	31748	32.4	36154	33.7	29676	30.2	33460	31.2
Complete	115022	91.7	62969	95.6	65184	66.4	69717	65.0	64468	65.7	69585	64.9
Complete (Strict)												
Off Path & Skipped	44	0.0	44	0.1	0	0.0	0	0.0	0	0.0	0	0.0
Off Path	7346	6.4	3611	5.7	379	0.6	375	0.5	375	0.6	375	0.5
Skipped	4449	3.9	3388	5.4	64	0.1	24	0.0	20	0.0	4	0.0
Path OK	103183	89.7	55926	88.8	64741	99.3	69318	99.4	64073	99.4	69206	99.5
Complete (Lax)												
Off Path & Skipped	44	0.0	44	0.1	0	0.0	0	0.0	0	0.0	0	0.0
Off Path	6851	6.0	3116	5.0	4	0.0	0	0.0	0	0.0	0	0.0
Skipped	4449	3.9	3388	5.4	64	0.1	24	0.0	20	0.0	4	0.0
Path OK	103678	90.1	56421	89.6	65116	99.9	69693	100.0	64448	100.0	69581	100.0

Further investigation into the *Crashed* samples showed that in many cases, the execution had passed into the test harness code used to invoke the sample and to report on its termination state. This additional analysis was archived by strategically placing breakpoints and infinite loops within the test harness and manually examining the machine's state after repeating runs using glitch patterns previously identified as causing crashes. The test harness was common to all the samples generated by both compilers. If this test harness had itself been generated by DCC, then it is likely that many of the samples categorised as *Crashed* would have been *Trapped*. In a practical deployment of the DCC's output, the whole executable image would be similarly defended, further improving on the figures shown in Table 6.3.

6.3.2 Branching

Branching instructions are widespread and especially vulnerable to glitch attacks. The instruction itself may be skipped, in which case the code immediately following it will be exercised instead. Alternatively, faulty arithmetic before a conditional branch may make a correctly executed instruction take an inappropriate path.

As with the CALL/RET behaviour, the first scenario is difficult to defend at the source code level. The second scenario is the one most commonly defended in source code and is considered in the Arithmetic Defences section below.

6.3.2.1 Branch Instruction

The efficient and straightforward defence the compiler can add is to make both the positive and negative outcomes of a test branch to their respective code fragments (Figure 6-12, line 2&3).

By making the fall-through option a `Trap` (line 4), we can handle the skipping of either of the branch instructions safely. The option of repeating the `jmp Trap` instruction (line 5) is used to add immunity to multiple skip events. This code structure cannot be directly described in source code, and this is an example of a defence that can only be implemented within a compiler (or as hand-crafted assembler

code).

```
1           ...
2           breq L_eq           // Branch if equal
3           brne L_ne           // Br. if not equal
4           jmp  Trap           // Error detected
5        // jmp  Trap           // <optional>
6
7   L_eq:   ...                 // Then body code
8           jmp  L_cont         //
9           jmp  Trap           //
10       // jmp  Trap           // <optional>
11
12  L_ne:   ...                 // Else body code
13
14  L_cont: ...                 // common continuation
```

Figure 6-12: Defended Branch

Once again, a simple test program was devised to test the efficacy of the defences when under attack - Figure 6-13. The start of the attack was synchronised with the first 'if' statement. This ensured the decision behaviour was being attacked rather than the arithmetic accuracy of the tested data. The execution path could be retrospectively inferred using the same triple counter logic used in the CALL-RET testing. Under normal, unperturbed conditions, the triples 1, 3, 5 & 6 would be expected to be set. Faulty behaviour could be detected by failure to set these ticks or by ticks 2 & 4 being set. The use of macros, `IF_LT` & `IF_GT`, added flexibility to enable the same code to be reused in later tests (Section 6.3.2.2) combining branch defences with Branch Arithmetic Defences.

To compare the compilers and to cover the expected error scenarios, five alternative compilations of the code were generated and tested. They were:

- **A_2: GCC**. This code was built with the GCC tools. Optimisations were reduced to ensure the initialised variables used in the comparisons were not substituted for constants and that the code path for the theoretically unused path was not removed.

- **B_2: DCC (undefended)**. The defensive compiler was used with defences disabled. The expectation was that this should display similar vulnerabilities to the GCC version.

```
#define IF_LT(x,y) if ((x)<(y))
#define IF_GT(x,y) if ((x)>(y))

    WORD a=1;
    WORD b=2;

    ...
    PORTA |= ((u8)1<<3);       // Timer Enable
    IF_LT (a,b) {
        TestTick(N_TICK_1);
    }
    else {
        TestTick(N_TICK_2);
    }
    TestTick(N_TICK_3);

    IF_GT (a,b) {
        TestTick(N_TICK_4);
    }
    else {
        TestTick(N_TICK_5);
    }
    TestTick(N_TICK_6);
    PORTA &= ~((u8)1<<3);      // Timer Disable
    ...
```

Figure 6-13: Branching Test Program

- **C_2: DCC (basic defence)**. The defensive compiler generating code as per Figure 6-12.

- **D_2: DCC (Jmp Trap ×2)**. As per C_2 with additional Jmp Trap instructions added (Figure 6-12, line 5&10) to catch the case where the first Jmp may have been skipped.

- **E_2: DCC (Jmp Trap ×3)**. As per D_2 but adding a third Jmp to the trap handler.

The results of a single pulse attack performed against each compilation are shown in Table 6.4. The most common error detected is *Off-Path-and-Skipped*. This behaviour occurs when the expected code block gets skipped, and the complimentary code path gets executed; in other words, the opposite outcome to that prescribed by the `if` clause. This behaviour is attributable to two error conditions: *i*) performing the comparison on corrupted data which results in a faulty conclusion, or *ii*) failing to take a prescribed branch and falling through into the code block associated with the opposite outcome. The decrease in the number of *Off-Path-and-Skipped* outcomes in

Table 6.4: Conditional Branches - Single Pulse Attack

Sample Set	**Undefended code**				**Defended code**					
	A_2		**B_2**		**C_2**		**D_2**		**E_2**	
	#	%	#	%	#	%	#	%	#	%
Total	344		308		316		316		316	
Crashed	46	13.4	1	0.3	0	0.0	0	0.0	0	0.0
Trapped	0	0.0	0	0.0	13	4.1	18	5.7	16	5.1
Complete	298	86.6	307	99.7	303	95.9	298	95.3	300	94.9
Breakdown of Complete										
OffPathAndSkipped	20	6.7	22	7.2	15	5.0	14	4.7	14	4.7
OffPath	12	4.0	12	3.9	4	1.3	0	0.0	0	0.0
Skipped	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0
PathOK	266	89.3	273	88.9	284	93.7	284	95.3	286	95.3

the defended code suggests the defence behaves as expected for the skipped instruction case *ii*. However, corrupt comparison, case *i*, is the dominant influence. This is unsurprising as case *i* can result from any one of many possible instruction errors. In contrast, case *ii* occurs as a result of the single branch instruction being skipped. The results associated with *Off-Path* also support this interpretation. The code for both the positive and negative outcomes of the `if` clause is being exercised. The most likely cause of this would be skipping the branch instruction at the end of the first conditional code block (Figure 6-12, line 8) and execution falling through to the code block for the `else` clause. The defended code effectively eliminates this behaviour by trapping after skipped branches.

The same code samples were subjected to all possible timing combinations of two laser pulses, shown in Table 6.5. Of the *Complete* runs, proportionately fewer reported *Path OK*, as would be expected given the additional error opportunities.

Off-Path-and-Skipped is attributable to taking a conditional branch for the wrong reason. Here the proportion of erroneous samples increased and the defences $\mathbf{C}_2$ - $\mathbf{E}_2$ were equally effective. Strongly suggesting the arithmetic that formed the conditional state was corrupted and the defences were not invoked.

Off-Path is attributable to failing to take a branch. Here defence $\mathbf{C}_2$ is an improvement on $\mathbf{B}_2$, but less effective that $\mathbf{D}_2$ or $\mathbf{E}_2$. In this scenario the defences are clearly having a meaningful effect as seen by the increase in the number of *trapped* executions. This demonstrates the benefit of repeating the `Jmp Trap` when multiple pulses are delivered within an attack.

While the defences work, it is clear that the weak link is the arithmetic that establishes the testable state for the conditional branches.

Table 6.5: Conditional Branches - Double Pulse Attack

Sample Set	**Undefended code**				**Defended code**					
	A_2		**B_2**		**C_2**		**D_2**		**E_2**	
	#	%	#	%	#	%	#	%	#	%
Total	14964		12012		12640		12640		12640	
Crashed	37	0.3	89	0.7	0	0.0	0	0.0	0	0.0
Trapped	0	0.0	0	0.0	1231	9.7	1648	13.0	1587	12.6
Complete	14927	99.7	11923	99.3	11409	90.3	10992	87.0	11053	87.4
Complete(Strict)										
OffPathAndSkipped	1641	11.0	1716	14.4	904	7.9	893	8.1	893	8.1
OffPath	965	6.5	836	7.0	591	5.2	268	2.4	274	2.5
Skipped	2	0.0	8	0.1	0	0.0	0	0.0	0	0.0
PathOK	12319	82.5	9363	78.5	9914	86.9	9831	89.4	9886	89.4
Complete(Lax)										
OffPathAndSkipped	1641	11.0	1713	14.4	904	7.9	894	8.1	893	8.1
OffPath	965	6.5	311	2.6	16	0.1	4	0.0	4	0.0
Skipped	2	0.0	11	0.1	0	0.0	0	0.0	0	0.0
PathOK	12319	82.5	9888	82.9	10489	91.9	10094	91.8	10156	91.9

6.3.2.2 Branch Arithmetic

It is clear from the simple conditional branch defences above that arithmetic errors are still the primary cause of *Off-Path-And-Skipped* errors. Where the branch decision depends on arithmetic accuracy, the branch instruction defence has limited effectiveness, as seen above. The usual approach is to repeat a comparison and treat contradictory results as evidence of an attack. In simple cases, this could also be automatically implemented by a compiler. However, there are many cases where this cannot be archived safely, particularly where repeating a comparison may change the semantics of an expression.

```
1  boolean Trap(void);    // non-returning function.
2
3  #define IF_LT(a, b)    if ((((a) < (b)) && (((b) > (a)) || Trap())) || \\
4                              (((b) > (a)) && Trap()))
5
6  #define IF_GT(a, b)    if ((((a) > (b)) && (((b) < (a)) || Trap())) || \\
7                              (((b) < (a)) && Trap()))
```

Figure 6-14: Double Checking Macros

The experiment was repeated after replacing the comparison macros with those shown in Figure 6-14. This demonstrates the effect of a hybrid defence using traditional source-code arithmetic defence and compiler-generated defended branches.

The results are presented in Table 6.6. In all cases, the number of *Off-Path-And-Skipped* has been reduced, confirming the earlier suggestion that this category of errors was attributable to unperturbed execution, guided by erroneous data. Double testing the data has predictably reduced the incidence of these errors.

For GCC with a single pulse attack, we see mainly *Off-Path* errors typical of skipping the jump over the code for the `else` block. DCC catches and traps this behaviour effectively, as we saw previously. With double pulses attacks, DCC traps significantly more errors and, of the runs that terminate cleanly, the proportion demonstrating erroneous execution is similarly reduced.

Table 6.6: Macro Defended Conditionals

Sample Set	Single Pulse				Double Pulse			
	GCC		DCC		GCC		DCC	
	#	%	#	%	#	%	#	%
Total	372		372		17484		17484	
Crashed	0	0.0	0	0.0	3	0.0	0	0.0
Trapped	20	5.4	36	9.7	1818	10.4	3170	18.1
Complete	352	94.6	336	90.3	15663	89.6	14314	81.9
Complete(Strict)								
Off Path & Skipped	4	1.1	0	0.0	551	3.5	24	0.2
Off Path	12	3.4	0	0.0	1024	6.5	770	5.4
Skipped	0	0.0	0	0.0	4	0.0	0	0.0
Path OK	336	95.5	336	100.0	14084	89.9	13520	94.5
Complete(Lax)								
Off Path & Skipped					551	3.5	24	0.2
Off Path	n/a		n/a		354	2.3	8	0.1
Skipped					4	0.0	0	0.0
Path OK					14754	94.2	14282	99.8

6.4 Code Efficiency

Besides the efficacy of the defences demonstrated above, there is invariably a competing need to keep code efficient in terms of both size and execution time. A clue to the efficiency of the defensive compiler's output is seen in the tables for single pulse attacks, (Tables 6.2, 6.4 & 6.6). Here the *Total* sample counts are directly proportional to the execution time of the expected, unperturbed path through the code. However, these are highly artificial test scenarios.

A better idea of what impact the automatically generated defences would have, in more realistic scenarios, can be gained by compiling real-life code samples. Three code samples were investigated, each providing a different insight into DCC's output.

There are two viable approaches to generating deployable code using the GCC toolchain,

- Turn off all optimisations and accept the most inefficient code, confident that source code implemented defences remain in the object code. This is the minimal effort approach, but the code will be unnecessarily large and inefficient. It is identified in the tables as **'No Optimisations'**.

- Compile with basic optimisations (GCC command-line switch -O1) enabled. The code is significantly better but requires a visual inspection to confirm that defences remain. When the compiler removes defences, they need to be modified or re-implemented to make them immune to the compiler's optimisations[§]. This approach is both time-consuming and prone to accidental errors. It does, however, generate smaller and faster code. It is identified in the tables as **'-O1 optimised'**.

[§]For higher levels of optimisation, this approach is impractical because they are too efficient at removing redundant code.

6.4.1 Large Application

The source code from a full implementation of a Global Platform V2.2 card manager [77] with a Javacard Classic V3.0.5 Virtual machine [130] was variously compiled. This application was originally targeted at an AVR smartcard core; it included source code level defensive coding and had been independently reviewed for payment scheme accreditation. The code's original target was a high-security device with bespoke peripherals and tools licensed explicitly for a particular commercial product development. Details of the device, beyond the fact it was an AVR core, are also covered under an NDA, making the tools for physical testing and emulation off-limits for this investigation.

For this exercise the code was compiled for nearest equivalent publicly available μC, the AVR xmega [13]. Sections of the original code that accessed hardware peripherals not present on this new target had to be replaced with dummy functions, which made meaningful code execution impossible. The source code could be compiled and linked to gain insights into the code volume, but the execution efficiency of the binary could not be measured here. The code volume results are presented in Table 6.7 and performance is considered separately in Section 6.4.5.

Table 6.7: Code Volume

Global Platform 2.2 + JC 3.0.5 VM		**Program Size**	**%**
GCC	Source code defences	242780	133%
	... -O1 optimised	182874	100%
DCC	Source code defences	261224	143%
	... + Branch Defence	282496	154%
	... + CALL/RET Defence	329782	180%
	... + Branch & CALL/RET Defences	351054	192%

These figures suggest that GCC unoptimised output is only a little bit more efficient than DCC's defenceless output. With defences turned up to maximum, the code size here increases is 90%. Examination of the generated code shows that GCC uses

a far more sophisticated register allocation strategy for local variables and efficiently removes redundant re-fetching of variables when their values are reused immediately after store operations. The most expensive defence is the CALL-RET protection, and this effect is exaggerated as a result of the source code programming style that uses many small functions to assist readability in the expectation that the compiler will ultimately place the code inline. The source code also includes defences that are unnecessary when DCC adds additional equivalent defences. These automatically inserted defences are also added to the unnecessary manually defined defences, making them additionally defended by DCC; unnecessary calls and branches become defended unnecessary calls and branches. It is, therefore, safe to assume these figures represent an unrealistic worst case.

6.4.2 Call Intensive Code

It is more realistic to compare defended code, compiled with GCC, against undefended code compiled by DCC. Both processes generate defended object code. This was tested by preparing two source code samples. One directly from the original Java Card sources, it is the original **Defended** version and contains all of the source level defences. The other, **Defenceless** version, has the source code defined flow-control defences removed. This more meaningful comparison is highlighted as green-tinted rows in the following tables.

The bootstrap code for the Java Card application was used because it can be executed on the DUT up until the point it passes control to its communications manager. Table 6.8 shows the resulting object code sizes and execution times when the two code versions are compiled with the two compilers.

This bootstrap code has many function calls within it and few branches; this biases the figures towards the CALL/RET defence.

6.4.3 Branch Intensive Code

A second code fragment, rich in branches and with fewer function calls, was similarly tested. The subroutine for ISO/IEC 9797-1 Algorithm 3, the so-called banking MAC [90] was used here. In the absence of a DES co-processor, an empty no-op function was substituted to replace the co-processor manipulation routine, leaving the simulated in/out buffer unchanged. All source-code prescribed function calls, data movements, exclusive-or operations and source code defined defences were performed. A test harness invoked this library function to generate the MAC of a 44 byte string, and all variants executed correctly, generating the same MAC values. The results are shown in Table 6.9.

6.4.4 Code Size

In both code samples, DCC's output after compiling defenceless code falls mid-way between GCC's outputs for the optimised and unoptimised builds of defended code. All of the outputs are, of course, defended, but DCC generated its code from simple undefended input. The cost is highest where function calling predominates, as all Call & Return operations are instrumented with integrity checks. The application of these defences is universal with DCC, whereas, in the GCC/*Defended* code combination, the defences were only applied to security-critical events. This represents a 10 . . . 25% code volume penalty in exchange for significantly faster development time, the confidence of universal coverage, and stronger defences.

6.4.5 Code Performance

The Java Card application's bootstrap code initialises a set of state variables during its first run. Subsequent runs of this code identify the initialised state and bypass the set-up processing. These two execution states are security-critical and consequently heavily defended. They are identified in Table 6.8 as the **Init** and **Initialized** columns respectively. This Call intensive code shows a 10% execution time penalty.

The branch intensive MAC algorithm shows a 10% speed improvement. Much of the redundant code DCC generates to implement branch defences is not executed unless there is an error. Therefore the code volume expansion does not manifest itself as additional code to be executed. Defences implemented at the source code level and seen by GCC are executed, highlighting an advantage of compiler-generated defences.

Table 6.8: Bootstrap - Code Compilation

	Source code	Compiler setting	Program Size	Program %	Execution time: Init cycles	Init %	Initialized cycles	Initialized %
GCC	Defended	No optimisations	1778	173%	2328	210%	1045	207%
		-O1 basic optimisations	1028	100%	1106	100%	504	100%
	Defenceless	No optimisations	1610	157%	1665	151%	852	169%
		-O1 basic optimisations	906	88%	755	68%	407	81%
DCC	Defended	No compiler defences	1478	144%	1536	139%	668	133%
	Defenceless	Branch & CALL/RET Defences	1288	125%	1137	103%	553	110%

Table 6.9: Banking MAC - Code Compilation

Banking MAC ALG_DES_MAC8_ISO9797_1_M1_ALG3		Program Size	Program %	Execution cycles	Execution %
GCC	Defended source, without optimisation	2862	118%	5795	108%
	Defended source, -O1 optimised	2416	100%	5380	100%
DCC	Defended source code	3046	126%	6280	117%
DCC	Defenceless source code	1956	81%	4460	83%
	... + Branch Defence	2228	92%	4645	86%
	... + Call / Return Defence	2394	99%	4635	86%
	... + Branch & CALL/RET Defences	2666	110%	4820	90%

6.5 Summary

It is practical to build a code generator that automatically inserts effective defensive code against multi-pulse fault attacks, and this is possible even with modest resources. Even though the code is not highly optimised, it compares favourably with that of mainstream compilers that need to have their optimisations constrained to prevent the elimination of deliberately redundant code. Most significantly, from a security standpoint, this approach eliminates the opportunity for the accidental omission of defences by the programmer and undetected removal of those same defences by a compiler's optimiser.

The defences demonstrated here cannot be described via the source code syntax. Their computational efficiency compensates for DCC's relatively poor general optimisation capabilities. Mainstream compilers may optimise the code better, but defences that can be implemented through them are less efficient. The automatically inserted defences demonstrated here constitute a novel and effective CFI defence.

A significant advantage is the speed of development of defended code. There is no need to instrument the source code with self-checking code. Such source-code defences are expensive in terms of development overhead, prone to accidental omission and, lead to inefficient code. These defences can be omitted and delegated to the DCC. Knowing that the compiler will insert the defences leaves the application programmer free to concentrate on the functional accuracy and readability of the code, uncluttered by double tests or code that has to be illogically organised in anticipation of skipped branches or out of sequence execution.

The fault model that assumes skipping instructions is equivalent to executing a NOP is surprisingly accurate. Despite its simplicity, defences based on this failure mode can trap the majority of errors injected into the program's control flow.

The methodology used here enables experimentation with various defence mechanisms. It is relatively easy to test their efficacy and adopt the most suitable for a particular target platform and presumed attack vectors. The physical tests performed on the defences found rare but repeatable exceptions to this fault model.

Defences were extended to trap these rare events, and the impact of the changes on the whole application's code volume was easy to reevaluate. The same approach will be applicable to other target μCs.

The effectiveness of the out-of-order defences implemented by DCC will increase if more of the support code and system libraries are compiled with the same tool. This particular CFI defence wraps and protects each function individually. Accidental crashes will be trapped at the earliest possible opportunity, and stack smashing attacks, such as the *return-into-library*, face the additional complication of preparing the dummy parameters as well as faking return addresses.

Defences against arithmetic errors have not been attempted, and these defences currently remain the programmer's responsibility. Arithmetic defences ultimately rely on a decision about whether or not to trust an outcome. This decision making process, critical to all arithmetic defences, can be securely defended with a combination of macro augmented source-code and DCC generated CFI defences.

As measured by execution speed, the cost of defences is acceptable here, it being $< 10\%$ in our sample. In many cases, the automatic defences are smaller and faster than the equivalents described in the source code.

6.5.1 Room for Improvement

This investigation considered attacks using two accurately timed laser pulse injections. Resistance to two pulses is a significant improvement upon current practice. Unfortunately, most source-code implemented defences fail and need to be used in combination to survive multiple pulse attacks, as seen earlier in Figure 5-17. All of the source-code defences tested demonstrated corrupted data outcomes with just a single pulse. Two pulses is a practical limit for a blind attack as it takes a long time to perform all the possible test patterns over anything more than small fragments of code. Some of the tests performed here took several days to collect the samples.

Where the defences consist of repeated fragments of identical code, an attacker may use a multi-pulse template (as described in Section 4.3) to search for a vulnerability. For these cases, the defence lies in the device's ability to remember it has been

attacked and change its behaviour in future. This is achievable with a more intelligent `Trap()` function. With the efficiency of the traps demonstrated here, a multi-pulse template attack will most probably trigger a trap before discovering the appropriate synchronisation for it to be effective. Therefore, the additional complexity and code volume required to survive more than two pulses may be overkill.

Purely on a practical note, some simple improvements could be made to the current DCC. These mainly relate to ease of use and impact the tool's practical deployment rather than its ability to generate secure code. Integration of the pre-processor into the compiler and improved integration with the target device's debugging tools would greatly facilitate application development. Currently, the best approach is to develop and debug using GCC and then re-compile the tested code with DCC. In a similar vein, the relatively simple optimisations relating to register allocation and elimination of redundant store/fetch operations, as demonstrated by GCC, need to be addressed. This quick-fix will have a significant beneficial impact on the size of DCC's output code.

Alternatively, having proved the concept is viable, it may be worthwhile implementing the defensive code generator as part of a quality open-source compiler. This approach would take advantage of that compiler's optimisation capabilities while augmenting the calls, returns and branches with the security refinements demonstrated here. The caveat here, of course, would be that many arithmetic defences rely on redundant code and mainstream compilers are particularly efficient at removing this.

Chapter 7

Security Impact

Contents

7.1 Implications **235**

7.1.1 Accessibility 235

7.1.2 Repeatability 237

7.1.3 Attackers 238

7.1.4 Defenders 239

7.1.5 Exploitations 240

7.1.6 Development, Review and Certification 242

7.2 Strategies **244**

Characterising the error responses of a DUT, developing a fault model and then prescribing and testing defences is a practical workflow for application development. When treated as a whole, the weakness of any particular stage is compensated for by the others. This can be achieved with easily accessible equipment and tools. And, since the capital expenditure is low, the methodology can be employed by modestly funded developers.

The economic and technical arguments relating to an attacker's motivations to attack have also changed. With operational expenditure far exceeding capital costs, amateur hackers (and other time-rich but cash-poor adversaries) must be considered to be viable threats. Consequently, the risk model used when prescribing defensive strategies for low-cost devices needs to be recalibrated.

During this investigation, we have established two key features: One, within a readily controlled environment, fault induction is reliably repeatable, and two, this can be achieved with modest resources.

While some errors are random, e.g. SEUs, control of the error stimulus induces errors that show consistent repeatable properties. Of the controllable parameters, two play a significant role in determining the response; they are *i*) Timing and *ii*) Location. The other two controllable parameters are less important. *iii*) Duration of a stimulus does not significantly alter the response, but longer durations do appear to reduce the power required to induce an error. *iv*) Power is effectively binary. That is to say, once a threshold is reached, an error occurs, but the nature of the error remains unaffected by the magnitude power.

The ability to choose the timing of an error and to confidently predict its effect is a serious threat to any application. Similarly, the ability to achieve this with readily available tools has multiple implications.

7.1 Implications

The most notable outcomes of this investigation have been identifying error repeatability and demonstrating the low-cost threshold for accessibility to the tools needed to investigate error injection.

Using the tools developed here, we have demonstrated a powerful and efficient attack that exploits the repeatability of error responses. These results also help defenders define and refine defences. Furthermore, the low cost of these tools makes them accessible to the whole spectrum of application developers.

7.1.1 Accessibility

A surprising outcome of this study has been the realisation that the tools required to perform semi-invasive attacks are widely available. While other researchers focussed on ever more powerful equipment and analysis in the hope of accurately defining the precise effects of an error, this study has shown that such precision is unnecessary.

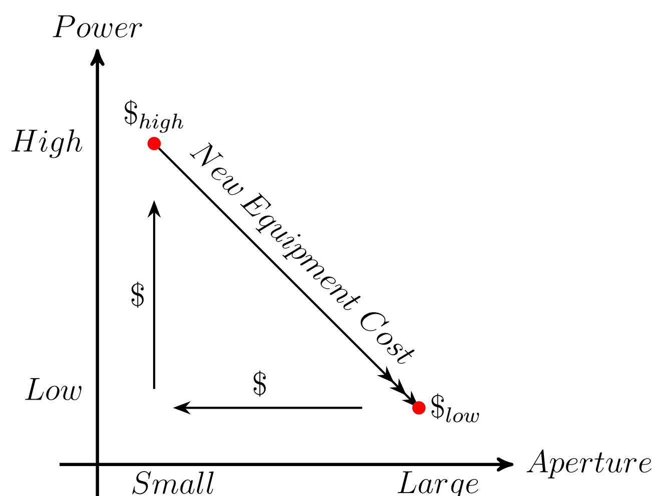


Figure 7-1: Equipment Availability

Many characteristic errors can be induced using relatively imprecise focus and with modest laser power. These are the errors an attacker will try to exploit first.

High power lasers cost more and are more difficult to control accurately. Similarly, fine focus requires expensive optics and vibration-free support for the equipment. Conversely, as illustrated in Figure 7-1, low-power and imprecise focus are relatively low-cost. These properties are readily attainable using easily obtained equipment. We must therefore assume a larger number of potential attackers exist.

From the defender's perspective, a device can be characterised using readily available samples while operating it in its normal release mode, i.e. without attached debuggers. An appropriate fault model can be defined from the characterisation, which can then be used to prescribe relevant defences. Others, e.g Dureuil [62]*, have observed that, from the attacker's perspective, it is the error effect that counts and not the detail of the mechanism. The methodology used here demonstrates that this observation is equally applicable to defenders. It also undermines the premise that a detailed understanding of a device's ISA is a prerequisite for effective software defences. Defences can be tested using the same equipment that was used to perform the characterisation. It may sound simple and obvious, but the approach is far from

*"While the exact fault is not of interest when attacking a single application (only the success of the attack matters)..." — Dureuil 2015

common industrial practice.

Multi-pulse attacks had been presumed to be infeasibly expensive and complicated [23, 118]. We can confidently dismiss this line of argument, and developers must consider multi-pulse attacks as part of their defensive strategy.

The important characterisation phase can be performed at a low cost without needing access to the intended target sample. This lack of obstacles means an attacker can perform much of the required preparation work without the risk of damaging samples of the target and can eliminate the risk of being discovered while doing so. The low-cost and low-risks involved in attacking a target mean even modest rewards may justify the expense of an attack. Many more devices should therefore be considered vulnerable.

Defenders cannot rely on expensive or generally inaccessible equipment to deter attackers.

7.1.2 Repeatability

Most obviously, repeatability, or *Test–Retest Reliability*, invalidates earlier studies that assumed random events and used Monte-Carlo simulation techniques. It is the difference between navigating a path to a destination or relying on Brownian motion to get there. If deterministic fault injection can be combined with knowledge of an executing program, then efficient strategies can be devised to modify that program's outcome.

While the errors may be repeatable, their outcomes are not necessarily controllable. Characterisation of errors for a particular device identifies the properties of errors, and these error responses then define the toolkit available to both attacker and defender. An attacker will try to exploit the effects that will happen when under attack. While at the same time, the defender uses those same properties to create traps for erroneous behaviour and uses the uninfluenceable outcomes to guard critical code. The importance of error characterisation as a first step when defining defences cannot be overstated.

Error repeatability is not unique to our chosen DUT, or our chosen fault induction mechanism†. Other researchers have noted similar effects in other devices. The effects are common to the semi-conductor materials and not a feature of the device design implemented using them. While the consequence of an error will relate to the device's design, it is unavoidable that a device can be perturbed and that the perturbation effect will be repeatable. Different samples of a particular type of device behave similarly. Thus characterisation can be performed on a per type basis, and defences can be tailored accordingly.

7.1.3 Attackers

Characterisation identifies behavioural weaknesses and discovers how, when and where to apply a stimulus that induces an error. This knowledge is transferable to other samples of the same device. Having identified a weakness, an attacker will then seek to exploit it. This process is greatly simplified with the equipment developed during this study. The low cost also means the techniques are available to a wider audience.

It has been recognised for a long time that individual errors are sufficient to break undefended code. Undefended code has always been regarded as vulnerable, but repeatability makes this more concerning. Predictable attack outcomes help define search strategies for locating weaknesses. In contrast to earlier random outcome fault models, the attacker does not need to perform many repetitions of a stimulus before testing an alternative. Search times are consequently reduced, providing the attacker has a reliable method to synchronise fault injection with the executing code.

Defences that can resist multiple errors are significantly harder to defeat without knowledge of the code being attacked. The difficulty here lies in the combinatorial explosion occurring when searching for suitably timed patterns of multiple pulses. There is little additional complexity involved in a multi-pulse attack and the defence results from the increased time needed to search for an effective pulse pattern.

†Riviere [148] performed a similar characterisation on the ARM Cortex-M4 using EM pulses to induce errors and to develop a target-specific fault model. Colombier [52] also identified instruction skipping as the most readily exploitable error that is inducible in an ARM μC.

Where the precise instruction sequence within a defence is known, the Creeping Barrage (Section 4.3) attack makes multi-pulse tolerant defences just as vulnerable as undefended code. The attacker has to align a single event, delivery of a predefined pulse pattern, with a vulnerable operation. This process is no more complicated or time-consuming than using a single pulse against an undefended target.

7.1.4 Defenders

The characterisation that potentially assists an attacker is also the source of the defensive mechanisms. Repeatability of errors means the defender can predict the error response and insert traps for erroneous behaviour. Therefore, defences against single errors are comparatively straightforward and inexpensive in terms of performance and code volume. This is achieved by placing an application's functional behaviour so that errors bypass it rather than invoke it.

Small fragments of defended code can be tested with the same tools used in the characterisation. Individual defensive code structures can be exhaustively tested. Any flawed assumptions made by misinterpreting results in the characterisation stage can be identified, and defences can be refined.

We have also demonstrated that defences can be built into a compiler, enabling the systemic deployment of defences within an application. It also enables the inclusion of defensive code sequences that would otherwise be impossible to achieve via a regular compiler. Furthermore, this removes the opportunity for oversight by the programmer. However, compiler-generated code introduces additional predictability into the structure of defended code and potentially assists an attacker who can then design a suitable pulse time template for a Creeping Barrage attack.

Liberal inclusion of defences appears to be the best defence against a Creeping Barrage. A Creeping Barrage will probably trigger a trap before the perfect alignment is identified. Therefore traps that modify a device's future behaviour will stop a barrage from progressing.

Traps are vulnerable to errors too. It is essential that traps cannot be faulted into returning; i.e. if they are not implemented as in-line code, they should be jumped to rather than called.

7.1.5 Exploitations

Easily accessible tools for error induction offer opportunities to both attackers and defenders. It is a concern that the outcome of this study may encourage attacks on devices that would otherwise have been ignored. However, the ability to recognise vulnerabilities and test defences significantly improves the pre-existing development environment. Historically, uncertainty relating to the nature of errors has led to the inefficient deployment of defences. Sometimes defences were ignored in the misguided belief that attack would be difficult and uneconomic. Conversely, on occasions, excessive defending resulted in inefficient code. This study has provided a workflow that can identify the most significant threats and compare the efficacy of defences.

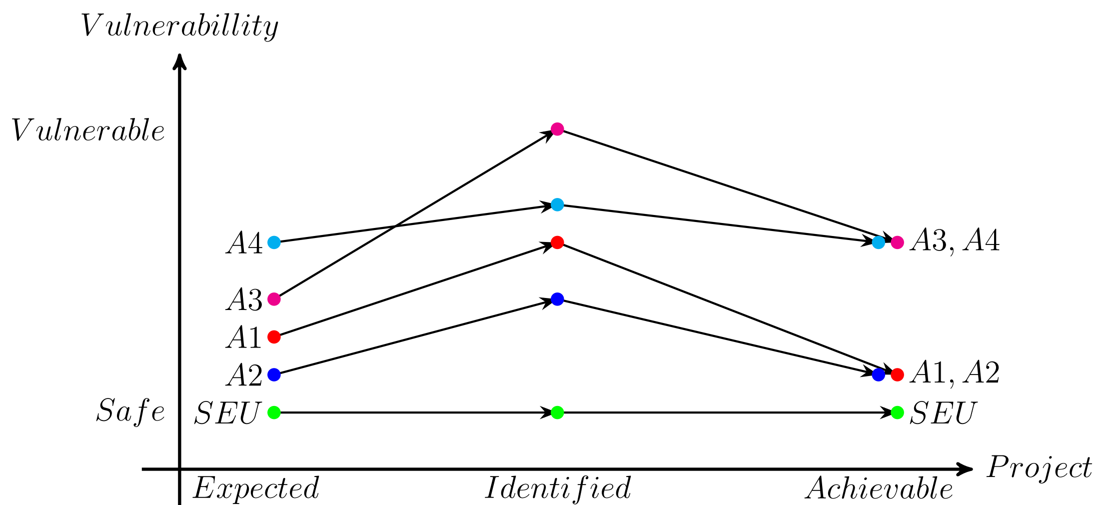


Figure 7-2: Device Vulnerability

Figure 7-2 shows how the perception of threats has evolved through the course of this study. The initial *Expected* vulnerabilities were shown to be overly optimistic after the early characterisation work. This work identified misconceptions about the nature of induced errors leading to the *Identified* threat recalibration. The later work,

evaluating defences and automating defence generation, corrects the misconceptions and, to a large extent, restores the previously assumed range of vulnerabilities attributable to differently equipped attackers.

By considering the adversaries described in section 2.2 and the spontaneous risk of SEUs, we can see how the results of this study can be used to improve a device's resilience when under attack.

SEU **Random single events** — These rare, unpredictable events are akin to single pulse attacks. As such, the historical defence techniques of crude double checking values and decisions remain effective, and the device's vulnerability does not need to be reevaluated.

A1 **Clever Outsiders** — Knowledge of the likely nature of an error response assists this class attacker when making the educated guesses required to perpetrate an attack. The reliable repeatability of responses also assists the outsider by speeding up the searches for effective pulse patterns. Therefore, the device is more suspectable to attack than had previously been assumed.

The same knowledge assists the defenders. Appropriate use of the defences discussed in this study, combined with the compiler's ability to apply them universally, shows it is possible to improve the resilience of a device against this group of attackers.

A2 **Script Kiddies** — Attack via this mechanism presumes a vulnerability has been discovered and made exploitable. All other attack classes have been identified as more potent than previously supposed; therefore, the likelihood of an attack being automated will also increase. The predictable nature of compiler-generated defences means that once the parameters for an attack have been identified, it is likely that the same parameters will be applicable to other parts of the device's code; therefore, the incentive to generalise and automate an attack is similarly increased.

A3 **Knowledgeable Insiders** — Combining predictable error responses with inti-

mate knowledge of the code makes this the most potent threat. Knowing exactly which instructions need to be skipped makes attacks quick and efficient.

The exploitable skipping weakness can be eliminated for low numbers of pulses, but large numbers of consecutive skips can be prescribed if the code under attack is known. Thus, even after applying improved defences, it is likely that this threat is more potent than has previously been assumed.

A4 **Funded Organizations** — In the absence of physical defences, attacks by well-funded organisations have historically been the most biggest threat to a μC. Knowledge of existing weaknesses increases the device's vulnerability to the extent that it simplifies or prioritises the deployment of different attack mechanisms. Defences against the predominant error category (skipping in this instance) will restore the status quo, but this group of attackers will always be amongst the most capable adversaries.

7.1.6 Development, Review and Certification

The study has identified additional skills required to generate secure code when using the currently available development tools. The specialist skills required by developers, therefore, need to be reconsidered; Figure 7-3.

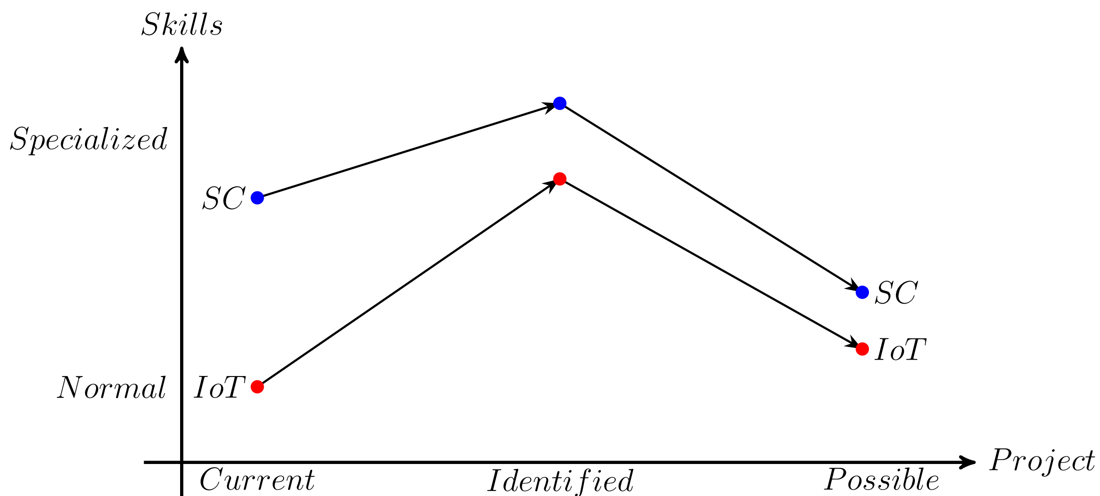


Figure 7-3: Development Skills

SC **Smart Cards** and similar security controllers — Effective defences rely on code structures that cannot be described in high-level languages. As a result, it has become apparent that developers need to understand a device's modes of failure and implement defences in assembler language.

The defensive compiler developed during this study demonstrates that it is possible to isolate the developer from understanding the CFI errors and the need to work with assembler code. However, specialist knowledge relating to a device's physical defences will still be required. Overall the speed and cost of development can be improved while simultaneously reducing a device's susceptibility to attack.

IoT **General Purpose μCs** — Devices relying solely on software defences will require specialist skills to make them suitably secure. It is, however, unlikely that the smart card industry's discipline relating to documentation, traceability, and test coverage will be required. With the adoption of a defensive compiler, it is conceivable that adequate security can be automatically generated. Normal development practices would be possible, providing the programmer was aware of the ongoing need to verify arithmetic results.

The workflow implied by this study changes the emphasis of effort involved in security review and certification of application code. Characterisation of error behaviour removes the uncertainty relating to defensive strategies and simplifies the code review process. Furthermore, automated defensive code generation reduces the need to verify the appropriate inclusion of defences.

The accuracy of the characterisation and the efficacy of individual defensive structures will become the focus of reviews. Both of which are more easily measured than defences against unclear/imprecise attacks or the identification of undefended critical code.

7.2 Strategies

The techniques and workflow described here are, to a large extent, self-correcting. Using physical attacks to both develop and test a fault model avoids the pitfalls experienced in earlier work. Exhaustive testing of defences based on simulations of a potentially flawed fault model (Theißing [173] and others) gives unverifiable results; while similarly flawed assumptions about error behaviour (Moro [118]) have invalidated attempts at formal verification of defence efficacy. The ability to readily characterise a device and test defences means effort is expended on relevant defences, and errors introduced while implementing defences can be easily identified. Most importantly, the workflow demonstrated here identifies the most easily created and most numerous errors. The limited budget for defences can then be spent covering the most relevant error scenarios.

The approach is not infallible. For example, with our DUT, the assumption that skipped instructions can be modelled as Nop instructions is simplistic, and some of the errors identified in Section 6.3 are only explainable if the number of errors exceeds the number of pulses injected. The most likely cause of this would be if the presumed Nop performed a meaningful instruction and modified the μC's state. The previously noted reliable repeatability of errors, in general, would suggest they should be more common than the results suggests. However, the detailed characterisation described in Section 3 found very few examples of register corruption. Where code is liberally defended, a hunt for the rare exploitable exceptions will trigger many other traps during the search.

Handling common errors is more important than detecting comparatively rare edge cases. Two competing actions must be considered when a fault is detected and a trap is invoked.

1. The trap itself is vulnerable to errors. It would therefore be prudent to freeze operation of the μC instantly, thus preventing additional faults from negating the trap. Many of the crashes and untrapped errors witnessed in this study have been traced to faults induced in the trap recording and management code.

2. The Creeping Barrage attack described in Section 4.3 involves sweeping a pulse pattern over executing code until the pattern of induced errors negates the defence. It is likely that faults will be detected during the sweep and before the perfect alignment is found. The most practical defence against this attack is to record these early detected faults and lock the μC. It becomes advantageous to remember a history of suspected attacks, and this is most efficiently done after a fault has been trapped.

With robust mechanisms in place to identify CFI errors, the vulnerability now lies in the mechanism used to react to detected faults. The recording of an error event, after its discovery, must be performed while under attack, and failure to record an attack leaves the μC vulnerable to Creeping Barrage attacks. The alternative strategy of pre-recording a potential attack and later clearing it if it failed to materialise imposes a performance penalty on all operations and would be impractical in most situations.

For CFI defences, the simple defensive strategy of resisting two induced errors can be implemented in a compiler's code generator. Resisting two errors means an attacker needs to induce three or more perfectly timed errors. Exhaustive searching of unknown code will be time-consuming and Creeping Barrage is the most likely attack strategy.

Automated generation of such defensive code avoids accidental omission and moves the responsibility for the defence from the programmer to the tool-set. In practice, this means products that would otherwise be inadequately defended can have their defences significantly strengthened. History shows that simple products, developed on a tight budget, provide a back-door into otherwise secure environments. Deskilling the development of secure code for these devices is a realisable benefit.

Chapter 8

Conclusions and Further Work

Contents

8.1 Original Goals **250**

8.1.1 Characterization 250

8.1.2 Repeatability 252

8.1.3 Defence Refinement 254

8.1.4 Defence Automation 256

8.2 Future Research Directions **258**

8.2.1 Logistical Obstacles 258

8.2.2 Pursuable Properties 259

8.2.3 Improving the Compiler 260

8.3 The Last Word **261**

The results and discoveries of this study are reviewed in the context of the original research questions. The findings relating to a device's behaviour under error conditions are at odds with what was the presumed but generally accepted behaviour at the time the study began. These findings make it possible to generate more realistic error models, which are invaluable when prescribing effective defences.

Unfortunately, the predictability of error responses can also benefit an attacker, and this emphasises the importance of obtaining reliable device characterisation data so that the defender can ensure defences fail-safe when under attack.

Finally, as with most discoveries, new questions arise. Properties that were not, or could not, be pursued here are considered for future investigation.

This study started with the premise that simulating fault induction in μCs was unreliable and misunderstood by programmers. Consequently, software defences were likely to be untrustworthy. This view was borne out by the conflicting advice received during security reviews of coded defences and the contemporary trend of modelling error outcomes as random effects. Defence strengths were measured using software emulation of devices and monte-carlo simulations of error injection [173].

The underlying suspicion was that instruction-level simulation would fail to account for subtle differences in error response expected to exist between instructions. Hardware Description Language (HDL) based simulations could simulate errors down to a gate level. However, they could not realistically account for multiple gate errors without knowing which set of gates to simulate errors within, and that required knowledge of the relative proximity of individual transistors within the target. Manufacturers would not willingly disclose such details, and reverse engineering of Very Large-Scale Integration (VLSI) devices is prohibitively expensive. Therefore it was assumed that a detailed characterisation of error responses for each instruction would be required. This approach is the logical extension of the mechanisms that permit bit-level memory editing [55]. Ever finer focus on ever-smaller components, yielding specific errors with the expectation that an attacker could exploit these errors. This is more or less the rationale driving the interest in ISA and the desire to get a detailed understanding of the mechanisms leading to specific faults [139].

In a world where faults can be separately induced in each individual transistor within an IC, there is no limit to the range of error effects that are theoretically achievable. However, the practical obstacles are considerable and possibly insurmountable with the current generation of tools. They include identifying the appropriate set of transistors to perturb, perturbing just this set, and synchronising such perturbations with an executing program. These obstacles dictated the alternative strategy of physically inducing errors in executing instructions, collecting the device's state post-error, and then attempting to identify the error effects. This approach would search for easy to induce exploitable effects rather than trying to engineer hard to induce predicted effects.

The surprising outcome of this study has been that a straightforward fault model can describe most observable errors. In this case, instruction skipping with a NOP replacing the skipped instruction. Failure to read memory accurately, process arithmetic correctly or update memory can be modelled as skipped instructions. Branches can be misdirected by failure to set up the appropriate conditions or by skipping the instruction entirely. With such a simple fault model, instruction-level simulation is efficient and realistic. This observation contradicts many of the assumptions that initiated this investigation.

8.1 Original Goals

The original research questions that instigated this study, and the assumptions that they were based upon, are reviewed here, taking into consideration the results of the proceeding experiments. We had four basic questions at the start of this investigation.

8.1.1 Characterization

> *Do induced errors have repeatable characteristics that would assist developers in predicting a device's likely modes of failure?* (RQ1)

Where the laser is focussed changes the error response, and for large areas of the DUT's surface, we detected no error response at all. This lack of effect is not surprising as the DUT, a micro-controller, contains peripherals that were not active at the time of the experiments. The widely used practice of quickly scanning the whole chip surface identifies the sensitive areas worthy of further investigation. The specific instruction executing while the laser pulse fires is not particularly important during this first pass zone identification step. The critical test property is the ability to detect faults in the DUT's state. Checksums of memory blocks and register dumps provide enough information to recognise areas worthy of deeper investigation.

The initial expectations were that combining very precise pulse injection times and fine focus would cause specific changes in an instruction's behaviour, and different instruction classes would show distinctive failure modes. For example, making

conditional jumps unconditional or preventing the status flags from being updated. The reasoning here reflects the same logic that drives the interest in ISA modelling. It is reasonable to assume that if individual transistors can be affected (as demonstrated by Courbon [55]), individual instructions can then be made to misbehave. However, the number of possible instructions, combined with the number of possible starting states and the number of active transistors that need testing, will make this approach infeasible without detailed knowledge of both the schematic design and the physical layout of the DUT. For purely practical reasons, not all individual transistors can be investigated, or multiple closely located transistors must be hit simultaneously. Either way, the effects will no longer be finely controllable.

With a very finely focused laser, we noted that the instruction skipping effect demonstrated here in section 3.3.5.2.1 could be achieved at one location with a pulse timed for the second quarter-cycle of the CPU clock. The same effect occurred in the third quarter cycle at a nearby location. When using a larger spot size encompassing both locations, the effect occurred in both the second and third quarter cycles. This observation shows that it is unnecessary to focus on ever-smaller features or improve the pulse's temporal accuracy. High-precision, and presumably expensive equipment, is not a prerequisite to mount a viable attack on unshielded devices. This observation contrasts with the ISA driven trend for additional precision and control, showing that neither lower precision equipment, nor ignorance of the DUT's microscopic layout, are obstacles to an attacker.

For one zone in particular on the DUT, the consistency of the error responses, across the whole range of instruction types, led to the realisation that it was not the currently executing instruction that was being perturbed but the pre-fetch of the next one. The ability to exploit this is very powerful indeed. Many of the predictable properties of ISA level attacks can be achieved via instruction skipping. Preventing memory write-back, corrupting memory reads, forcing faulty arithmetic computation or neutralising jumps; all can be replicated by instruction skipping. To create an arithmetic error, it does not matter if the read, maths, or write-back fails; the result will be corrupt either way.

The purpose of *Characterisation* is to define a fault model that can then be used to prescribe a defence strategy. Such a fault model must be a compromise between the theoretically possible and the realistically achievable. Models that map specific effects to manipulating individual transistors will be too complicated to exploit economically. Furthermore, attacks based on such scenarios have near-insurmountable practical difficulties relating to intra-cycle laser realignment and synchronisation with an executing CPU. Run-time software defences are unlikely to be effective on a device that can be manipulated to this extent.

The techniques and equipment developed here enable the characterisation of μCs. Readily available sacrificial samples are used in non-debug mode, replicating the environment of a deployed device. For different μCs, the behaviour under attack may differ and lead to alternative fault models. However, the evidence of reliable repeatability of effect across differing μC architectures [148] and via different perturbation mechanisms [52] suggests the techniques can be applied generically to μCs that are available as engineering samples. For our chosen sample, we have shown that *Instruction Skipping*, modelled as the arbitrary replacement of instructions with NOP, is a realistic fault model, despite its apparent simplicity.

8.1.2 Repeatability

> *Is it practical for attackers to induce multiple errors into software executing on a μC and exploit their effects without needing access to sophisticated laboratory equipment?* (RQ2)

The answer proved to be yes; and, the unsophisticated, home constructed apparatus proved to be significantly more versatile than the expensive equipment it replaced. The characterisation exercise was carried out using an industrial YAG laser [124]. The presumption was that relatively high power and quality optics were required. In practice, however, this equipment has drawbacks that make it unusable for this investigation. Foremost it delivers its energy in a very intense short pulse, which involves a prolonged inter-pulse recharge time. This recharge time limits the pulse

repetition rate to approximately 50 Hz. Critically, as demonstrated in Table 3.10, it had been noted that focus and very high power were not limiting requirements for inducing one-off repeatable errors. What was missing was the ability to generate laser pulses at or above the rate of the DUT's execution clock.

The most difficult problems encountered were purely engineering issues. Switching relatively high currents on and off for precise short intervals caused inevitable power spikes and reverse voltages. The problem had already been solved in the LIDAR world, and we were able to repurpose existing solutions. Similarly, getting a timing signal from the circuitry close to the DUT to the laser, mounted on the microscope's eye-piece, involved using techniques common in the fast network and USB world. The resulting equipment, described in Section 4.1, enabled us to investigate all the points we were seeking to demonstrate.

Lower power for a more extended period achieves the same repeatable effects with significantly less energy input per fault. We could also repeat the categorisation experiments using low-quality optics and obtain the same error response behaviour. Therefore difficulty of access to expensive and hard to obtain tools is not a defence against semi-invasive fault attacks. The cost of equipment to attack a device has been reduced from many tens of thousands of pounds to just a couple of hundred pounds, making the limiting factor the operational cost of manpower and time. For amateur hackers in particular, time also has a relatively low cost, and reward is often measured in kudos within their peer group. Consequently, the attacker's cost to benefit ratio versus the manufacturer's reputational and financial liabilities need to be reassessed. The implication is that many more devices should now be considered vulnerable. Modest rewards do not require excessive investment.

The new equipment also enabled us to demonstrate that error effects that are repeatable when induced in isolation are also repeatable on consecutive instructions. There is no observable after-effect that modifies the effect of a second, closely timed, error injection stimulus. This behaviour had been speculated upon (Barenghi [23], Moro [118]), but exploitation has been dismissed as impractical. Our demonstration in Section 4.2, directing execution through a matrix of branch instructions to a specified

outcome, is the first, and possibly only, published example [95] of this behaviour. Here, up to four pulses, delivered at pre-determined intervals, guide the execution path to a specified end-point. Any one of the sixteen possible outcomes can be achieved with the appropriate pulse pattern.

Predictable repeatability provides a new technique for efficiently exploiting multi-pulse attacks; this Creeping Barrage is demonstrated in Section 4.3. If the structure of a vulnerable code fragment is known at the instruction level, then, in many cases, a pre-calculated pattern of pulses can be defined to make it fail, thereby neutralising simple defences. A programmer's stylistic habits, or deterministic output from compilers, mean that predictable code fragments will repeatedly occur, and therefore pulse pattern templates can be defined to break them. These pulse patterns can be exercised against relatively large code sections, and knowledge of the larger-scale structure or implementation is unnecessary. A single pass, testing at each time interval over the executing code sample, exploits multi-pulse attacks without the combinatorial explosion that occurs when testing all variations of multiple pulses. The ease with which this can be achieved was demonstrated with an attack on Speck described in Section 4.3.

8.1.3 Defence Refinement

> *Can a better understanding of a device's modes of failure be translated into improved security via targeted software countermeasures?* (RQ3)

The characterisation experiments have shown the types of failures that can be expected from a laser pulse injection attack on the DUT. The techniques are straightforward and can be utilised on any generic μC. The repeatability experiments have also confirmed that these single error effects can be sequentially combined to defeat defences that use repetition and comparison as a defence. Knowing this enables a programmer to plan accordingly. Since it is easier to skip a conditional jump instruction than to take the jump inappropriately, it is wise to place protected code at

the branch destination rather than as the fall-through behaviour to be executed after branch-not-taken. For testing arithmetic results, simple repetition of a calculation is likely to be weak. The repeatability of injected errors suggests that the second calculation would be identically corruptible and may lead to the same erroneous result. A wise programmer would modify the arithmetic of the recalculation to ensure synchronised errors do not yield synchronised erroneous results.

A replacement or supplementary question would be — How much control does the programmer have to influence the executable code?

Testing this question led to some surprising results. After testing a wide range of defences, in Section 5.2, it is apparent that many factors are at play, and the application programmer cannot directly address all of the weaknesses. Analysis of failed defences shows several prominent features that are regularly overlooked when code is reviewed during security certification.

- Optimising compilers eliminate redundancy — Double testing of states, duplicated calculations, and logically unreachable code may get eliminated from the executable image. The only reliable solution is to reduce the compiler's optimisation level and suffer the consequence of slower and bigger binary images.
- Duplicated calculations invariably rely on a single comparison — No matter how carefully the calculations are performed to prevent duplicate error injection, the two results ultimately need to be compared. Errors injected into the comparison process are the Achilles heel of most redundant-code defences.
- Induced errors fall into two categories — Abnormal control flow and Normal flow with faulty data. Hybrid defences that address both issues are significantly stronger than the sum of their parts.
- Some defences cannot be implemented in the syntax of the source code language — Ensuring protected code does not follow as the default behaviour of an untaken conditional branch is impossible to guarantee when a compiler may reorder some code blocks and inline others.
- Code that traps errors is itself vulnerable to errors — This is perhaps the

most extreme example of the above issue. Branching to a shared error handler after detecting an error frequently leads to the situation where a skipped branch-to-error results in the normal continuation of the program. Negating the proceeding defence entirely.

While code may be vulnerable and some aspects of the resulting executable binary difficult to predict, the most reassuring feature is that testing of defences is uncomplicated and inexpensive. Developers, armed with equally powerful tools as the attackers, have a significant advantage in this respect. The ability to selectively test and refine defences is a capability that has hitherto only been available to the best-equipped research laboratories and development houses. We have demonstrated a viable development process of categorisation, implementation, testing and refinement that can be adopted without significant capital expense.

8.1.4 Defence Automation

Is it practical to automate the generation of defensive measures within a μC's software development tools? (RQ4)

Many commercially deployed compilers for μCs are based on GCC, or increasingly on LLVM. They have been repeatedly improved and refined and are the product of many authors and many man-years of expert development. Therefore, the first question is, could a one-off bespoke development generate code of comparable quality? The reality is that most of the expert development effort has focussed on optimisations that negate software defences. Optimisations need to be switched off to get executable code that retains source-level defences. This necessary and unavoidable downgrading of the compiler's capabilities is frequently overlooked, and when recognised, much of the power of state-of-the-art tools is wasted.

The output from the compiler developed for this project, DCC, compares favourably with the unoptimised output from the DUT's GCC compiler. Therefore, adopting a home-grown compiler does not imply unmanageable volumes of inefficient code will result. Code with no source-level defences compiled with DCC is smaller and faster

than the necessarily unoptimised code generated by GCC from sources embellished with defences. DCC's executable is also more robust in resisting exhaustive attacks.

The most significant single benefit of having a customisable code generator is the ability to insert defensive code structures automatically and to apply them universally. Compiler generated defences for simple code structures are significantly more efficient than convoluted source code defences processed by a traditional compiler. This removes the need for the application programmer to insert redundant code manually and removes the opportunity for accidental omission. The approach provides better defences, fewer errors and accelerated development times. All of which are highly desirable features.

In many cases, such as detecting unexpected fall-throughs after skipped branch instructions, defensive code is not executed during normal operation. Therefore despite increasing the code volume, it has a negligible impact on performance. When using a traditional compiler and implementing defences within the source code, the additional tests in the defensive code are executed, resulting in performance degradation and code expansion.

These results are highly encouraging, but they are still just proof of concept and indicate greater benefits could be obtained in future. The defences implemented here are all applied at the code generator stage. As such, the same instruction combinations could be generated by the equivalent stage in one of the more sophisticated GCC or LLVM compilers. The defences for CFI do not require any source code description and should therefore be unaffected by the more aggressive optimisations performed by the industrial-strength compilers.

Improved defences, in terms of both efficacy and code efficiency, can be achieved. The low-cost laser workstation was used to identify vulnerabilities in the DUT. The vulnerabilities defined a relevant fault model, and from this, appropriate defensive code structures were proposed and tested using the same equipment. The most effective defences were then implemented within DCC to provide a DUT specific compiler. In bringing together the results relating to error categorization, defence testing, and efficient deployment, we demonstrated that this workflow is straightforward and elim-

inates many weaknesses inherent in current best-practice. In particular, it ensures the appropriate choice of defensive structures and provides confidence relating to accurate and complete defence deployment.

8.2 Future Research Directions

The work to date has identified a practical new approach to defensive coding. However, some questions remain unanswered, and some opportunities have not yet been pursued. This section identifies some of these issues and speculates on their relevance.

8.2.1 Logistical Obstacles

We, unfortunately, lost access to the wet-lab and the safe environment for sample preparation halfway through this investigation. While *Health & Safety* may not be a significant issue for attackers, it does limit the opportunities for responsible organisations. With ample supplies of our chosen target pre-prepared, we chose to concentrate on this device. Latterly, reports from other researchers have shown that similar effects have been witnessed on other architectures. The strength of the argument presented here relies on the transferability of the techniques to other targets. This appears to be the case but, as of yet, has not been fully demonstrated.

ARM would be an obvious second target, given its ubiquity in embedded applications. Riviere [148] has already demonstrated that ARM shows repeatable error characteristics and has defined an exploitable fault model for it. It may therefore be more informative to look at an alternative target.

The other issue we were unable to peruse was using a rear-side attack with a NIR laser. NIR from the rear-side has advantages because it is unobstructed by metal layers and will enable a more comprehensive characterisation of a device. The difficulties here relate to equipment setup, which is easier to overcome than we initially expected. By using a CCD, NIR light is visible on a video display, so focus and alignment should be solvable engineering problems. Standard microscope lenses are optimised for visible light, so very fine focus may be difficult to achieve. However, we

have shown that exploitable error responses do not require fine focus. An attack via the rear side may also simplify the sample preparation. The top side of a chip is very fragile, and wet etching is the only practical way to avoid damage to the device. In manufacturing, the chip is usually placed on a carrier to hold it in position while bond wires are attached between it and the legs. The whole assembly is then encapsulated in black plastic. Mechanical etching on the rear side can expose the carrier, which can then be cut and removed without damaging the chip's top surface or the bond wires. Rear side attacks may remove the unfortunate obstacle we faced mid-project; that of sample preparation.

8.2.2 Pursuable Properties

We have noted that a longer low-power pulse from a diode can substitute for the very short, very powerful YAG pulse. We have also noted that the same effect can sometimes be obtained by differently timed pulses at closely located positions on the DUT. Presumably, this is catching a propagating signal at different stages of the machine cycle. Where an effect can be induced in different areas at different times, a laser spot encompassing both areas can induce the effect at both time intervals. This leads to the question, would a beam focussed on either zone produce the same effect if the pulse duration was extended to cover both time intervals?

This result would be interesting as it may mean longer pulses would be more effective than many short pulses in triggering errors. In a clock synchronised circuit, biasing the transistor for a longer period encompassing the critical moment may not matter, and we could speculate that error induction would become even more reliable. We identified and concentrated on a zone that gave consistent results and ignored other areas that generated similar effects (akin to instruction skip) while being less reliable. If an extended pulse does increase the reliability, then more of the DUT's surface may demonstrate exploitable errors. Consequently, the first stage early scan (as seen in Figure 3-14) could be quicker to perform and would locate larger areas with reliable error responses. Again, we speculate that lower precision in both targeting and timing may prove to be readily exploitable.

Another exploitation of predictable and repeatable error effects would invoke hidden code. Stack smashing techniques aim to take control of an executing CPU by injecting executable code, disguised as data, and invoking it by corrupting a return address within the stack frame. Similarly, a developer could hide uncalled code within an embedded application and invoke it via an execution error. Bukasa [47] identified this risk and cited the unpredictability of errors as the factor that prevents exploitation of such *Fault Activated Backdoors*. Hamadouche [82] has described a mechanism to generate code that contains such a trojan and still passes byte-code verification on a Java-Card. These researchers have addressed the issue of the payload's creation and delivery while identifying reliable invocation of the trojan code as the last remaining obstacle. We have seen that this is not an obstacle; trojan code could be inserted into an application with confidence that a prescribed error injection pattern will be able to invoke it.

8.2.3 Improving the Compiler

By far, the most complicated parts of a compiler relate to the creation of the IR along with the manipulations and optimisation performed upon it. Many of these activities are independent of the source code language and the target instruction set. Code generators inherit this universal behaviour while being intimately related to the target device. This separation is ideal for exploiting the results of this investigation. Device-specific defences can be added to the only device-specific stage of the compilation process. This study has demonstrated that the code generator, independent of the IR, can generate efficient defences. Therefore, it would be logical to apply the techniques to a high-quality open-source compiler and benefit from its more sophisticated optimisations.

The compiler defences part of this study have concentrated on CFI. However, in the defences investigations, Section 5.2, we established that hybrid arithmetic and CFI defences complement each other. Detecting faulty arithmetic relies on trustable test and branch behaviour, while test and branch behaviour needs accurate data to inform its decisions. Presumably, the subtle source-level semantics must have been preserved

through the IR stage. Compound operations, such as read with post-increment, which if blindly repeated at source code level change a statement's semantics, will be sequenced within the IR's register transfer description language. Therefore, it may be practical to add redundant operations or employ shadows registers within the target to generate defended arithmetic code safely. Implementing defensive arithmetic operations within the code generator would potentially isolate the programmer from all aspects of defensive coding.

8.3 The Last Word

The low-cost and relative ease of construction of our laser error injector suggest that developers of IoT devices need to seriously consider the likelihood and consequences of an attack on their products. This study should encourage IoT developers to use defensive coding as normal practice, and in many cases, consider using devices with physical defences against this category of attack. Such a paranoid outlook is crucial because it must be assumed that these attack techniques are readily available to criminals, malicious attackers, and amateur hackers.

The low-cost equipment makes device characterisation open to all interested parties, not just the well-equipped laboratories. Characterisation demonstrates repeatable outcomes when injecting errors, and repeatable outcomes mean predictable behaviour when under attack. This predictability can be exploited by attackers who can refine techniques on readily available samples before moving to meaningful targets. Predictable error responses can also guide the defender to identify relevant defences and be confident about how those defences will respond when under attack. These defences can also be efficiently encapsulated in a compiler, thus ensuring full application-wide deployment, avoiding accidental oversight and the corresponding need for meticulous independent review. For a given device, there should no longer be uncertainty or confusion about which style of defensive code to deploy. Furthermore, the previously unresolvable debates with security evaluators that inspired this study (see page 26) can now be concluded.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] Alfarez Abdul-Rahman. The pgp trust model. In *EDI-Forum: the Journal of Electronic Commerce*, volume 10, pages 27–31, 1997.

[3] M. Agoyan, J. Dutertre, A. Mirbaha, D. Naccache, A. Ribotta, and A. Tria. How to flip a bit? In *2010 IEEE 16th International On-Line Testing Symposium*, pages 235–239, 2010.

[4] AV Aho, R Sethi, and JD Ullman. Compilers: Principles, techniques, and tools, second ed. 1985.

[5] Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. Enhancing java runtime environment for smart cards against runtime attacks. In *European Symposium on Research in Computer Security*, pages 541–560. Springer, 2015.

[6] Altera Coorporation. Introduction to single-event upsets. White paper - WP-01606-1.0, 2013. Accessed: 2021-01-06.

[7] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In *International Workshop on Security Protocols*, pages 125–136. Springer, 1997.

[8] Ross Anderson and Roger Needham. Programming satan's computer. In *Computer Science Today*, pages 426–440. Springer, 1995.

[9] AP Technologies Ltd. Photodiode theory of operation. `http://www.aptechnologies.co.uk/support/photodiodes/photodiode-theory-of-operation`, 04 2014. Accessed: 2021-03-07.

[10] AP Technologies Ltd. Simon and speck on avr. `https://github.com/openluopworld/simon_speck_on_avr`, 12 2016. Accessed: 2021-06-18.

[11] Ars Staff. How security flaws work: The buffer overflow. `https://arstechnica.com/information-technology/2015/08/how-security-flaws-work-the-buffer-overflow/`, August 2015. Accessed: 2020-12-31.

[12] Atmel Corporation. *ATtiny441/ATtiny841 Datasheet – 8-bit AVR Microcontroller with 4/8K Bytes In-System Programmable Flash - Datasheet*, 05 2014. Rev. 8495H.

[13] Atmel Corporation. *Datasheet*, 2 2014. Rev.: Atmel-2549Q.

[14] Atmel Corporation. Security for intelligent, connected iot edge nodes. White Paper, 2015. Rev.:Atmel-8994A-CryptoAuth-Security-for-Intelligent-Connected-IoT-Edge-Nodes-WhitePaper_112015.

[15] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and J-P Seifert. Fault attacks on rsa with crt: Concrete results and practical countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 260–275. Springer, 2002.

[16] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114. IEEE, 2011.

[17] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[18] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java card operand stack: fault attacks, combined attacks and countermeasures. In *International Conference on Smart Card Research and Advanced Applications*, pages 297–313. Springer, 2011.

[19] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on java card 3.0 combining fault and logical attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 148–163. Springer, 2010.

[20] Alessandro Barenghi, Guido Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. Low voltage fault attacks to aes and rsa on general purpose processors. *IACR Cryptol. ePrint Arch.*, 2010:130, 2010.

[21] Alessandro Barenghi, Guido M Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. Injection technologies for fault attacks on microprocessors. In *Fault Analysis in Cryptography*, pages 275–293. Springer, 2012.

[22] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[23] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented aes: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, pages 1–10, 2010.

[24] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.

[25] Robert C Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *IEEE Transactions on device and materials reliability*, 1(1):17–22, 2001.

[26] Robert C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316, 2005.

[27] Pierre Bayon, Lilian Bossuet, Alain Aubert, Viktor Fischer, François Poucheret, Bruno Robisson, and Philippe Maurine. Contactless electromagnetic active attack on ring oscillator based true random number generator. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 151–166. Springer, 2012.

[28] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Friedrich Beck. *Integrated circuit failure analysis: a guide to preparation techniques*. John Wiley & Sons, 1998.

[30] Robert Bellinger. Scientific imaging: Near-ir microscopes image through silicon without damaging the finished product. `https://www.laserfocusworld.com/optics/article/16548175/scientific-imaging-nearir-microscopes-image-through-silicon-without-damaging-the-finished-product`, 01 2017. Accessed: 2021-03-07.

[31] Noemie Beringuier-Boher, Marc Lacruche, David El-Baze, Jean-Max Dutertre, Jean-Baptiste Rigaud, and Philippe Maurine. Body biasing injection attacks in practice. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 49–54, 2016.

[32] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 513–525, 1997.

[33] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997.

[34] Johannes Blömer and Jean-Pierre Seifert. Fault based cryptanalysis of the advanced encryption standard (aes). In *International Conference on Financial Cryptography*, pages 162–181. Springer, 2003.

[35] Mishap Investigation Board. Mars climate orbiter mishap investigation board phase i report november 10, 1999, 1999.

[36] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[37] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 37–51, 1997.

[38] Richard Bornat. Phases and passes. In *Understanding and Writing Compilers*, pages 8–21. Macmillan International Higher Education, 1979.

[39] Jakub Breier, Xiaolu Hou, and Yang Liu. Fault attacks made easy: Differential fault analysis automation on assembly code. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 96–122, 2018.

[40] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. Laser profiling for the back-side fault attacks: with a practical laser skip instruction attack on aes. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pages 99–103, 2015.

[41] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son-Tuan Vu. Fault attack vulnerability assessment of binary code. In *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*, pages 13–18, 2019.

[42] Sébastien Briais, Jean-Michel Cioranesco, Jean-Luc Danger, Sylvain Guilley, David Naccache, and Thibault Porteboeuf. Random active shield. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 103–113. IEEE, 2012.

[43] Dennis M. Ritchie Brian W. Kernighan. *The C Programming Language, Ansi C, Second Edition*. Prentice Hall, 1988. isbn: 0-13-110362-8.

[44] Brendan Bridgford, Carl Carmichael, and Chen Wei Tseng. Single-event upset mitigation selection guide. *Xilinx Application Note, XAPP987 (v1. 0)*, 2008.

[45] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[46] Julien Brouchier, Tom Kean, Carol Marsh, and David Naccache. Temperature attacks. *IEEE Security & Privacy*, 7(2):79–82, 2009.

[47] Sebanjila K Bukasa, Ronan Lashermes, Jean-Louis Lanet, and Axel Leqay. Let's shock our iot's heart: Armv7-m under (fault) attacks. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–6, 2018.

[48] Wayne Burleson, Shane S Clark, Benjamin Ransford, and Kevin Fu. Design challenges for secure implantable medical devices. In *DAC Design Automation Conference 2012*, pages 12–17. IEEE, 2012.

[49] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[50] George K Celler and Sorin Cristoloveanu. Frontiers of silicon-on-insulator. *Journal of Applied Physics*, 93(9):4955–4978, 2003.

[51] Chien-Ning Chen and Sung-Ming Yen. Differential fault analysis on aes key schedule and some countermeasures. In *Australasian Conference on Information Security and Privacy*, pages 118–129. Springer, 2003.

[52] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–10. IEEE, 2019.

[53] Common Criteria. Common criteria for information technology security evaluation - part 1: Introduction and general model version 3.1 rev.5. `https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf`, 04 2017. Accessed: 2020-12-18.

[54] Jean-Sébasticn Coron, Paul Kocher, and David Naccache. Statistics and secret leakage. In *International Conference on Financial Cryptography*, pages 157–173. Springer, 2000.

[55] Franck Courbon, Philippe Loubet-Moundi, Jacques JA Fournier, and Assia Tria. Adjusting laser injections for fully controlled faults. In *International workshop on constructive side-channel analysis and secure design*, pages 229–242. Springer, 2014.

[56] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[57] Ronald De Keulenaer, Jonas Maebe, Koen De Bosschere, and Bjorn De Sutter. Link-time smart card code hardening. *International Journal of Information Security*, 15(2):111–130, 2016.

[58] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, Philippe Orsatelli, Philippe Maurine, and Assia Tria. Injection of transient faults using electromagnetic pulses-practical results on a cryptographic system-. *IACR Cryptol. ePrint Arch.*, 2012:123, 2012.

[59] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15. IEEE, 2012.

[60] Nachum Dershowitz and Edward M Reingold. Calendrical calculations. *Software: Practice and Experience*, 20(9):899–928, 1990.

[61] Bipin C. Desai. Iot: Imminent ownership threat. In *Proceedings of the 21st International Database Engineering & Applications Symposium*, pages 82–89. Association for Computing Machinery, 2017.

[62] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In *International conference on smart card research and advanced applications*, pages 107–124. Springer, 2015.

[63] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on aes. In *International Conference on Applied Cryptography and Network Security*, pages 293–306. Springer, 2003.

[64] Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, and Assia Triaz. Reproducible single-byte laser fault injection. In *6th Conference on Ph. D. Research in Microelectronics & Electronics*, pages 1–4. IEEE, 2010.

[65] EMVCo. LLC. Emv integrated circuit card specifications for payment systems book 2 security and key management version 4.3, 2011.

[66] EMVCo. LLC. Issuer and application security guidelines - version 2.6. `https://www.emvco.com`, 08 2018. Accessed: 2020-12-18.

[67] Luis Entrena, Mario García-Valderas, Almudena Lindoso, Marta Portela-Garcia, and Enrique San Millán. Fault injection methodologies. In *Radiation Effects on Integrated Circuits and Systems for Space Applications*, pages 127–144. Springer, 2019.

[68] Dana Ford. Cheney's defibrillator was modified to prevent hacking. *CNN News*, 2013.

[69] Peter G. Putin's fsb hacked once again: Russia's new cyber weapon aimed to spy on every device exposed! Tech Times: https://www.techtimes.com/articles/248256/20200322/putins-fsb-hacked-once-again-russias-new-cyber-weapon-aimed.htm, 3 2020. Accessed: 2021-01-15.

[70] Georges Gagnerot. *Study of attacks on embedded devices and associated countermeasures*. PhD thesis, PhD thesis. University of Limoges, 2014.

[71] Daniel Genkin, Adi Shamir, and Eran Tromer. Acoustic cryptanalysis. *Journal of Cryptology*, 30(2):392–443, 2017.

[72] Samuel Gibbs. Jeep owners urged to update their cars after hackers take remote control. The Guardian, https://www.theguardian.com/technology/2015/jul/21/jeep-owners-urged-update-car-software-hackers-remote-control, 6 2015. Accessed: 2021-01-24.

[73] Samuel Gibbs. Raspberry pi 2 is camera shy: flash causes mini-computer to switch off. The Guardian, https://www.theguardian.com/technology/2015/feb/09/raspberry-pi-2-camera-flash-power-off, 2 2015. Accessed: 2021-01-15.

[74] Christophe Giraud. Dfa on aes. In *International Conference on Advanced Encryption Standard*, pages 27–41. Springer, 2004.

[75] Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In *Smart Card Research and Advanced Applications VI*, pages 159–176. Springer, 2004.

[76] Christophe Giraud and Adrian Thillard. Piret and quisquater's dfa on aes revisited. *IACR Cryptol. ePrint Arch.*, 2010:440, 2010.

[77] GlobalPlatform Inc. Global platform card specification - version 2.2.1. https://globalplatform.org/wp-content/uploads/2018/06/GPC_Specification-2.2.1.pdf, 01 2011. Accessed: 2020-10-01.

[78] Sudhakar Govindavajhala and Andrew W Appel. Using memory errors to attack a virtual machine. In *2003 Symposium on Security and Privacy, 2003.*, pages 154–165. IEEE, 2003.

[79] James Gratchoff, Niek Timmers, Albert Spruyt, and Lukasz Chmielewski. Proving the wild jungle jump. 2015.

[80] Oscar M Guillen, Michael Gruber, and Fabrizio De Santis. Low-cost setup for localized semi-invasive optical fault injection attacks. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 207–222. Springer, 2017.

[81] Donald H Habing. The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits. *IEEE Transactions on Nuclear Science*, 12(5):91–100, 1965.

[82] Samiya Hamadouche, Jean-Louis Lanet, and Mohamed Mezghiche. Hiding a fault enabled virus through code construction. *Journal of Computer Virology and Hacking Techniques*, 16(2):103–124, 2020.

[83] Jaap-Henk Hoepman and Bart Jacobs. Increased security through open source. *Communications of the ACM*, 50(1):79–83, 2007.

[84] Matthew Hughes. Why the ikettle hack should worry you (even if you don't own one). `https://www.makeuseof.com/tag/ikettle-hack-worry-even-dont-one/`, 10 2015. Accessed: 2021-01-15.

[85] Michael Hutter and Jörn-Marc Schmidt. The temperature side-channel and heating fault attacks. volume 8419, 11 2013.

[86] iC Haus Gmbh. *Datasheet - iC-HG. 3A Laser Switch*, 2014. Rev. B2.

[87] Infineon Technologies AG. *Security Guidelines. M7820 Controller Family for Secure Application.*, 06 2015. Revision: 2025-06-19 - Under NDA.

[88] Infineon Technologies AG. *Integrity Guard*, 06 2018. Order No. B181-I0677-V1-7600-EU-EC.

[89] INSIDE Secure, Aix-en-Provence, Farance. *Security Recommendations for* 0.13 µm *products - 2*, 03 2012. Ref: TPR0456EX - Under NDA.

[90] ISO/IEC. *Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher*, 2011. Edition 2.

[91] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, pages 13–26. Citeseer, 1997.

[92] Burton S Kaliski Jr and Matthew JB Robshaw. Comments on some new attacks on cryptographic devices. *RSA Laboratories Bulletin*, 5:1–5, 1997.

[93] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.

[94] Mark Karpovsky, Konrad J Kulikowski, and Alexander Taubin. Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard. In *International Conference on Dependable Systems and Networks, 2004*, pages 93–101. IEEE, 2004.

[95] Martin S Kelly and Keith Mayes. High precision laser fault injection using low-cost components. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 219–228. IEEE, 2020.

[96] Martin S Kelly, Keith Mayes, and John F Walker. Characterising a cpu fault attack model via run-time data analysis. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 79–84. IEEE, 2017.

[97] Auguste Kerckhoffs. *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef*. Librairie militaire de L. Baudoin, 1883.

[98] Swati Khandelwal. Hackers take remote control of tesla's brakes and door locks from 12 miles away. The Guardian, `https://thehackernews.com/2016/09/hack-tesla-autopilot.html`, 9 2016. Accessed: 2021-01-24.

[99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[100] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[101] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2014.

[102] Michael Lackner, Reinhard Berlach, Michael Hraschan, Reinhold Weiss, and Christian Steger. A defensive java card virtual machine to thwart fault attacks by microarchitectural support. In *2013 International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pages 1–8. IEEE, 2013.

[103] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card c codes. In *European Symposium on Research in Computer Security*, pages 200–218. Springer, 2014.

[104] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A taxonomy of computer program security flaws, with examples. Technical report, NAVAL RESEARCH LAB WASHINGTON DC, 1993.

[105] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[106] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. On the importance of analysing microarchitecture for accurate software fault models. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 561–564. IEEE, 2018.

[107] Arjen K Lenstra. Memo on rsa signature generation in the presence of faults. Technical report, 1996.

[108] Dindayal Mahto and Dilip Kumar Yadav. Rsa and ecc: a comparative analysis. *International journal of applied engineering research*, 12(19):9053–9061, 2017.

[109] Tal G Malkin, François-Xavier Standaert, and Moti Yung. A comparative cost/security analysis of fault attack countermeasures. In *International Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 159–172. Springer, 2006.

[110] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[111] MasterCard Worldwide. *Security Guidelines for M/Chip Advance Developers*, 5 March 2010.

[112] Adam Matthews. Low cost attacks on smart cards: the electromagnetic sidechannel. *Next Generation Security Software, Sept*, 2006.

[113] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[114] Trend Micro. Into the battlefield: A security guide to iot botnets. `https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/into-the-battlefield-a-security-guide-to-iot-botnets`, 12 2019. Accessed: 2021-01-15.

[115] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005.

[116] Michael G Monnett. Media predictions in operations desert shield/desert storm. Technical report, ARMY COMMAND AND GENERAL STAFF COLL FORT LEAVENWORTH KS, 1992.

[117] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[118] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.

[119] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 58–75, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[120] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 58–75. Springer, 2012.

[121] Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes. Precise instruction-level side channel profiling of embedded processors. In *International conference on information security practice and experience*, pages 129–143. Springer, 2014.

[122] NASA, Jet Propulsion Laboratory. Mariner1. `https://www.jpl.nasa.gov/missions/mariner-1/`. Accessed: 2020-12-31.

[123] NASA, Space Science Data Coordinated Archive. Mariner1. `https://nssdc.gsfc.nasa.gov/nmc/spacecraft/display.action?id=MARIN1`. Accessed: 2020-12-31.

[124] NewWave Research. *QuikLaze-50ST Operator's Manual*, 1 2005.

[125] Nichia Corporation, Tokushima 77-8601, Japan. *Datasheet - Specifications for Nichia BULE laser diode bank.* NUBM08, UTZ-SF0119E.

[126] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 914–926, 2015.

[127] Karsten Nohl, David Evans, Starbug Starbug, and Henryk Plötz. Reverse-engineering a cryptographic rfid tag. In *USENIX security symposium*, volume 28, 2008.

[128] Numato Systems, LLC. *Datasheet - Mimas - Spartan 6 FPGA Development Board.*, February accessed 2020-01-12. Mimas - Spartan 6 FPGA Development Board, https://numato.com/docs/mimas-spartan-6-fpga-development-board/.

[129] Rachid Omarouayache, Jérémy Raoult, Sylvie Jarrix, Laurent Chusseau, and Philippe Maurine. Magnetic microprobe design for em fault attack. In *2013 International Symposium on Electromagnetic Compatibility*, pages 949–954. IEEE, 2013.

[130] Oracle Corporation. Java card classic edition 3.0.5. `https://docs.oracle.com/javacard/3.0.5/`, 2015. Accessed: 2020-10-01.

[131] OSI Laser Diode Inc. *Telecom / Datacom Laser Modules*, 07 2012. https://www.laserdiode.com/.

[132] Martin Otto. *Fault attacks and countermeasures.* PhD thesis, Citeseer, 2005.

[133] Sikhar Patranabis, Abhishek Chakraborty, Phuong Ha Nguyen, and Debdeep Mukhopadhyay. A biased fault attack on the time redundancy countermeasure for aes. In *International workshop on constructive side-channel analysis and secure design*, pages 189–203. Springer, 2015.

[134] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy. In *International Conference on Selected Areas in Cryptography*, pages 231–244. Springer, 2016.

[135] Darren Pauli. Brit-american hacker duo throws pwns on iot bbqs, grills open admin). `https://www.theregister.com/2015/12/10/american_hacker_duo_throws_pwns_on_iot_bbqs_grills_open_admin/`, 12 2015. Accessed: 2021-01-15.

[136] Radia Perlman. An overview of pki trust models. *IEEE network*, 13(6):38–43, 1999.

[137] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In *International workshop on cryptographic hardware and embedded systems*, pages 77–88. Springer, 2003.

[138] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. Compiler-assisted loop hardening against fault attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4):1–25, 2017.

[139] Julien Proy, Karine Heydemann, Fabien Majéric, Albert Cohen, and Alexandre Berzati. Studying em pulse effects on superscalar microarchitectures at isa level. *arXiv preprint arXiv:1903.02623*, 2019.

[140] FIPS Pub. Data encryption standard. nist fips 46-3. *FIPS PUB*, pages 46–3, 1999.

[141] Heather M Quinn, Dolores A Black, William H Robinson, and Stephen P Buchner. Fault simulation and emulation tools to augment radiation-hardness assurance testing. *IEEE Transactions on Nuclear Science*, 60(3):2119–2142, 2013.

[142] Kevin Quinn. Ever had problems rounding off figures. *This stock exchange has. The Wall Street Journal*, page 37, 1983.

[143] Hooman Rashtian. *On the use of body biasing to improve the performance of CMOS RF front-end building blocks.* PhD thesis, University of British Columbia, 2013.

[144] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. Swift: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254. IEEE, 2005.

[145] Nozomi-Networks: Research Report. Ot/iot security report: Rising iot botnets and shifting ransomware escalate enterprise risk. https://www.nozominetworks.com/downloads/US/Nozomi-Networks-OT-IoT-Security-Report-2020-1H.pdf, 2020. Accessed: 2021-01-15.

[146] Defense World Staff Reporter. Russian federal security service seeks iot intrusion devices: Hackers group. Defense World: https://www.defenseworld.net/news/26558/Russian_Federal_Security_Service_Seeks_IoT_Intrusion_Devices__Hackers_Group#.YA3o3eieTAQ, 3 2020. Accessed: 2021-01-25.

[147] Ronald L Rivest, Adi Shamir, and Len Adleman. On digital signatures and public-key cryptosystems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1977.

[148] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015.

[149] Teri Robinson. Fsb contractor breach exposes secret cyber weapons program leveraging iot vulnerabilities. SC Magazine: https://www.scmagazine.com/home/security-news/fsb-contractor-breach-exposes-secret-cyber-weapons-program-leveraging-iot-vulnerabilities/, 3 2020. Accessed: 2021-01-25.

[150] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. Iot goes nuclear: Creating a zigbee chain reaction. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 195–212. IEEE, 2017.

[151] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault model analysis of laser-induced faults in sram memory cells. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 89–98. IEEE, 2013.

[152] Samsung Electronics Co., Ltd. *Security Application Note - S3CC91A*, 06 2015. Revision: 2007-04-27 - Under NDA.

[153] Sarah Schafer. With capital in panic, pizza deliveries soar. https://www.washingtonpost.com/wp-srv/politics/special/clinton/stories/pizza121998.htm, 12 1998. Accessed: 2021-01-24.

[154] Falk Schellenberg, Markus Finkeldey, Nils Gerhardt, Martin Hofmann, Amir Moradi, and Christof Paar. Large laser spots and fault sensitivity analysis. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 203–208. IEEE, 2016.

[155] Jörn-Marc Schmidt and Christoph Herbst. A practical fault attack on square and multiply. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 53–58. IEEE, 2008.

[156] Jörn-Marc Schmidt and Michael Hutter. *Optical and em fault-attacks on crt-based rsa: Concrete results*. na, 2007.

[157] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[158] Martin Schobert. Degate project website. Online: `https://www.degate.org/`, 9 2020. Accessed: 2021-02-16.

[159] Bodo Selmke, Johann Heyszl, and Georg Sigl. Attack on a dfa protected aes by simultaneous laser fault injections. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 36–46. IEEE, 2016.

[160] Adi Shamir. A polynomial time algorithm for breaking the basic merkle-hellman cryptosystem. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 145–152. IEEE, 1982.

[161] Fred R Shapiro. Etymology of the computer bug: History and folklore. *American Speech*, 62(4):376–378, 1987.

[162] Silvo Inc. Semiconductor process and device simulation. Web, `https://silvaco.com/tcad/`, 7 2021. Accessed: 2021-04-06.

[163] Sergei Skorobogatov. Optical fault masking attacks. In *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 23–29. IEEE, 2010.

[164] Sergei Skorobogatov. Fault attacks on secure chips. *Design and Security of Cryptographic Algorithms and Devices*, 2011.

[165] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 2–12, 2002.

[166] Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002.

[167] Eugene H Spafford. The internet worm program: An analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–57, 1989.

[168] BBC staff reporter. Raspberry pi 2 is 'camera shy'. `https://www.bbc.co.uk/news/technology-31294745`, 2 2015. Accessed: 2021-01-15.

[169] Richard M Stallman et al. *Using and porting the GNU compiler collection*, volume 86. Free Software Foundation, 1999.

[170] Takeshi Sugawara, Daisuke Suzuki, and Toshihiro Katashita. Circuit simulation for fault sensitivity analysis and its application to cryptographic lsi. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 16–23. IEEE, 2012.

[171] K Tan Tan, SH Tan, and SH Ong. Functional failure analysis on analog device by optical beam induced current technique. In *Proceedings of the 1997 6th International Symposium on the Physical and Failure Analysis of Integrated Circuits*, pages 296–301. IEEE, 1997.

[172] The VergeArs Staff. News corp allegedly hacked uk pay-tv competitor out of business. `https://www.theverge.com/2012/3/26/2904197/news-corp-nds-funded-tv-hacking-piracy-thoic-allegations`, March 2012. Accessed: 2021-09-26.

[173] Nikolaus Theißing, Dominik Merli, Michael Smola, Frederic Stumpf, and Georg Sigl. Comprehensive analysis of software countermeasures against fault attacks. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 404–409, 2013.

[174] Andrew Tierney. Embedded security fails in ics. `https://www.pentestpartners.com/security-blog/embedded-security-fails-in-ics/`, 6 2020. Accessed: 2021-01-15.

[175] Karim Tobich, Philippe Maurine, Pierre Yvan Liardet, and Thomas Ordas. Yet another fault injection technique: by forward body biasing injection. In *YACC'2012: Yet Another Conference on Cryptography*, 2012.

[176] Elena Trichina and Roman Korkikyan. Multi fault laser attacks on protected crt-rsa. In *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 75–86. IEEE, 2010.

[177] Michael Tunstall. Attacks on smart cards, 2005.

[178] J. G. J. van Woudenberg, M. F. Witteman, and F. Menarini. Practical optical fault injection on secure microcontrollers. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 91–99, 2011.

[179] Nidish Vashistha, M Tanjidur Rahman, Olivia P Paradis, and Navid Asadizanjani. Is backside the new backdoor in modern socs? In *2019 IEEE International Test Conference (ITC)*, pages 1–10. IEEE, 2019.

[180] Raoul Velazco, Dale McMorrow, and Jaime Estela. *Radiation Effects on Integrated Circuits and Systems for Space Applications.* Springer, 2019.

[181] James Vincent. Researchers hack cars to remotely control steering and brakes. The Guardian, `https://www.independent.co.uk/life-style/gadgets-and-tech/researchers-hack-cars-remotely-control-steering-and-brakes-8733723.html`, 7 2013. Accessed: 2021-01-24.

[182] Visa Inc. *Visa Security Guidelines - Multi-application Platforms*, March 2009.

[183] Colin D Walter and Susan Thompson. Distinguishing exponent digits by observing modular subtractions. In *Cryptographers' Track at the RSA Conference*, pages 192–207. Springer, 2001.

[184] Steve H Weingart. Physical security devices for computer subsystems: A survey of attacks and defences. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 302–317. Springer, 2000.

[185] Xilinx. *Spartan-6 Family Overview*, 10 2011. v2.0.

[186] Xilinx Inc. *Datasheet - Spartan-6 Family Overview.*, October 2011. DS160 (v2.0).

[187] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software fault resistance is futile: Effective single-glitch attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. IEEE, 2016.

[188] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software: Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2(2):111–130, 2018.

[189] Michael Zhivich and Robert K Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

Appendix A

Test Harness

Contents

A.1 Components **280**

A.2 Roles and Responsibilities **282**

A.2.1 Sample Alignment 282

A.2.2 Zone Identification 283

A.2.3 Executing a Test Campaign 284

A vital component of this investigation was the ability to automate the collection of experimental data. The tool described here was developed to control the laser targeting and run individual experiments. It proved to be a reliable and flexible utility that was instrumental in collecting the data used in this study.

The common requirement in nearly all of the experiments was to accurately focus the laser on the DUT's surface and move the DUT precisely to target multiple locations in a repeatable way. The solution adopted here is shown schematically in Figure A-1.

A.1 Components

- **PC workstation** — Coordinates all of the other components and collects the results of each experiment. Because it is responsible for selecting all the parameters for the experiment, it can record the results and other relevant parameters without operator intervention.

- **Stage Controller** — The PC workstation controls this serially attached peripheral. The unit enables the PC workstation to move the controller board

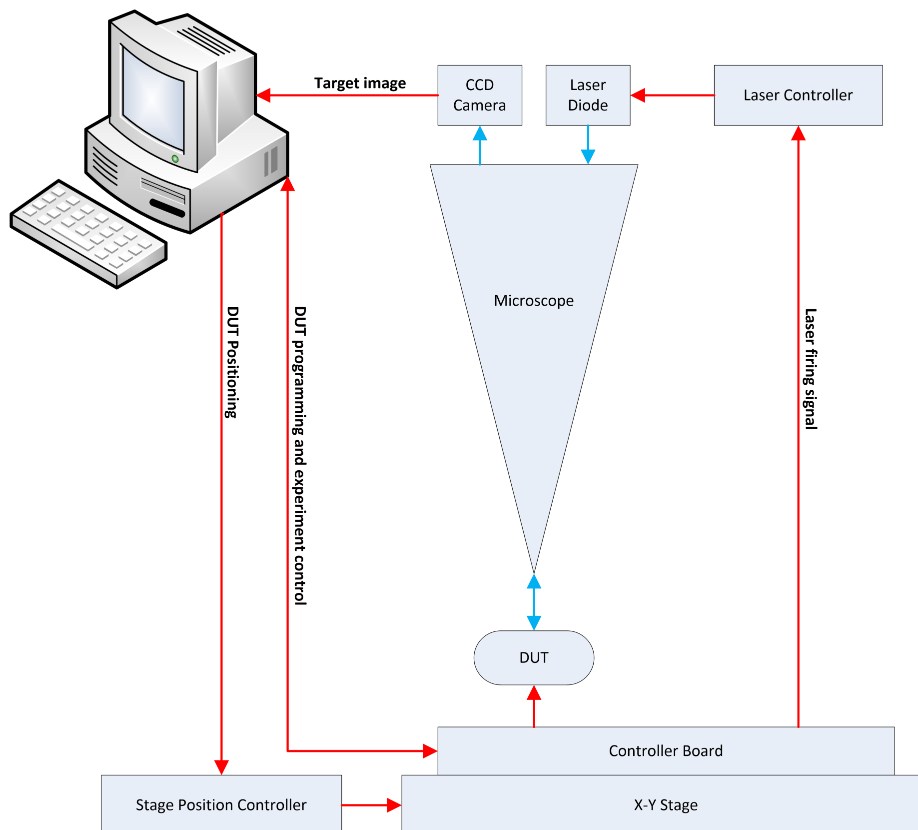


Figure A-1: Test Rig Schematic

and DUT assembly to precise positions under the microscope's objective lens.

- **X-Y Stage** — A replacement for the microscope's stage that can be electronically directed to move the DUT by the Stage Controller.

- **Controller board** — Multi-purpose, custom built, circuit board. Through a connection to the PC workstation, it can re-program the DUT and deliver control data to programs executing on the DUT. It also receives signals from the DUT that it directs to the Laser Controller and forwards experimental results to the PC workstation.

- **Laser Controller** — This component performs the fast switching of high currents needed to generate the required laser pulses. It is located as close as possible to the Laser Diode to minimise any distortion of the power signals.

- **Laser Diode** — Located on the microscope's camera port, it shines directly through the objective lens onto the DUT.

- **CCD Camera** — Located in one of the microscope's eyepieces. It delivers an image of the DUT to the PC workstation.

A.2 Roles and Responsibilities

The controller program's role is to coordinate the actions of the various components.

A.2.1 Sample Alignment

A critical aspect in all fault injection experiments is the accurate alignment of the DUT and the error stimulus. This was achieved by using a reference image of the DUT juxtaposed with the CCD image of the device in situ on the microscope stage. As seen in Figure A-2.

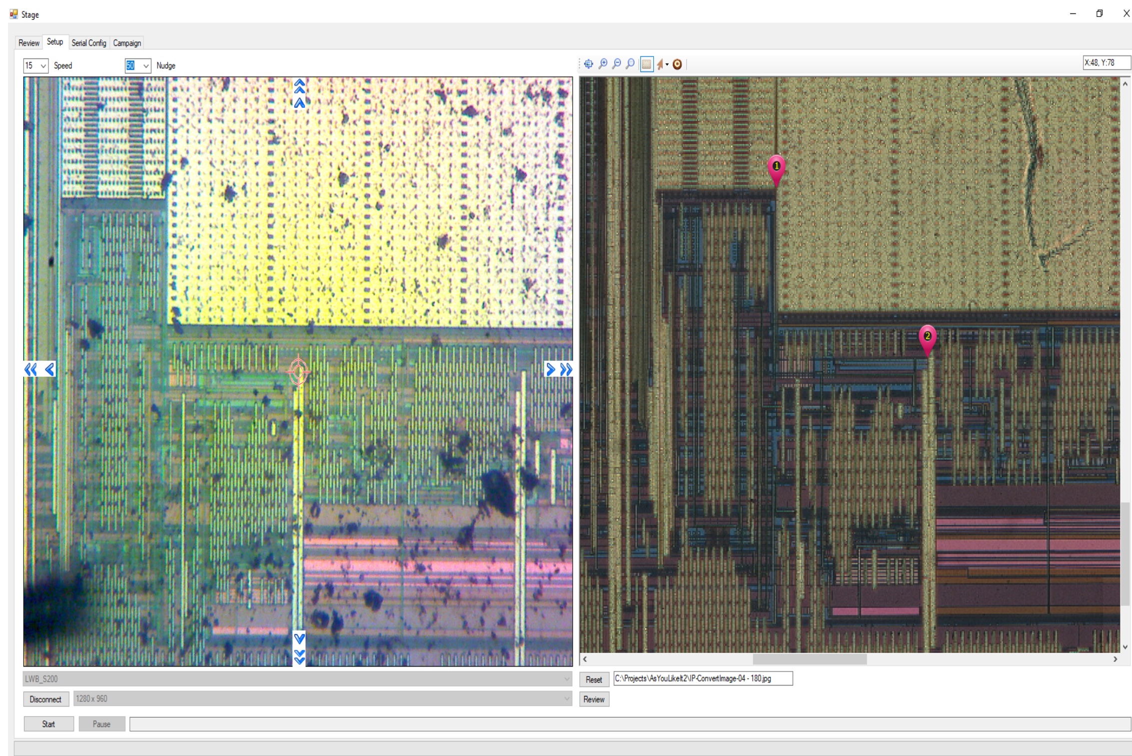


Figure A-2: Sample Alignment

The configuration has two coordinate systems. One defines the position of the XY-Stage, and the other defines the pixel position within the reference image. Alignment of the two images is achieved by moving the stage to align the cross-hair in the CCD image with a recognisable landmark on the DUT, and marking the same feature on the reference image. Both coordinate systems have an aspect ratio of 1 : 1; thus, a unit of travel on the X-axis represents the same distance on the Y-axis. These two

cross-referenced landmarks are sufficient to map between the coordinate systems and account for any rotation of the DUT*. All experiments, on multiple samples of the DUT, can then be defined in terms of the coordinate system of their shared reference image.

A.2.2 Zone Identification

Except for the course-scaled preliminary scans of the whole chip, most experiments are performed at many closely positioned sites within a confined region on the DUT. The area to be scanned can be indicated on the reference image, as shown here in Figure A-3.

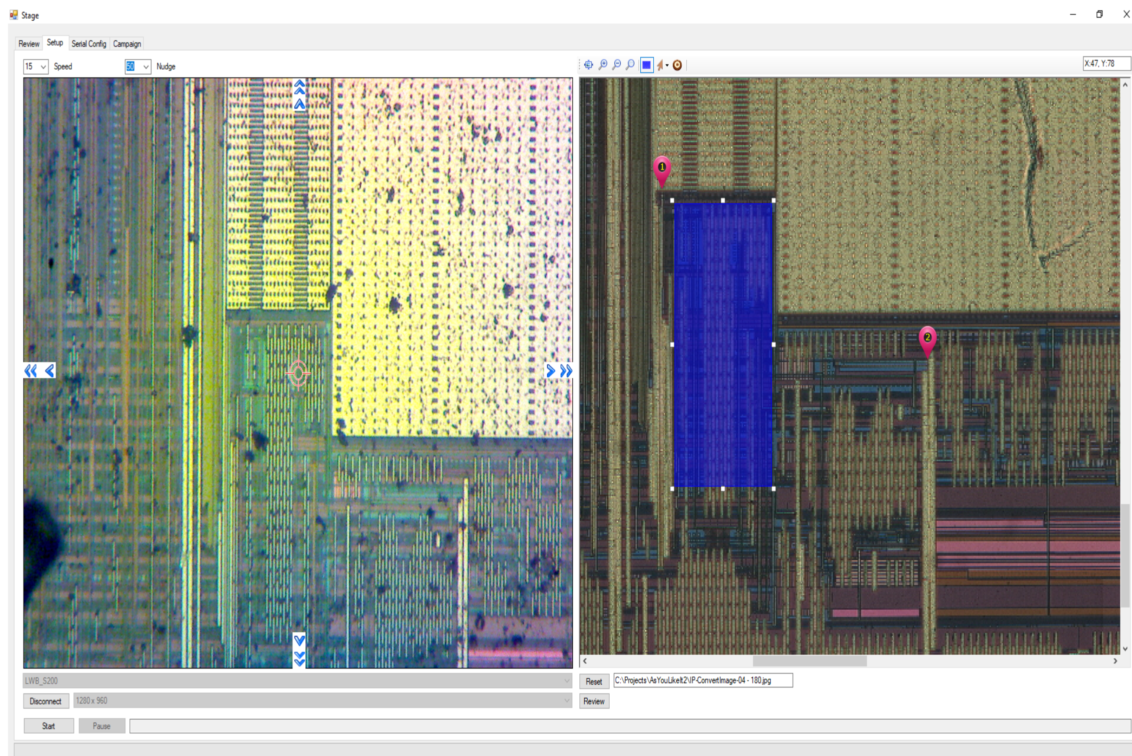


Figure A-3: Zone Identification

The test harness then performs the indicated tests only on sites within the prescribed zone. This simple feature saves a lot of time by avoiding the need to scan the whole chip or having to align many individual tests at specific sites manually.

*The hand-built nature of the controller boards means the DUTs are seldom aligned perfectly, and the microscopic size of the features of interest mean minor rotations are significant when scanning the DUT.

A.2.3 Executing a Test Campaign

A test campaign involves repeating a common test at many sites on the DUT. A campaign involves running a specific test program at all relevant locations within the *Prescribed Zone*. The test program is a separate executable that is invoked by the test harness. This separation of responsibilities means the test harness remains unmodified for many test scenarios without restricting the actions within a test.

A typical campaign's parameters are shown here in Figure A-4. In this example, the reference image is divided into a grid of 100×100 squares. The program defined by `"Test.Bat"` is executed once on each square in that grid coinciding with the *Prescribed Zone*.

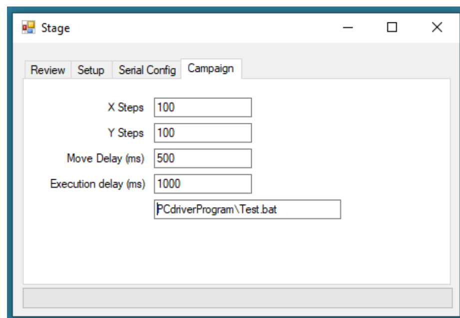


Figure A-4: Campaign Parameters

It was necessary to add delays between moving the stage and executing the test program. The 'Move delay' was added when the stage movement involved both X and Y-axis changes. Sending the Y-axis change before the X-axis move had been completed resulted in the remainder of the X move being abandoned; subsequent experiments were then marginally misaligned. The delay ensures this does not happen.

The 'Execution delay' injects a pause before the invocation of the test program. This delay was initially required because the stage swayed after each move. Latterly, with the improved mounting of the stage onto the microscope, this delay became unnecessary.

A typical test script is shown here in Figure A-5. The script file sets parameters specific to the test program before executing it; `zap3.exe` in this example. The Test Harness also indicates the test coordinates (See line 7), which can be recorded

alongside the test results.

```
1  echo off
2  set COMMS = -port=6 -baud=38400
3  set RANGE = -delay=1 -phase=0 -samples=32 -step=1
4  set COUNT = -repeat=10
5  set OUTPUT= -file=Test.Txt
6
7  set CTRL  = -v -dx=%1 -xmax=%2 -dy=%3 -ymax=%4
8  zap3.exe %OUTPUT% %RANGE% %COUNT% %COMMS% %CTRL%
```

Figure A-5: Test Script

Finally, the progress of the campaign can be monitored in the test harness as the current test site is indicated on the reference image, as shown in Figure A-6

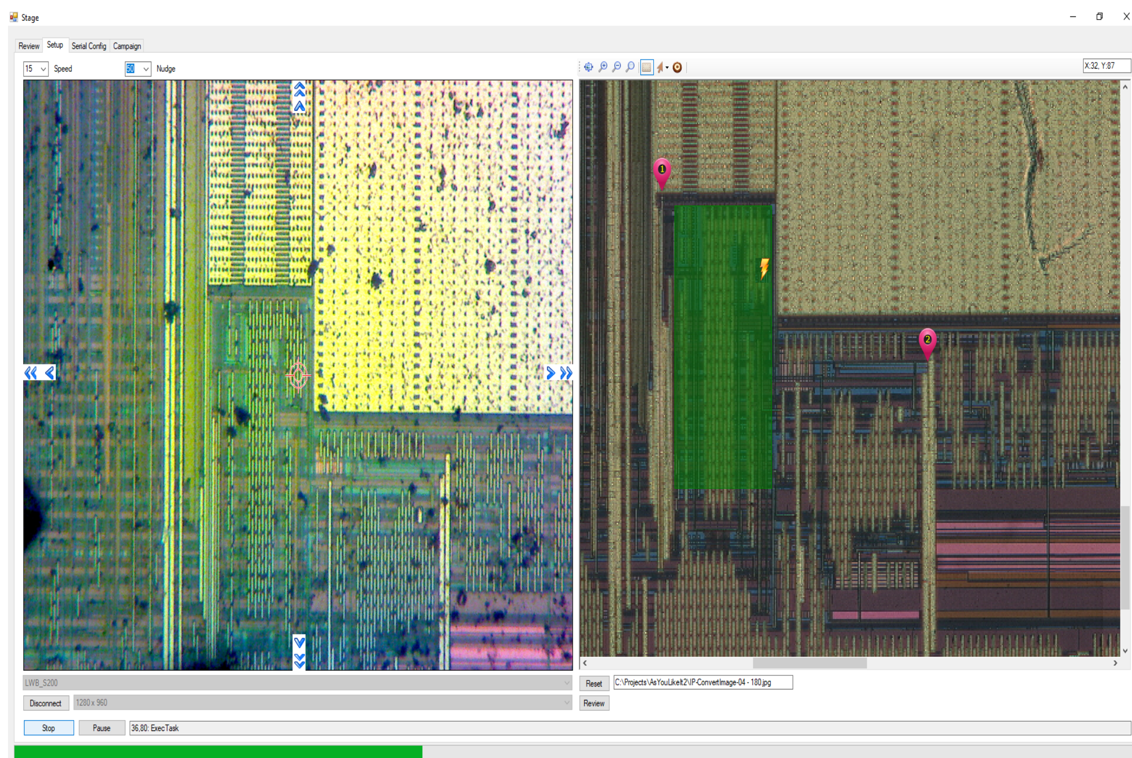


Figure A-6: Campaign Progress

The format of a campaign's results depends on the behaviour of the specific test program. In most instances, each invocation of a test program appends a line of comma-separated values (CSV) to a shared results file. This universal file format, CSV, simplifies the task of importing the data into various analysis tools.

Appendix B

Test Circuit Boards

Contents

B.1 Purpose **288**

B.2 Board 1 . . . . **288**

B.3 Board 2 . . . . **291**

B.4 Board FPGA **294**

B.4.1 FPGA program 299

During this study, a series of circuit boards were specified and built to control the DUT and laser apparatus. As understanding of the DUT's behaviour advanced, so too did the complexity of the board's role. Ultimately the design evolved into a simple and highly flexible reconfigurable device. The features of these boards, their roles, and the weaknesses that lead to their replacement are described here.

B.1 Purpose

The controller boards served four purposes.

1. They provided the electrical signals, Power, Ground, Clock, ..., to run the DUT.

2. They enabled the DUT to be reprogrammed in-situ. In-situ reprogramming meant the DUT did not need to be moved between experiments, and consequently, time was saved by avoiding the need to realign the DUT and laser between experiments.

3. They enabled the DUT to communicate with the host PC. This capability enabled the host to deliver experiment parameters and for the DUT to report results back to the host. Automating data capture is crucial to the experimental strategy adopted here.

4. Perhaps the most critical function is to synchronise the execution of software within the DUT with the trigger for firing a laser pulse. Accurate synchronisation gave certainty to the provenance of the data, avoiding the need to aggregate and average many samples.

B.2 Board 1

This board provides the necessary logic signals to drive the YAG laser. The YAG requires one signal to drive a lamp that primes the laser's crystal and another signal that activates the Q-switch. This timing and control are coordinated by the chip IC1/CONTROL, seen in Figure B-2. IC1 manages the 'LAMP' signal and 'READY' signals. The latter initiates a delay mechanism that ultimately fires the laser. The laser control signals are delivered via co-axial cables seen on the left of Figure B-1.

The READY signal feeds into the DUT IC6/SLAVE to initiate the execution of experimental code and serves to enable a bank of shift registers. A pulse then ripples

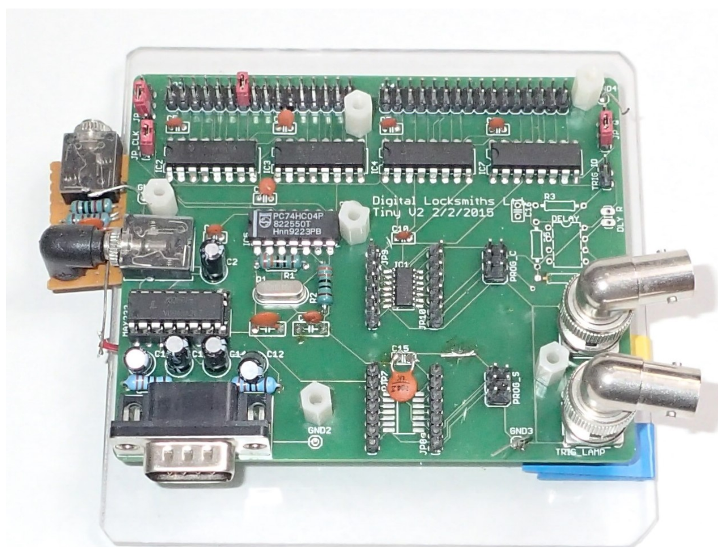


Figure B-1: Board 1 Populated

through the shifters driven by the same clock as the DUT. A jumper, placed on any one of the 32 outputs of the shifter, determines the delay between the initiation of the test program and the moment the laser fires. Another Jumper 'JP CLK' makes the shifter run on either the rising or falling edge of the 'CLK', giving the opportunity to fire the laser at half cycle intervals.

This board was well suited for the initial experiments that characterised the behaviour of specific instructions.

The weaknesses that led to its replacement were, *i*) all of the components shared the same power source, *ii*) the delay was manually selected, and, *iii*) the delay's resolution was only 50% if the CPU clock cycle. The small attachment seen on the left of Figure B-1 is a fix to allow the whole board to be reset by the controller PC.

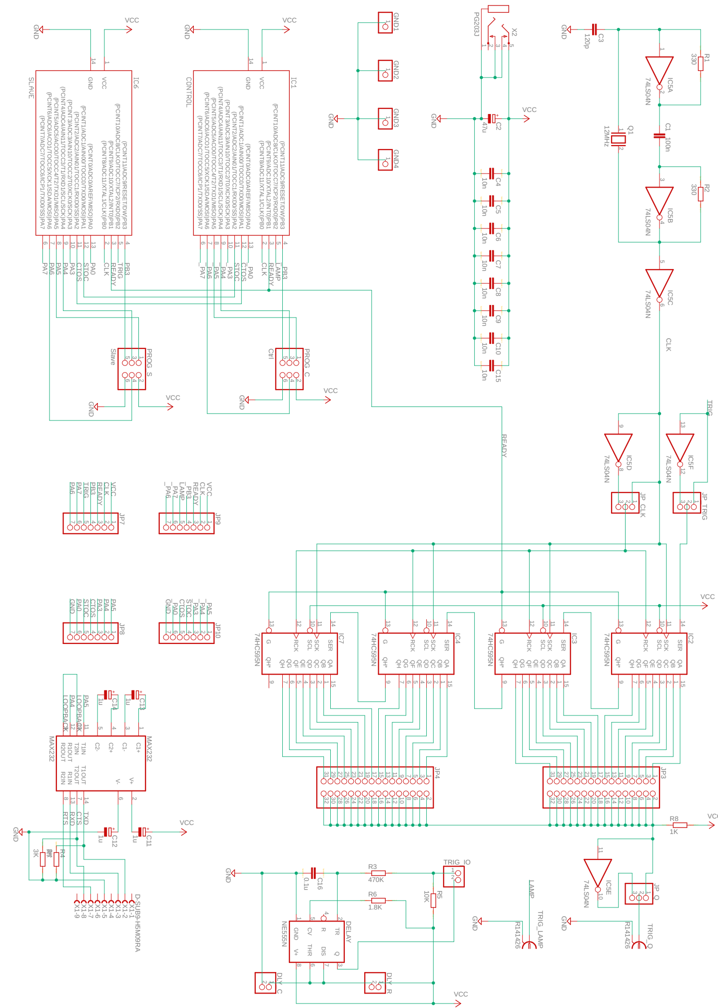


Figure B-2: Board 1 Schematic

B.3 Board 2

This board addressed the issues discovered when using Board 1. The delay between the 'Ready' signal and the laser Q-Switch triggering is managed by a programmable timer. This timer has a quarter cycle resolution and can be set from the control PC, removing the need for physical interaction with the board. See the 'SYS_CLK' & 'CTR_CLK' signals in Figure B-4. The DUT 'TARGET' has an independent power source that is separately controllable by the host PC.

The soft-programmable delay meant this board was well suited for testing the effects of pulses at different times within an executing program. This capability greatly simplified the exploitation of the results from the earlier static characterisation exercise.

An inconvenience of this board is that the laser trigger 'T_TRIG_Q' is fed directly to the DUT without additional delay. 'T_TRIG_Q' initiates an interrupt in the DUT causing it to report its status to the host PC. Interrupt latency in the DUT means the state reporting interrupt service routine is invoked four cycles after the laser pulse. This delay is insufficient for experiments that require multiple instructions to execute after the laser has fired. This minor inconvenience was resolved by adding a further delay between the laser trigger and the DUT. It can be seen in small circuit attached below the main PCB in Figure B-3.

This board and its predecessor were sufficient for categorising and exploiting the effects of single errors. The YAG laser is only capable of generating single pulses, and devices complemented each others' capabilities.

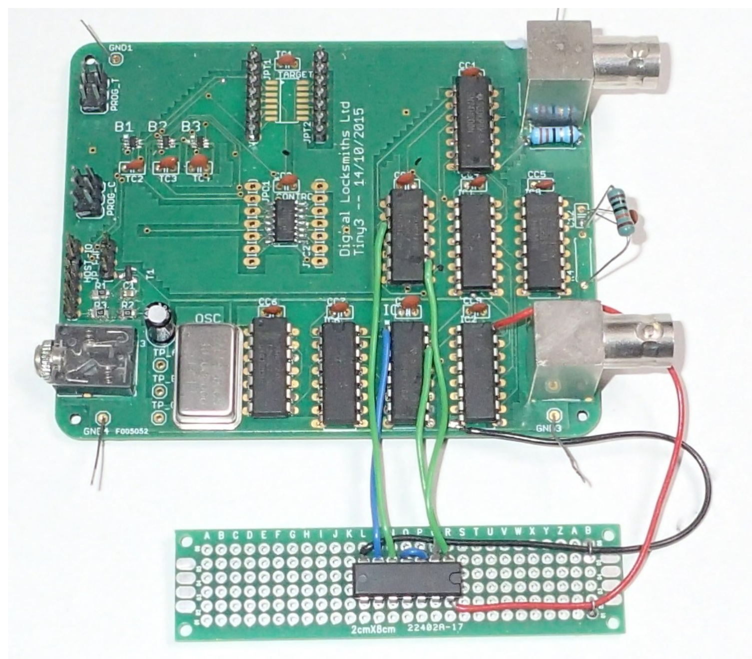


Figure B-3: Board 2 Populated

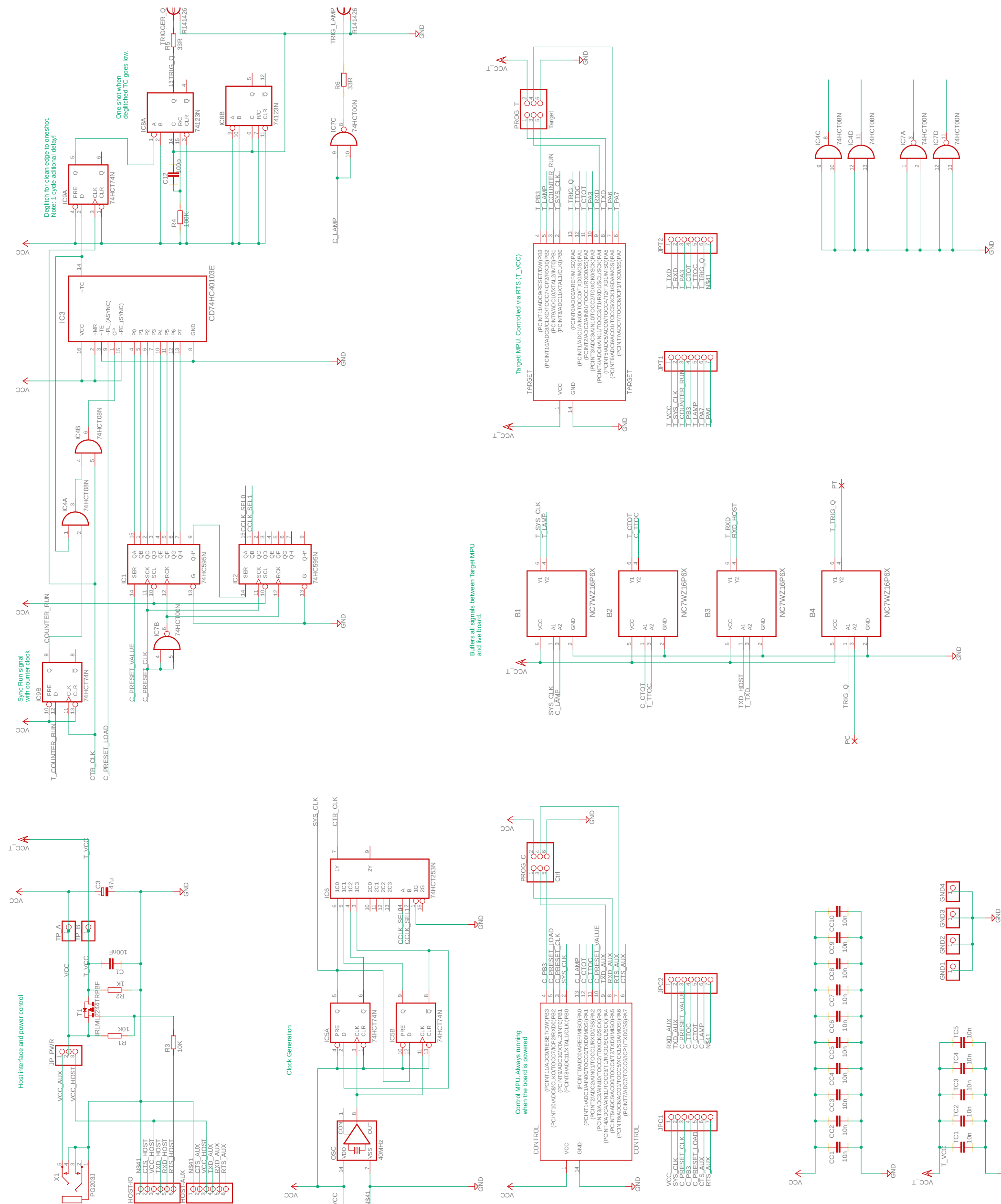


Figure B-4: Board 2 Schematic

B.4 Board FPGA

The last board in the series was developed to support multiple laser pulses. Unlike the YAG laser, laser diodes can be repeatedly fired, and timing multiple such events was beyond the capability of the earlier boards. Here a Spartan6 [185] FPGA evaluation board [128] is mounted on the underside of the main controller board's PCB, see Figure B-6. This FPGA was then programmed to provide all of the signals required to support the DUT.

As before, the DUT was independently powered to enable it to be fully reset between experiments. This power isolation necessitated buffering between the FPGA and the DUT. The configuration for each of the DUT's pins, as input or output, was managed by jumper settings seen on either side of the DUT in Figure B-5. The DUT communicates with the host PC via a serial link in the same way as the earlier boards.

The laser control signal is sent via a standard CAT6 ethernet RJ45 socket directly from the FPGA. This signalling mechanism allows the laser's power control to be co-located with the diode, ensuring the minimum of signal distortion between the FPGA and the diode itself.

The controller board was also made to fit the glass plate receptacle on the microscope's stage. This arrangement simplified the mounting of the controller board under the microscope's objective lens.

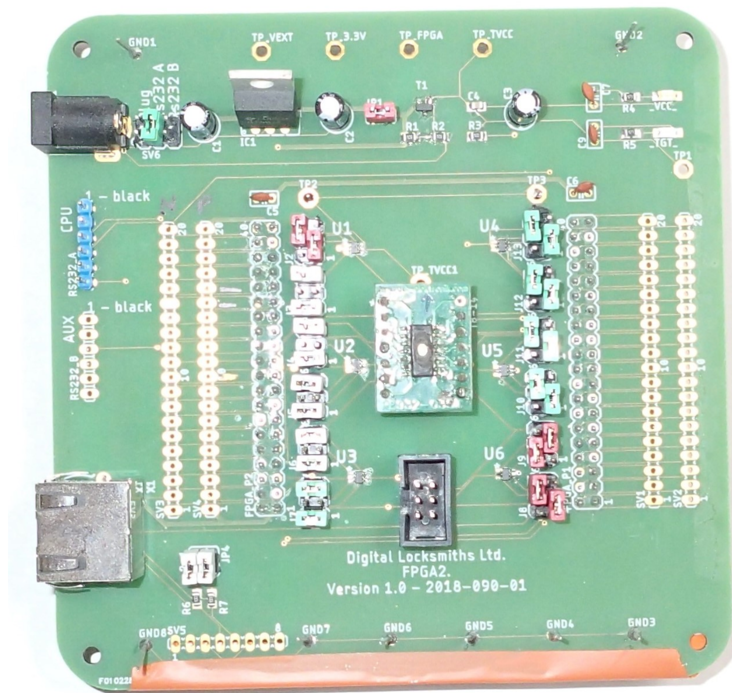


Figure B-5: Board FPGA

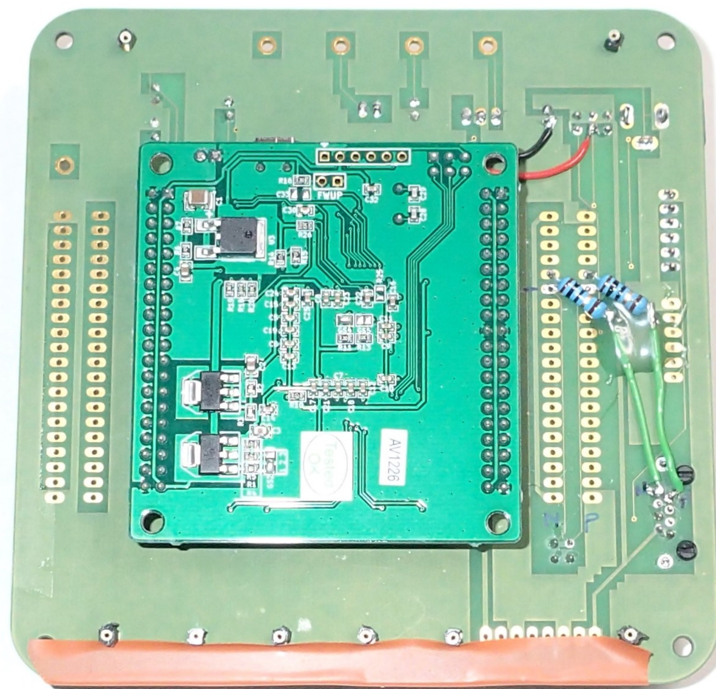


Figure B-6: Board FPGA Rear

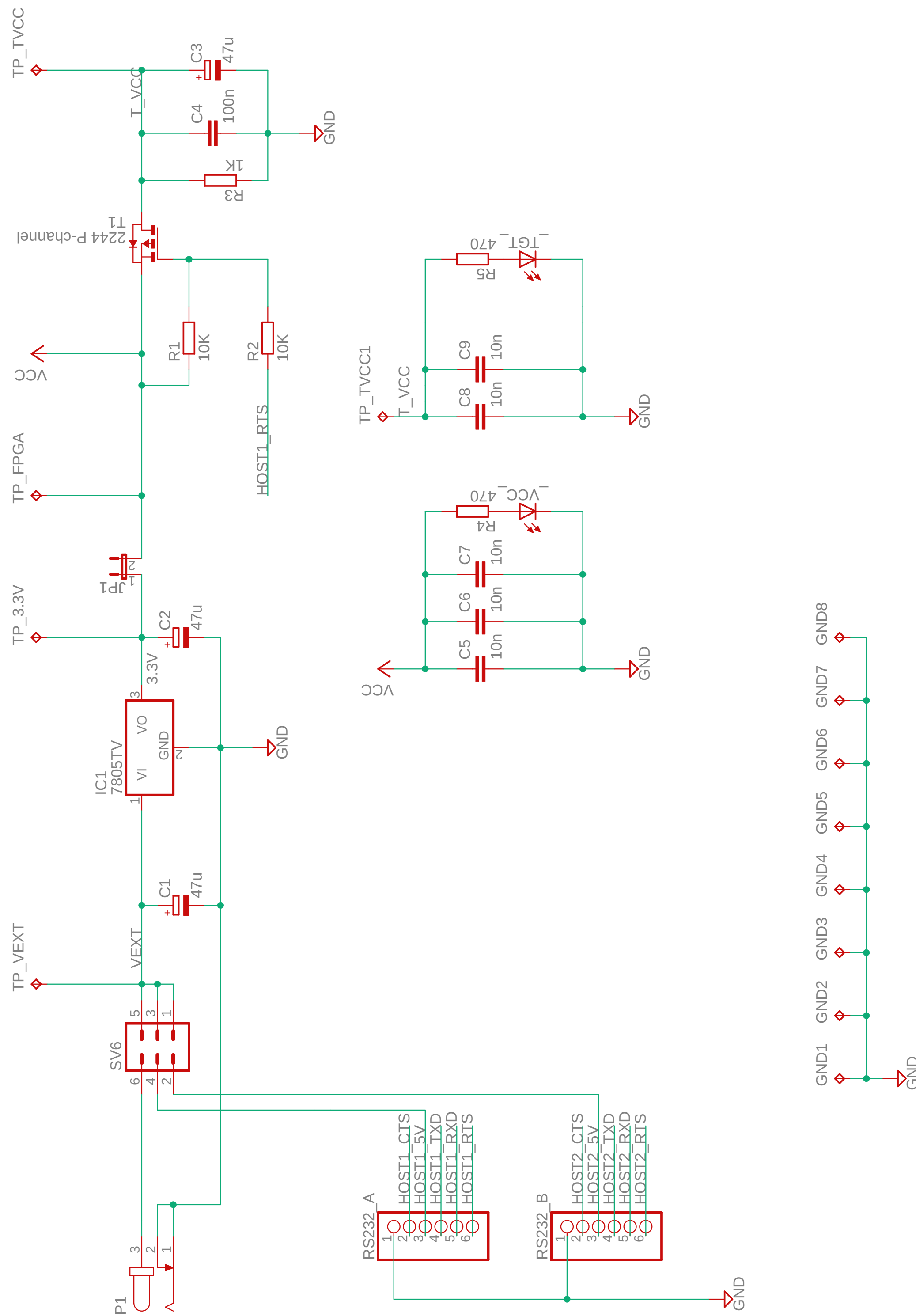


Figure B-7: Board FPGA Power Schematic

Figure B-8: Board FPGA DUT

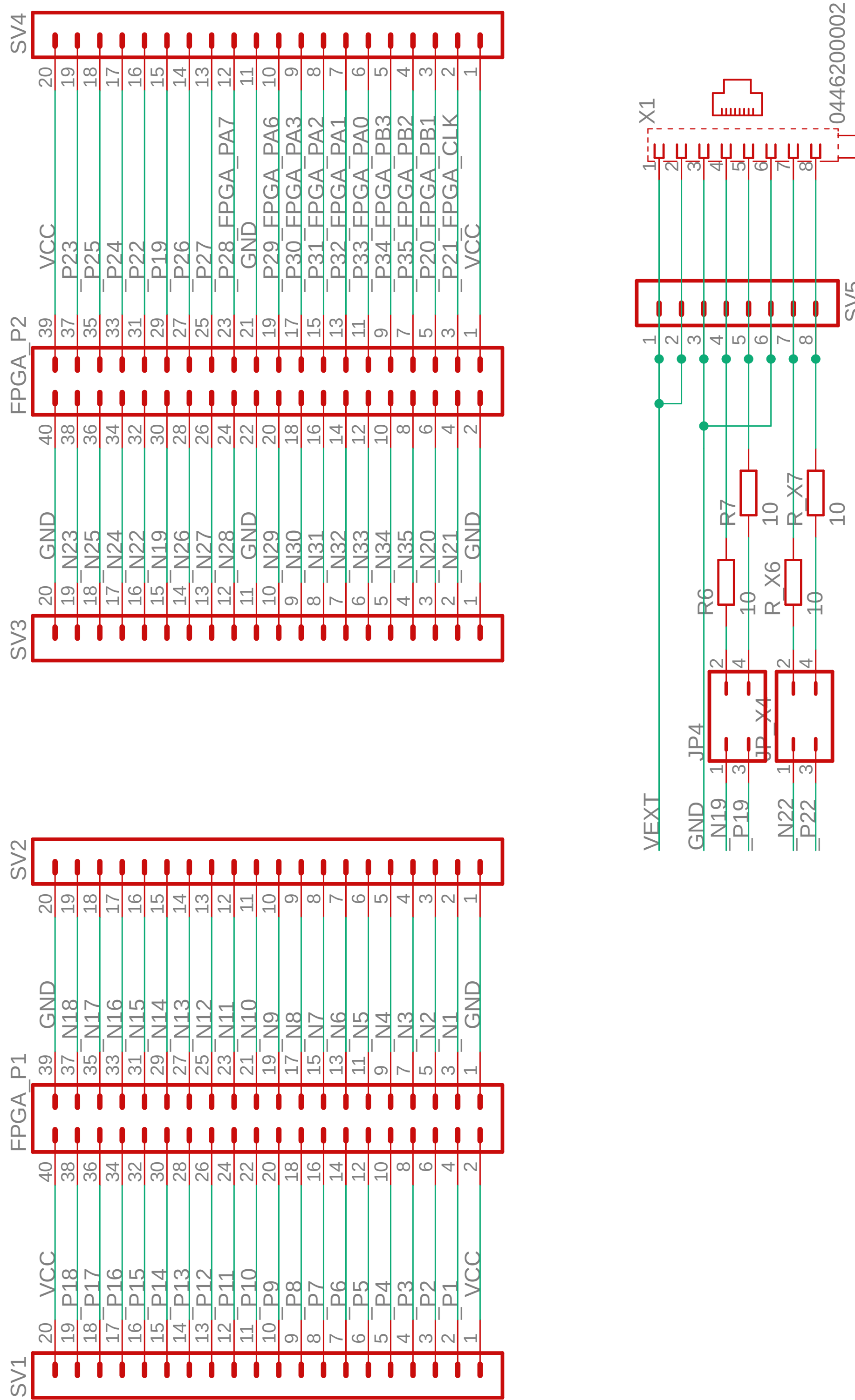


Figure B-9: Board FPGA Connections

B.4.1 FPGA program

The FPGA is programmed to support multiple timers. Each timer consists of two counters. One defines the delay before a laser entablement signal gets generated, the other defines the duration of that signal.

The counters for all the timers are connected in a chain to form an extended shift register. This configuration enables the pulse times for all the timers to be programmed into the device via a two-wire Clock & Data protocol. Given the simplicity of the task and the size of the supporting FPGA, the number of timers that can be implemented far exceeds the number required for any conceivable experiment.

Figure B-10 shows the FPGA operation within the development environment. 'DIN' & 'DCLK' are used to preset the timers. Values are clocked in to define the delay and duration for each timer.

A pulse on the 'GO' signal starts the counters. This pulse is generated by the DUT to indicate that it has started to execute a sample program. After the prescribed delays, the output signal 'p1' then exhibits two pulses, the duration of which had also been set earlier.

Within the FPGA, signal 'p1' is routed to the LVDS connection and on to directly control the laser diode.

The 'CLK' signal is generated internally and defines the rate at which the timers count down. The same signal, divided by four, drives the DUT ensuring the laser firing and program execution remain synchronised.

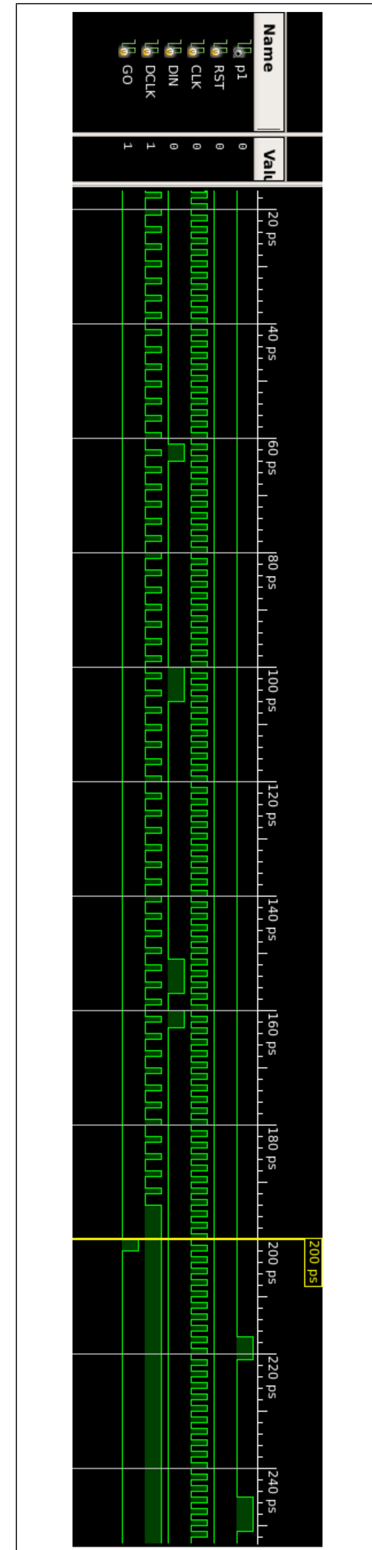


Figure B-10: Simulation

Appendix C

Defensive C-Compiler

Contents

C.1 Operation **303**

C.2 Samples **304**

C.2.1 Call & Return Defences 304

C.2.2 Branching Defences 307

C.2.3 Data Placement 310

We developed DCC, our Defensive C Compiler from K&R's YACC grammar defined "The C Programming Language (Ed.2)" [43]. Here we describe the program structure and source files. We also show how it is used, samples of the code it generates, and how to extend it.

The complete set of source code files for DCC can be found at `https://github.com/digitallocksmiths/DCC`. The application can be built with Microsoft Visual Studio *"Community Edition"*, and the directory structure of the files making up the DCC source code bundle is shown here in Figure C-1.

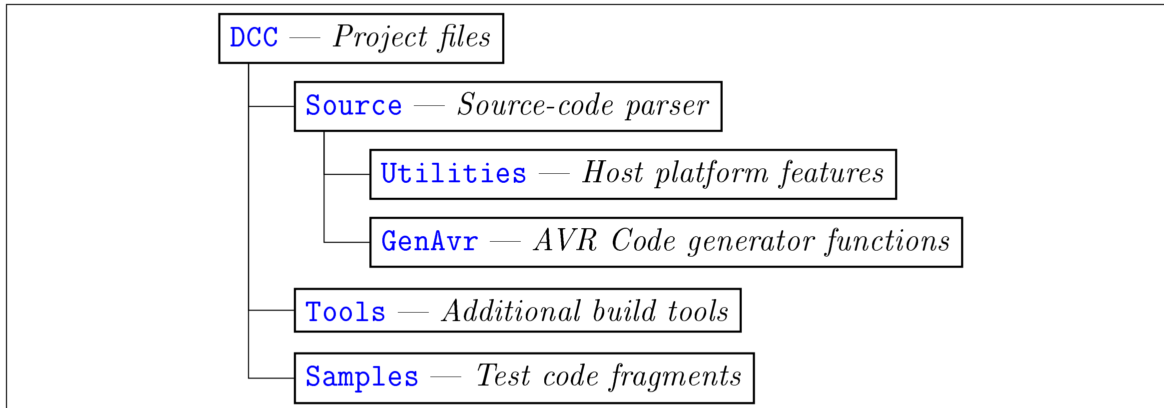


Figure C-1: DCC Project Directory Structure.

DCC — Visual Studio project files.

Source — Code for file I/O, input file parsing and IR generation. This code is independent from the targetted CPU.

Utilities — Funtions that access windows platform specific features. If the project is ported to an alternative host operating system, these files will probably need updating. Features such as console colour printing and application versioning are managed here.

GenAvr — The code generator for AVR. These files translate the IR into AVR assembler files. If an alternative target CPU is required, these files are the ones to modify.

Tools — Extra executables used in this projects. Yacc.Exe translates the 'C' YACC grammar into compilable source files.

Samples — Small snippits of parsable 'C' code, used to test the output of the code generator.

C.1 Operation

The compiler can be executed from a shell command line.

```
> CC_Avr <input file> <output file> <options>
```

<input file> — The C sourcecode file to be translated. In the absence of this parameter the default filename of Code.c will used.

<output file> — The filename for the translated output. In the absence of a this parameter the input file name will be used with the extension replaced with .Asm.

<options> — May be any number of the following.

-v Verbose. Prints information about the parameters being used.

-debug_output Echo the text being sent to the output file to <stdout>. This feature is useful when debugging the code generator. The output is instantaneous rather than cached and delayed, as happens with the target file updating.

-debug_expr Pretty-print to <stdout> each expression within the IR as it is being processed. This feature is useful when debugging the translator and when implementing expression transformations for optimization.

-Debug_objects Pretty-print to <stdout> each IR feature. This is, more or less, the parse tree of the C source code. It provides a useful reference for understanding the IR and is of particular value when implementing a code generator.

-d1=<int> Defence setting for Call & Return. -d1=0 is undefended. -d1=n adds the defence shown in Figure C-4. When n controls the number of consecutive "jmp SysTrap" statements to insert. Additional statements provide immunity from consecutive skip errors.

-d2=<int> Defence setting for unconditional jump statements. -d2=0 is undefended. -d2=n adds the prescribed number of "jmp SysTrap" statements after unconditional jump statements. Unconditional jumps typically occur at the end of a loop's body code to return to that loop's test condition, or to exit

from an `if`'s `then` clause by jumping over the `else` body code, as seen in lines 44..46 of Figure C-7.

`-d3=<int>` Defence setting for Conditional branch statements. `-d3=0` is undefended. `-d3=n` adds the prescribed number of `"jmp SysTrap"` statements after conditional branches. Conditional branches are implemented as specific branches on both the positive and negative test conditions with a fall through being trapped, as seen in lines 30..34 of Figure C-7.

C.2 Samples

One feature common to all of the compiled output is the extensive code commenting. Extensive commenting assists code reviewers by demonstrating the nature of defences and their coverage.

C.2.1 Call & Return Defences

The code shown in Figure C-2 demonstrates simple call and return behaviour. The output from an undefended compilation is shown in Figure C-3 and the defended equivalent in Figure C-4.

```
1   void Called(void) {
2       ;
3   }
4
5   void Caller(void) {
6       Called();
7   }
```

Figure C-2: Call & Return — Source Code

```
1
2   // Commandline parameters Samples\Code.c -d1=0
3
4                      ; ################################################################
5                      ; # Function: Called
6                      ; ################################################################
7                      ; #    Params: <none>
8                      ; #    Locals: <none>
9                      ; #    Return: void
10                     ; ################################################################
11                     .global    Called
12  Called:            ; #### no frame needed ############### ; no parameters or locals
13                     ; #### body text ##################### ;
14                     ; ==== Compound Statement ============ ; L_Statement_compound_000001
15                     ;      Compound Body -------            ; L_Statement_compound_000001
16                     ;      Compound End --------            ; L_Statement_compound_000001
17                     ; #### exit ########################## ;
18  L_Exit_000000:     ret                                     ;    4:
19
20
21
22                     ; ################################################################
23                     ; # Function: Caller
24                     ; ################################################################
25                     ; #    Params: <none>
26                     ; #    CPU   : <return addr>    @Y+3   size=2
27                     ; #    CPU   : <saved fptr>     @Y+1   size=2
28                     ; #    Locals: <none>
29                     ; #    Return: void
30                     ; ################################################################
31                     .global    Caller
32  Caller:            ; #### frame setup ################### ;
33                     push       r29                          ;    2: (YH) preserve Frame
34                     push       r28                          ;    2: (YL)
35                     in         r29, _IO_SPH                 ;    1: FP <- SP
36                     in         r28, _IO_SPL                 ;    1:
37                     ; #### body text ##################### ;
38                     ; ==== Compound Statement ============ ; L_Statement_compound_000003
39                     ;      Compound Body -------            ; L_Statement_compound_000003
40                     ; ==== Expr Statement ================ ; L_Statement_Expression_000004
41                     call       Called                       ;    3: Invoke the function
42                                                             ;     :
43                     ;      Expr end ------------            ; L_Statement_Expression_000004
44                     ;      Compound End --------            ; L_Statement_compound_000003
45                     ; #### exit ########################## ;
46  L_Exit_000002:     out        _IO_SPH, r29                 ;    1: SP <- Y
47                     out        _IO_SPL, r28                 ;    1:
48                     pop        r28                          ;    2: FP <- preserved Frame
49                     pop        r29                          ;    2:
50                     ret                                     ;    4:
```

Figure C-3: Call & Return — Undefended Output

```
1
2   // Commandline parameters Samples\Code.c -d1=1
3
4                       ; ##############################################################
5                       ; # Function: Called
6                       ; ##############################################################
7                       ; #   Params: <none>
8                       ; #   Locals: <none>
9                       ; #   Return: void
10                      ; ##############################################################
11                      .global   Called
12  Called:             ; #### no frame needed ############### ; no parameters or locals
13                      ; #### intended call check ########### ;
14                      subi      r16, lo8(Called-1)           ;   1:    Check the call was deliberate
15                      breq      L_OK_000001                  ;   2:    Clean setup, or
16                      jmp       SysTrap                      ;   2:    Accidental arrival?
17  L_OK_000001:        ; #### body text ##################### ;
18                      ; ==== Compound Statement ============ ; L_Statement_compound_000002
19                      ;      Compound Body -------           ; L_Statement_compound_000002
20                      ;      Compound End --------           ; L_Statement_compound_000002
21                      ; #### exit ########################## ;
22  L_Exit_000000:      ldi       r16, lo8(Called+1)           ;   1: Returner's identity
23                      ret                                    ;   4:
24
25
26
27                      ; ##############################################################
28                      ; # Function: Caller
29                      ; ##############################################################
30                      ; #   Params: <none>
31                      ; #   CPU   : <return addr>    @Y+3   size=2
32                      ; #   CPU   : <saved fptr>     @Y+1   size=2
33                      ; #   Locals: <none>
34                      ; #   Return: void
35                      ; ##############################################################
36                      .global   Caller
37  Caller:             ; #### frame setup ################## ;
38                      push      r29                          ;   2: (YH) preserve Frame
39                      push      r28                          ;   2: (YL)
40                      in        r29, _IO_SPH                 ;   1: FP <- SP
41                      in        r28, _IO_SPL                 ;   1:
42                      ; #### intended call check ########### ;
43                      subi      r16, lo8(Caller-1)           ;   1:    Check the call was deliberate
44                      breq      L_OK_000004                  ;   2:    Clean setup, or
45                      jmp       SysTrap                      ;   2:    Accidental arrival?
46  L_OK_000004:        ; #### body text ##################### ;
47                      ; ==== Compound Statement ============ ; L_Statement_compound_000005
48                      ;      Compound Body -------           ; L_Statement_compound_000005
49                      ; ==== Expr Statement ================ ; L_Statement_Expression_000006
50                      ldi       r16, lo8(Called-1)           ;   1: Defended call
51                      call      Called                       ;   3: Invoke the function
52                      subi      r16, lo8(Called+1)           ;   1:    Check who returned
53                      breq      L_OK_000007                  ;   2:    Expected returner?
54                      jmp       SysTrap                      ;   2:    Accept no substitutes.
55  L_OK_000007:                                               ;    :
56                      ;      Expr end ------------           ; L_Statement_Expression_000006
57                      ;      Compound End --------           ; L_Statement_compound_000005
58                      ; #### exit ########################## ;
59  L_Exit_000003:      ldi       r16, lo8(Caller+1)           ;   1: Returner's identity
60                      out       _IO_SPH, r29                 ;   1: SP <- Y
61                      out       _IO_SPL, r28                 ;   1:
62                      pop       r28                          ;   2: FP <- preserved Frame
63                      pop       r29                          ;   2:
64                      ret                                    ;   4:
```

Figure C-4: Call & Return — Defended Output

C.2.2 Branching Defences

The code shown in Figure C-5 demonstrates defended branch behaviour. The output from an undefended compilation is shown in Figure C-6 and the defended equivalent in Figure C-7.

```
1 void Test(char  x, char y) {
2         char z;
3         if (y) {
4             z=x;
5         }
6         else {
7             z=y;
8         }
9     }
```

Figure C-5: Branching Test — Source Code

```
1
2  // Commandline parameters Samples\Code.c -d2=0 -d3=0
3
4                    ; ###############################################################
5                    ; # Function: Test
6                    ; ###############################################################
7                    ; #    Param: y                  @Y+7   size=1 char , Class[_____]
8                    ; #   CPU   : <return address>  @Y+4   size=3
9                    ; #         : <saved fptr>      @Y+2   size=2
10                   ; #    Param: x                  r2     size=1 char , Class[_____]
11                   ; #   Locals: z                  @Y+1   size=1 char , Class[_____]
12                   ; #         : sizeof(locals) = 1
13                   ; #   Return: void
14                   ; ###############################################################
15                   .global   Test
16 Test:             ; #### frame setup ##################  ;
17                   push      r29                          ;   2: (YH) preserve Frame
18                   push      r28                          ;   2: (YL)
19                   in        r29, _IO_SPH                 ;   1: FP <- SP
20                   in        r28, _IO_SPL                 ;   1:
21                   sbiw      r28, 1                       ;   2: FP -= sizeof(locals))
22                   out       _IO_SPH, r29                 ;   1: SP <- FP
23                   out       _IO_SPL, r28                 ;   1:
24                   mov       r2, r20                      ;   1: First parameter is regified
25                   ; #### body text ####################  ;
26                   ; ==== Compound Statement ============ ; L_Statement_compound_000001
27                   ;      Compound Body -------           ; L_Statement_compound_000001
28                   ; ==== If Statement ================= ; L_Statement_If_000002
29                   ldd       r20, Y+7                     ;   1: reg8 <== Local y @Frame:7
30                   tst       r20                          ;   1: Cast Byte to Flag
31                   breq      L_IfElse_000004              ; 1/2: branch if FALSE
32                                                          ;    : fall through on TRUE
33 L_IfThen_000003:  ;      If Then ------------            ; L_Statement_If_000002
34                   ; ==== Compound Statement ============ ; L_Statement_compound_000006
35                   ;      Compound Body -------           ; L_Statement_compound_000006
36                   ; ==== Expr Statement ================ ; L_Statement_Expression_000007
37                   mov       r20, r2                      ;   1: reg8 <== register variable x
38                   std       Y+1, r20                     ;   2: local <== reg z
39                   ;      Expr end ------------           ; L_Statement_Expression_000007
40                   ;      Compound End --------           ; L_Statement_compound_000006
41                   rjmp      L_IfDone_000005              ; 1/2:
42 L_IfElse_000004:  ;      If Else ------------            ; L_Statement_If_000002
43                   ; ==== Compound Statement ============ ; L_Statement_compound_000008
44                   ;      Compound Body -------           ; L_Statement_compound_000008
45                   ; ==== Expr Statement ================ ; L_Statement_Expression_000009
46                   ldd       r20, Y+7                     ;   1: reg8 <== Local y @Frame:7
47                   std       Y+1, r20                     ;   2: local <== reg z
48                   ;      Expr end ------------           ; L_Statement_Expression_000009
49                   ;      Compound End --------           ; L_Statement_compound_000008
50 L_IfDone_000005:  ;      If End --------------           ; L_Statement_If_000002
51                   ;      Compound End --------           ; L_Statement_compound_000001
52                   ; #### exit ########################## ;
53 L_Exit_000000:    adiw      r28, 1                       ;   2: Y += sizeof(locals)
54                   out       _IO_SPH, r29                 ;   1: SP <- Y
55                   out       _IO_SPL, r28                 ;   1:
56                   pop       r28                          ;   2: FP <- preserved Frame
57                   pop       r29                          ;   2:
58                   ret                                    ;   4:
```

Figure C-6: Branching Test — Undefended Output

```
1
2  // Commandline parameters Samples\Code.c -d2=2 -d3=2
3
4                      ; ################################################################
5                      ; # Function: Test
6                      ; ################################################################
7                      ; #    Param: y                 @Y+7    size=1 char , Class[_____]
8                      ; #   CPU    : <return address> @Y+4    size=3
9                      ; #          : <saved fptr>     @Y+2    size=2
10                     ; #    Param: x                 r2      size=1 char , Class[_____]
11                     ; #   Locals: z                 @Y+1    size=1 char , Class[_____]
12                     ; #          : sizeof(locals) = 1
13                     ; #   Return: void
14                     ; ################################################################
15                     .global   Test
16 Test:               ; #### frame setup ################## ;
17                     push      r29                          ;   2: (YH) preserve Frame
18                     push      r28                          ;   2: (YL)
19                     in        r29, _IO_SPH                 ;   1: FP <- SP
20                     in        r28, _IO_SPL                 ;   1:
21                     sbiw      r28, 1                       ;   2: FP -= sizeof(locals))
22                     out       _IO_SPH, r29                 ;   1: SP <- FP
23                     out       _IO_SPL, r28                 ;   1:
24                     mov       r2, r20                      ;   1: First parameter is regified
25                     ; #### body text #################### ;
26                     ; ==== Compound Statement ============ ; L_Statement_compound_000001
27                     ;      Compound Body -------          ; L_Statement_compound_000001
28                     ; ==== If Statement ================== ; L_Statement_If_000002
29                     ldd       r20, Y+7                     ;   1: reg8 <== Local y @Frame:7
30                     tst       r20                          ;   1: Cast Byte to Flag
31                     breq      L_IfElse_000004              ; 1/2: branch if FALSE
32                     brne      L_Dft_000006                 ; 1/2: not-breq defence
33                     jmp       SysTrap                      ;    : Defence against skipping
34                     jmp       SysTrap                      ;    :
35 L_Dft_000006:                                               ;    : defended fall through.
36 L_IfThen_000003: ;      If Then ------------               ; L_Statement_If_000002
37                     ; ==== Compound Statement ============ ; L_Statement_compound_000007
38                     ;      Compound Body -------          ; L_Statement_compound_000007
39                     ; ==== Expr Statement ================ ; L_Statement_Expression_000008
40                     mov       r20, r2                      ;   1: reg8 <== register variable x
41                     std       Y+1, r20                     ;   2: local <== reg z
42                     ;      Expr end ------------           ; L_Statement_Expression_000008
43                     ;      Compound End --------           ; L_Statement_compound_000007
44                     rjmp      L_IfDone_000005              ; 1/2:
45                     jmp       SysTrap                      ;    : Defence against skipping
46                     jmp       SysTrap                      ;    :
47 L_IfElse_000004: ;      If Else ------------               ; L_Statement_If_000002
48                     ; ==== Compound Statement ============ ; L_Statement_compound_000009
49                     ;      Compound Body -------          ; L_Statement_compound_000009
50                     ; ==== Expr Statement ================ ; L_Statement_Expression_000010
51                     ldd       r20, Y+7                     ;   1: reg8 <== Local y @Frame:7
52                     std       Y+1, r20                     ;   2: local <== reg z
53                     ;      Expr end ------------           ; L_Statement_Expression_000010
54                     ;      Compound End --------           ; L_Statement_compound_000009
55 L_IfDone_000005: ;      If End --------------              ; L_Statement_If_000002
56                     ;      Compound End --------           ; L_Statement_compound_000001
57                     ; #### exit ######################### ;
58 L_Exit_000000:      adiw      r28, 1                       ;   2: Y += sizeof(locals)
59                     out       _IO_SPH, r29                 ;   1: SP <- Y
60                     out       _IO_SPL, r28                 ;   1:
61                     pop       r28                          ;   2: FP <- preserved Frame
62                     pop       r29                          ;   2:
63                     ret                                    ;   4:
```

Figure C-7: Branching Test — Defended Output

C.2.3 Data Placement

An area of interest for code reviewers is the location and initialization states of global data resources. Figure C-8 shows various global variables and Figure C-9 shows the resulting compiled output. Preservation of variable names and annotations for elements of structures enables reviewers to quickly confirm where variables are allocated, as well as if and how they are initialized.

```
1    int a;                // uninitialized var
2    int b = 1;            // initialized var
3
4    typedef struct { char cF; int iF; } S;
5
6    S   s1;               // uninitialized structure
7    S   s2 = { 2, 3};     // initialized structure
```

Figure C-8: Variable Declaration — Source Code

```
1
2                                           ; ########################
3                                           ; # Start of INIT segment #
4               .section    INIT,  "a"      ; ########################
5                                           ; This is the source of the initialization data
6   ;:          b                           ; (int)
7               .word      1                ;         0x0001 -> b
8   ;:          s2                          ; structure ''
9               .byte      2                ;           0x02 -> s2.cF
10              .word      3                ;         0x0003 -> s2.iF
11
12                                          ; ########################
13              .section    .data           ; # Start of DATA segment #
14                                          ; ########################
15                                          ; This is where the initialization data will be copied to
16              .global    b
17  b:          .space        2             ; (int)
18              .global    s2
19  s2:         .space        3             ; structure ''
20
21                                          ; #######################
22              .section    .bss            ; # Start of BSS segment #
23                                          ; #######################
24                                          ; Here starts the uninitialized data
25              .global    a
26  a:          .space        2             ; (int)
27              .global    s1
28  s1:         .space        3             ; structure ''
```

Figure C-9: Variable Declaration — Output